Benchmarking and Evaluation of Blockchain Systems and Applications

Amjad Aldweesh
School of Computing
Newcastle University

A thesis submitted for the degree of
Doctor of Philosophy

March 4, 2020

Dedication

This work is dedicated to my family and my friends.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, foot notes,tables and equations and has fewer than 150 figures.

Amjad Aldweesh
March 4, 2020

Acknowledgements

First of all, praises and thanks to Allah, who has granted me countless blessings, the patience and perseverance during this research project, and indeed, throughout all my life. I gratefully acknowledge the deepest gratitude to my supervisor, prof. Aad van Moorsel, for his great guidance and wise supervision. Without his grate continuing support and encouragement, this research would never be a reality. In addition, I would like to thank my family, friends and indeed my wife who gave me all support I needed during this research

Abstract

In the past ten years, we have witnessed the evolution of cryptocurrencies. With market capitalizations of $189bn and $19bn respectively in September 2019, Bitcoin and Ethereum are the world's most successful cryptocurrencies. Blockchain, which is a public ledger that is immutable, is the main innovation behind these cryptocurrencies. In Bitcoin, the blockchain is introduced to exchange and trade a single asset, whereas in Ethereum it is used to store and execute a smart contract with a Turing Complete Machine.

Ethereum's Gas mechanism, which charges the execution of each operation code, ensures the termination of smart contracts that run in the EVM (Ethereum Virtual Machine) and to compensate the computational usage. Thus, the gas awarded should be proportional to the required computational, to ensure aligned incentives and to avoid denial of services attacks. Currently, in Ethereum, gas awarded is set statically for each opcode in the smart contract, but it is unknown whether these values are correct for various computer architectures.

Therefore, in this thesis, firstly, we propose a benchmark approach to measure the CPU times required to deploy and execute real smart contracts obtained from the Ethereum blockchain and compare it with the gas award in the PyEthApp client running over a single machine. The result of our benchmark study shows the collected Gas is not always proportional to the invested CPU for both deploying and executing smart contracts.

Secondly, we focus more in-depth on the operational codes (opcodes) and conduct a benchmark study to investigate whether the Gas cost set by Ethereum for each opcode is aligned with the CPU usage. The experiments are conducted on three Ethereum clients running over different hardware platforms and operating systems. The results show that the Gas cost is not always proportional to CPU usage.

Finally, we implement and analyze the performance of blockchain and smart contract technologies in different domains, in particular cloud computing and distributed database management systems. In cloud computing, we create the first smart contract implementation that achieves both verifiability and cost-efficiency using any two client providers. For distributed database management systems, we implement the first smart contract-based two-phase commit protocol in Ethereum's blockchain. For both systems, we investigate the cost, the performance and trade-offs in the blockchain. We tested the implementations of these systems on Ethereum's official, test and private networks. We also provide a financial and computational analysis of their costs.

Contents

1 Introduction **1**

1.1 Research Questions 2
1.2 Contributions 4
1.3 Thesis Structure 5
1.4 Publications 6

2 Background: Blockchain and Benchmarking **9**

2.1 Summary 9
2.2 Blockchain Overview 9
2.3 Bitcoin 11
2.3.1 Bitcoin Address 12
2.3.2 Bitcoin Transaction 12
2.3.3 Proof-of-Work (PoW) Algorithm 14
2.3.4 Bitcoin Mining 14
2.3.5 Applications of Bitcoin 14
2.4 Ethereum 15
2.4.1 Smart Contracts 17
2.4.2 Ethereum Virtual Machine (EVM) 19
2.4.3 Ethereum Layers 19
2.4.4 Ethereum Execution Model 20
2.4.5 Ethereum Clients 21
2.4.6 Ethereum Peer-to-Peer Network 22
2.4.7 Ethereum Mining 23
2.4.8 Ethereum Gas Mechanism 24
2.4.9 Solidity 25
2.5 Benchmarks 25
2.5.1 Definition for Benchmarks 25
2.5.2 Benchmarks Requirements 26

2.5.3 Classifications of Benchmarks . . . 27
2.5.4 Workload and Metrics . . . 27
2.6 Conclusion . . . 28

3 Performance Benchmark of Blockchain Smart Contracts 29
3.1 Summary . . . 29
3.2 Introduction . . . 30
3.3 Smart Contracts Selection . . . 31
3.4 Design of Benchmark Measurement System . . . 33
3.5 Results and Discussion . . . 34
3.5.1 Contract Creation Transactions . . . 34
3.5.2 Function Execution Transactions . . . 35
3.5.3 Comparison Between Contract Creation and Function Execution Transactions . . . 38
3.6 Conclusion . . . 39

4 Performance Benchmark of Blockchain Smart Contract Operation Code (Opcode) 41
4.1 Summary . . . 41
4.2 Introduction . . . 42
4.3 Related Work . . . 43
4.4 Design of OpBench System . . . 44
4.4.1 OpBench Overview . . . 45
4.4.2 Workload: Classification of Opcodes . . . 47
4.4.2.1 Computation-based Opcodes . . . 48
4.4.2.2 Formula-based Opcodes . . . 48
4.4.3 Manipulating the Stack . . . 50
4.5 Implementation . . . 52
4.5.1 PyEthApp . . . 52
4.5.2 Go-Ethereum . . . 54
4.5.3 Parity . . . 55
4.6 Conclusion . . . 56

5 Experimental Results and Discussion 57
5.1 Summary . . . 57
5.2 First Experimental Results . . . 58
5.2.1 Absolute CPU time . . . 60

5.2.2 Relative CPU time . . . 61
5.2.3 Absolute gas/CPU . . . 62
5.2.4 Normalized gas/CPU . . . 63
5.3 Second Experimental Results . . . 64
5.3.1 Comparison of Platforms Absolute CPU Time . . . 66
5.3.2 Sensitivity of Platform Speed to Relative CPU Time . . . 68
5.3.3 Platforms Comparison for Gas/CPU Ratio . . . 69
5.3.4 Comparison of Clients Absolute CPU Time . . . 73
5.3.5 Comparison of Different Operating Systems . . . 76
5.4 Result Validation . . . 78
5.4.1 Evaluation of Individual Opcodes and Composed Complete Contract . . . 79
5.4.2 Evaluating the Overhead Effects of the POP Opcode and the Timing . . . 80
5.5 Conclusion . . . 81

6 Implementation and Evaluation of Counter-Collusion Smart Contracts for Verifiable Cloud Computing 83
6.1 Summary . . . 83
6.2 Introduction . . . 84
6.3 Adversary Model and Assumptions . . . 88
6.4 Monetary Variables . . . 90
6.5 The Prisoner's Contract . . . 91
6.5.1 The contract . . . 91
6.6 The Colluder's Contract . . . 93
6.6.1 The Contract . . . 93
6.7 The Traitor's Contract . . . 95
6.7.1 The Contract . . . 95
6.8 Address and Pseudocode . . . 98
6.8.1 Contract Account Address . . . 98
6.8.2 Prisoner's Contract . . . 98
6.8.3 Colluder's Contract . . . 100
6.8.4 Traitor's Contract . . . 100
6.9 Implementation . . . 102
6.10 Overhead and Cost . . . 102
6.11 Conclusion . . . 103

7 Implementation and Evaluation of Non-Blocking Two Phase Commit Protocol Using Blockchain **105**
7.1 Summary . . . 105
7.2 Introduction . . . 106
7.3 The Atomic Commit Problem . . . 108
7.3.1 Synchronous *vs* Asynchronous Systems . . . 109
7.3.2 Synchronous *vs* Asynchronous Blockchains . . . 110
7.4 2PC in Synchronous Systems . . . 111
7.4.1 Inevitability of Blocking in 2PC . . . 113
7.5 Non-Blocking with Blockchain . . . 114
7.5.1 Approach . . . 114
7.5.2 Synchronous Blockchain . . . 115
7.5.3 2PC with Synchronous Blockchain . . . 115
7.5.3.1 Protocol 1 . . . 116
7.5.3.2 Protocol 2 . . . 117
7.5.4 Smart Contract Pseudo Code . . . 120
7.5.4.1 Protocol 1 Pseudo Code . . . 120
7.5.4.2 Protocol 2 Pseudo Code . . . 121
7.6 Asynchrony & Impossibilities . . . 121
7.6.1 Implications of Synchrony Violations . . . 123
7.7 Implementation and Evaluation . . . 124
7.7.1 Delay Bound Estimation . . . 125
7.7.2 Cost of 2PC Coordination . . . 127
7.7.3 2PC Execution Latencies . . . 129
7.7.3.1 Estimated Latency Bound . . . 129
7.7.3.2 Observed Latencies . . . 130
7.7.4 Impact of Synchrony Violations on Commit-Validity . . . 131
7.8 Conclusion . . . 133

8 Conclusion and Future Work **135**
8.1 Summary . . . 135
8.2 Future Work . . . 136

Bibliography **138**

List of Figures

2.1 The structure of a block in the blockchain. 10
2.2 Bitcoin Transaction Overview. 13
2.3 Schematic representation of PoW. 13
2.4 Ethereum's contract creation and call-contract transactions [72]. . . . 15
2.5 How Ethereums users interact with the smart contract on the blockchain [28]. 17
2.6 The Execution model of the Ethereum Virtual Machine's (EVM) parts and their interactions. 18
2.7 The basic architecture of the Ethereum system and where the Ethereum Virtual Machine (EVM) fits into the system. 20
2.8 Simulating the addition operation of two numbers on the EVM. . . . 21
2.9 Ethereum's block header and transactions [72]. 23

3.1 Amount of used gas awarded per each microsecond of CPU usage for contract creation. 35
3.2 Amount of used gas awarded per each microsecond of CPU usage for function execution. 36

4.1 OpBench overview. 45
4.2 Utilizing the `POP` opcode to overcome the stack size limitation. 52

5.1 CPU time (in μs) for each opcode on a logarithmic scale. 60
5.2 CPU time for each opcode, relative to the fastest platform (Windows Go-Ethereum 3.6GHz). 61
5.3 Used Gas (per [94]) per CPU time unit (in Gas/μs). Reward and cost are proportional for a platform if the lines are straight. 62
5.4 Normalized Used Gas per CPU time unit (results in 5.7 divided by the platform's result for opcode `Byte`). Reward and cost are proportional if the lines are straight at value 1. 63
5.5 CPU time (in microseconds) for each opcode. 67

5.6 CPU time for each opcode, relative to the fastest platform (Windows Parity 3.5GHz). 69
5.7 Used Gas (per [94]) per CPU time unit (in Gas/microsecond). Reward and cost are proportional for a platform if the curve is a straight line. 71
5.8 Normalized Used Gas per CPU time unit (results in 5.7 divided by the platform's result for opcode `BYTE`). Reward and cost are proportional if the curve is a straight line at value 1. 72
5.9 CPU time (in microseconds) for each opcode in the PyEthApp client. 75
5.10 CPU time (in microseconds) for each opcode in the Go-Ethereum client. 75
5.11 CPU time (in microseconds) for each opcode in the Parity client. . . . 76
5.12 Absolute CPU time for Windows machine for all clients. 77
5.13 Absolute CPU time for Linux machine for all clients. 78
5.14 Absolute CPU time for Windows and Linux machines for all clients. . 78

7.1 Two phase commit protocol. 111
7.2 2PC State Transition Diagram for Process P_i. 113
7.3 State Diagram for 2PC with Blockchain. 119
7.4 Smart Contract pseudo-code for 2PC coordination protocol 1. 122
7.5 Pseudo-code for 2PC coordination smart contract protocol 2. 122
7.6 Block awareness delay. 126
7.7 Block entry delay. 126
7.8 Transmission delay. 127
7.9 Probability for commit-validity. 132

List of Tables

2.1 Comparison between public and private blockchains. 11
2.2 EVM client implementations. 22

3.1 The most profitable and expensive contract creation transactions. . . 35
3.2 The average execution time (in microsecond) and the amount of used gas for all function execution transactions. 37

4.1 List of all operation codes (opcodes) in the Ethereum Virtual Machine [94]. 51

5.1 Experimental 1 platforms. 58
5.2 The average CPU time for each of the opcodes for all platforms in μs. The right-most column provides the Used Gas. 59
5.3 Experimental 2 platforms. 64
5.4 The average CPU time for each of the opcodes for all platforms, by category. The right-most column provides the Used Gas with the opcode. All results in (μs). 66
5.5 Comparison of CPU time (in microseconds) between different clients and operating system for selected slowest opcodes. 68
5.6 Selected opcodes where other clients outperform the Windows Parity clients. 70
5.7 Selected opcodes by highest Gas per CPU on the three clients. 73
5.8 Comparison between the PyEthApp, the Go-Ethereum, and the Parity clients on formula-based opcodes. 74
5.9 The CPU time for functions execution in (second) and each selected opcode in (microsecond). 80

6.1 Cost of using the smart contracts on the official Ethereum network. The transactions are viewable on the blockchain. 103

7.1 Cost of executing 2PC-Blockchain contracts. 128

7.2 Total Cost in Various Voting Scenarios. 128
7.3 Minimum (Min), Maximum (Max) and Average (Avg) Latency in Minutes (Mn) and Seconds (Ss) expressed as Mn:Ss. 131

Chapter 1

Introduction

Blockchain and cryptocurrencies have gained considerable popularity in recent years. Blockchain is a decentralized network of peers that provides the infrastructure to keep tracking of a public ledger is the idea of the cryptocurrencies. A public ledger stores and maintains all transactions of the network. The blockchain format and a consensus protocol are utilized to store the ledger and to maintain the state of the blockchain.

Cryptocurrencies, as the name indicates, use the concept of cryptography to both secure transactions and to manage the creation of currency units. A smart contract is a piece of a computer program that is built on cryptocurrencies platforms to allow defining and executing terms (i.e. contracts) on the blockchain. The execution of the smart contract is triggered by a transaction added to the blockchain. Peers who maintain the blockchain execute the code and the consensus protocol achieves the correctness of the execution. In general, a smart contract can be seen as a program run by a global computer that will honestly execute each instruction of the code.

One of the leading cryptocurrencies and platforms used to deploy and execute a smart contract is Ethereum [94]. At the time of writing, Ethereum is the second largest blockchain behind Bitcoin. Ether is the nomination of the Ethereum cryptocurrency. Ether can be transferred and held in and moved between accounts. In Ethereum, there are two types of accounts. First, an *externally owned account* is associated with a public-private key pair. It has an Ether balance and is owned by someone who has the private key used to sign a transaction from the accounts. The externally owned account has no associated code. Second, a *contract account*, which has no private key. It also has an Ether balance and its associated code. The associated code is triggered by call-contract transactions sent by an externally owned account or a contract account.

Each transaction is constructed and signed cryptographically by an account and it has two address fields; sender and receiver. A smart contract is *deployed* into the blockchain by sending a transaction with blank receiver address and code added to the data field. A transaction can *invoke* any function in a contract. In this a case, the receiver field contains the contract address, and the function name and the data field contains the function's arguments.

A transaction also includes a Gas and a Gas price. The code execution is not free; it consumes Gas and Gas are converted to a cryptocurrency using the Gas price. The account of the sender of the transaction is charged an amount of cryptocurrency as a transaction fee. Thus, in order to hold the continuity of the blockchain, transactions' Gas should be proportional to the CPU invested by miners who validate and run transactions and so the instructions of the contract.

The proportionality in smart contract ecosystem is important to ensure avoiding a potential denial-of-service attack and to compensated miners who run the blockchain fairly. Denial-of-service attacks in smart contact ecosystem were first identified in [19] exploiting the fact that an operation code (i.e., `EXTCODESIZE`) has a fairly low Gas cost and requires a very long time to be executed. The attacker initiated a smart contract that calls this operation code roughly 50k times. As a result, miners who run and validate the block that contains this smart contract need to spend a very long time to process this block. Even though the fee price of this operation code has been modified, it is still unknown if other operation codes are mispriced or not.

Moreover, to ensure the continuity of the blockchain, the Gas cost should be proportional to the invested CPU by the miners. Nowadays, almost all cryptocurrencies rely on the Proof-of-Work (PoW) as a consensus mechanism that requires a high amount of CPU overhead. Miners who solve the PoW puzzle and create new blocks are compensated by a fixed amount of currency and the transaction fees. In the future PoW is likely to be replaced by other mechanism, such as PoS (Proof-of-Stake) [60]. In that case, the computational effort shifts toward the creation, execution, and validation of smart contracts. It is yet more important that fees are proportional to the CPU overhead, otherwise there is no incentive to operate the blockchain correctly.

1.1 Research Questions

Based on the reasoning on the previous section, we want to assess the whether miners are correctly reimbursed for the computation time they spend. We address the following research questions.

Q1. How to determine the computation time of software code (and smart contracts in particular)?

We find that the current estimation of the computation fees on almost all smart contracts platforms, such as Ethereum, is not accurate relative to the invested CPU. To determine the computation time, we need to consider multiple factors. Machine dependence is one factor that should be considered, since we cannot know the specification of miners machines. An additional factor is the code termination problem, which means we cannot proof the termination of software code in advance, so we need to limit our investigation to subsets of programs, function levels, for example. Other specifics of the program might affect the computation time. E.g., in smart contracts systems, a contract could invoke another contract, which might increase the computation time. Our research will take into account some of these challenges and propose a suitable solution.

Q2. Can we build a benchmark system to assess the fee against CPU?

Evaluation of approaches to estimating the computation time of smart contracts can be done at two levels. First, and most obvious, one can consider if the estimate is close to the actual computation time. Secondly, one can evaluate with respect to the operation of the overall system: that is, does the improved estimate of the computation time make the system operate better (e.g., in a blockchain, will the operation be fairer and less prone to misuse?).

For both of the above evaluation targets, we envisage using a measurement study. The research will need to develop a sound manner to evaluate the execution time of the contracts and opcodes. In particular, we may need to develop a benchmark set of code. For the benchmark code set, we need to establish whether this is done at the level of smart contracts or the level of instructions (opcodes). To evaluate the impact of accurate estimation of computational effort on the overall operation of the system, we need to consider the blockchain architecture and identify typical user patterns, e.g., where a block might have multiple contracts executing requests which could affect the result.

Q3. Are the fee costs for smart contract's instructions reasonable and proportional to the invested CPU time?

In Ethereum, when executing a smart contract, the fee a miner receives is determined by the *Gas* required to execute operation code (opcode), multiplied by a price the submitter of a transaction pays per unit of gas. The Ethereum client (more precisely, the Ethereum Virtual Machine (EVM)) tallies the total gas as it executes a smart contract based on values specified in the Ethereum yellow paper [94], which

statically associates an amount of gas to each opcode. That means that the gas used (and therefore the fee received) per smart contract is independent of the hardware or software used by the client; it directly follows from the opcodes in the smart contract.

From the miners' perspective, there are several implications of this static approach. The cost of executing smart contracts can be expected to be different on different computing platforms since the execution time of individual opcodes is likely to be different across platforms. As a consequence, a miner would want to choose a platform that optimizes the reward for the used energy. The benchmark should be designed, when carried out for different platforms, will *help select the best platform.*

1.2 Contributions

The work carried out in this PhD research makes a number of contributions to the subject of smart contracts:

- Conducting the first experiment to benchmark the CPU performance of real smart contracts on the Ethereum network.
- Designing, developing and implementing a framework that benchmarks the performance of the Ethereum opcodes on multiple client software and multiple operating systems and hardware.
- Conducting the first experiment that benchmarks the performance of the Ethereum opcodes on different Ethereum clients.
    1. We propose OpBench as the first CPU performance benchmarking system for Ethereum smart contract operation codes.
    2. We present a design framework for our proposed OpBench system, which is independent of the client's language and can be implemented in any language.
    3. We conduct experiments by implementing a proof-of-concept of our system for three different clients: PyEthApp, Go-Ethereum and Parity.
    4. We report the results of our implementations for three clients.
- Implementing and conducting a performance evaluation of a blockchain-based solution that achieves verifiability, high performance and cost-efficiency in the cloud computing system.

- Implementing and conducting an evaluation of the 2PC protocol over the blockchain and present the cost, impossibilities, the possibilities, and the trade-offs in this blockchain-based approach to blocking-free management of distributed transactions.

1.3 Thesis Structure

Chapter 1 shows the motivation behind the work in this thesis and highlights the main contributions of the research. Finally, we describe the related peer-reviewed publications produced throughout the research our of the PhD.

Chapter 2 presents technical background material closely related to the work carried out in the chapters of this thesis. We highlight the technical background behind blockchain technology such as smart contract and its popular platform, Ethereum and cryptocurrency such as Bitcoin. Additionally, it provides a discussion of the concept of benchmarking in the computer science field and its types.

Chapter 3 presents the first experiment that compares the CPU time to execute a contract against the gas rewarded for miners. In this Chapter, we explore how the incentive mechanism in the Ethereum system could impact miners decision on selecting transactions in their block. This is the first performance benchmarking conducted in the smart contract level.

Chapter 4 shows the first performance benchmarking design and implementation for the smart contract *opcodes*. To the best of our knowledge, this is the first benchmarking to be conducted at the opcodes level. It also outlines the approach we adopted to design and implement the benchmark performance of almost all opcodes of the Ethereum in three different clients namely Python, Go and Parity over three operating systems (i.e., Mac, Windows and Linux). This chapter includes a details design framework and implementations of these clients.

Chapter 5 extends the previous chapter and presents two set of experimental results. It investigates the correlation between the invested CPU time and the fee collected for the execution of the smart contract at the opcodes level. The experimental results are conducted on different clients, machines and operating systems.

Chapter 6 leverages the smart contract system and blockchain technology to propose a smart contract based solution that achieves verifiability at a reasonable cost. Briefly, a client lets two clouds compute the same task, and uses smart contracts to stimulate tension, betrayal and distrust between the clouds, so that rational clouds will not collude and cheat. we prove that the contracts will be effective under certain reasonable assumptions. By resorting to smart contracts, we are able to avoid heavy cryptographic protocols. The client only needs to pay two clouds to compute in the clear, and a small transaction fee to use the smart contracts. We also conducted a feasibility study that involves implementing the contracts in Solidity and running them on the official Ethereum network.

Chapter 7 investigates possibility, impossibility, the cost, the performance and the trade-offs in 2PC using a blockchain that supports execution of user-defined smart contracts. It demonstrates that the 2PC blocking can be eliminated at a moderate financial cost, if the blockchain also meets the synchrony requirements. Otherwise, despite the blockchain being a reliable state-machine, eliminating 2PC blocking may well be impossible, depending on whether the cluster hosting the database is synchronous or not. It implements the 2PC protocol in solidity and tests it on both the Ethereum private and test networks.

Chapter 8 summarizes the conclusions of the work presented in this thesis and motivates future directions for work in the area.

1.4 Publications

The work that has been presented in this thesis includes a list of co-authored publications. Chapters 4, 5 and 6 reflect papers that we published in international conferences and Chapter 7 reflects a paper we published in a journal. A list of these publications is provided below:

- **A Survey about Blockchain Software Architectures**, A Aldweesh, A van Moorsel, in 32nd Annual UK Performance Engineering Workshop & Cyber Security Workshop-2016 [7].

  In this paper, we study the literature of blockchain and its application and summarise each paper and its contribution. We also present an introduction about the essential concepts on the blockchain such as Bitcoin, Ethereum, smart contract, etc. This paper forms the basis of Chapter 2.

- **Performance Benchmarking for Smart Contracts to Assess Miner Incentives in Ethereum**, A Aldweesh, M Alharby, E Solaiman, A van Moorsel, in 2018 14th European Dependable Computing Conference (EDCC), 144-149 [6].

  In this paper, we demonstrate the performance impact of smart contracts and incentives that miners gain when executing the smart contracts. We We find that the cost of executing smart contracts is not always proportional to the fee miners receive. This paper forms the basis of Chapter 3.

- **Performance Benchmarking for Ethereum Opcodes**, A Aldweesh, M Alharby, A van Moorsel, in 2018 IEEE/ACS 15th International Conference on Computer Systems and Applications (AICCSA) [3].

  In this paper, we introduce our preliminary investigation into the performance of so-called opcodes and the relation between the CPU usage and the fee received. We explore that there are miss-priced opcodes that could affect the operation of the blockchain. This paper partly forms the basis of Chapter 4.

- **OpBench: A CPU Performance Benchmark for Ethereum Smart Contract Operation Code** A Aldweesh, Maher Alharby, Maryam Mehrnezhad Aad van Moorsel, in Proceedings of the 2nd IEEE International Conference on Blockchain 2019 [5].

  In this paper, we propose OpBench, a platform-independent benchmarking approach. OpBench measures the CPU time required to execute opcodes in the Ethereum Virtual Machine. We implemented OpBench for the PyEthApp, Parity and Go-Ethereum clients, and present results for both platforms on three different machines and operating systems. The results show that the static fees set by Ethereum are not always proportional to the invested CPU time, with up to an order of magnitude difference across opcodes. This paper contributes to parts of Chapters 4 and 5.

- **Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing**, C Dong, Y Wang, A Aldweesh, P McCorry, A van Moorsel, in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security [34].

  In this paper, we introduce the first smart contract-based solution for cost-effective verifiable cloud computing. We implement and execute the solution

on both private and public Ethereum networks. This paper forms the basis of Chapter 6.

- **Non-blocking Two-Phase Commit Using Blockchain**. Ezhilchelvan, P, Aldweesh, A, Moorsel, Concurrency Computat Pract Exper. 2019; e5276. `https://doi.org/10.1002/cpe.5276` [38].

  In this paper, we present the implementation, the performance analysis, and the possibility of 2PC using a blockchain-based smart contract. We demonstrate that the 2PC blocking can be eliminated under certain requirements. This paper forms the basis of Chapter 7.

- **Non-Blocking Two Phase Commit Using Blockchain**, P Ezhilchelvan, A Aldweesh, A van Moorsel, in Proceedings of the 1st Workshop on Cryptocurrencies and Blockchains for Distributed Systems [37].

  This paper investigates eliminating that 2PC blocking vulnerability by coordinating 2PC using a blockchain that supports execution of user-defined smart contracts. It demonstrates that the 2PC blocking can be eliminated at a moderate financial cost, if the blockchain also meets the synchrony requirements. Otherwise, despite the blockchain being a reliable state-machine, eliminating 2PC blocking may well be impossible, depending on whether the cluster hosting the database is synchronous or not. This paper forms the basis of Chapter 7.

- **Blockchain-based Smart Contracts: A Systematic Mapping Study of Academic Research (2018)**, M. Alharby, A. Aldweesh, and A. Van Moorsel, in Cloud Computing, Big Data and Blockchain (ICCBB 2018), International Conference on. 2018: IEEE, 2018 [9].

  In this research, our interest is twofold, namely to provide a survey of the scientific literature and to identify academic research trends and uptake. We only focus on peer-reviewed scientific publications to determine how academic researchers have taken up smart contract technologies and established scientific outputs. We classified the output papers into six categories, namely, security, privacy, software engineering, application, performance & scalability, and other smart contract related topics. We found that the majority of the papers fall into the applications (about 64%) and software engineering (21%) categories. This paper contributes in part to Chapter 2.

Chapter 2

Background: Blockchain and Benchmarking

2.1 Summary

This chapter outlines relevant background material motivating the work carried out this thesis. Section 2.2 provides an overview of blockchain definition, types and applications. In Section 2.3, we introduce the first application using the blockchain. This section gives an overview of Bitcoin, its transactions, its mining process, Proof-of-Work algorithm and its applications. Then we give details about the inner-working of Ethereum in Section 2.4. We discuss Ethereum definition, transaction and accounts before we summarize smart contracts and the incentive mechanisms. We also highlight the mining process and network node types. In Section 2.5, we discuss the benchmarking in the field of computer science, providing its types and classifications. Finally, Section 2.6 concludes the chapter.

2.2 Blockchain Overview

Blockchain is a distributed ledger where all transactions are recorded and shared by all nodes participating in the blockchain network. The structure of the blockchain is that a block that consists of transactions is connected with the previous block in a chain form. To add a new block to the chain a process called proof-of-work (POW) is completed, see Section 2.3.3. POW prevents attackers from forging the blockchain and avoid the double spending issue [25]. Figure 2.1 depicts the structure of a block in the blockchain. As depicted in the Figure, each block consists of a main body that has all the block's transactions and a header with meta-data. The header contains

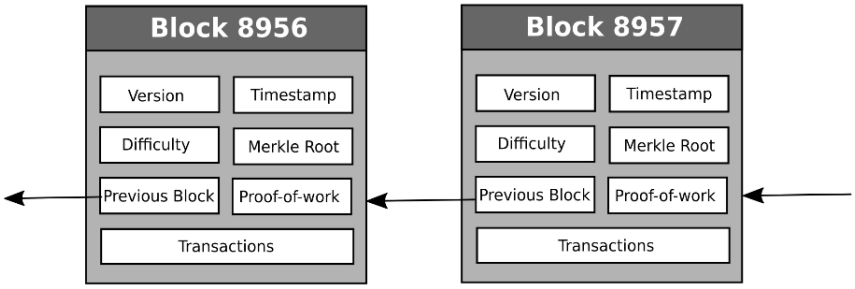

Figure 2.1: The structure of a block in the blockchain.

the protocol version, the timestamp of its creation, the difficulty, the Merle Tree root of the transactions, the hash of the previous block and the PoW (see Section 2.3.3).

Blockchains can be classified into two types, namely public blockchain and private blockchain [70]. The public blockchain is a blockchain that anyone in the world can read, send a transaction to and verify a block (e.g. Bitcoin blockchain). This type is useful when the participants of the network do not know (and thus, do not trust) each other. The private blockchain is a blockchain where writing permissions are restricted to a known set of users or an organization.

Both types of blockchain have their merits and demerits. The main difference is that private blockchain is more efficient since it does not have a decentralized consensus process such as PoW. However, the immutability feature could be broken by a few dishonest nodes. Private blockchains offer more privacy to users. This is because this type of blockchain is not visible for outside. A summary of a comparison between these two types is presented in Table 2.1. As shown in the Table, consensus determination in the public blockchain is through open membership, where any user can take part of the mining process, whereas only certain users are allowed to create and validate new blocks in the private blockchain. The data on the public blockchain is stored on each node so the availability is to every node. However, in the private blockchain the administrators decide who can read and/or write to the data. In the public blockchain, as a result of storing the data on each node propagating a new block takes considerable time. So, the efficiency of the Private blockchain is higher than the public blockchain and thus the cost.

A cryptocurrency is a digital or virtual currency based on cryptography. Using the blockchain as a database for users to exchange their virtual money. Signed data called transactions are handled by users to interact and transfer their digital assets. Nodes called (Miners) are responsible for gathering, and applying transactions in a peer-to-peer network.

Property	Public	Private
Consensus determination	All users	Only authorized miners
Read/write permission	Public	Private
Immutability	Almost impossible	Could be tampered
Efficiency	Low	High
Cost	Expensive	Cheap

Table 2.1: Comparison between public and private blockchains.

Thanks to the consensus algorithm of the blockchain system, the data on the blockchain is guaranteed (i.e., there is no double spending or invalid signatures). Data on the blockchain is public for users, and no one can be prevented from submitting a transaction that would be included in the blockchain.

Bitcoin [75] is the first cryptocurrency built atop of the blockchain infrastructure and protocol. It is the most popular and most valuable cryptocurrency. Nowadays, many alternate cryptocurrencies are cloned, forked of Bitcoin, while others are new that different than the existing one.

Decentralized Applications (Smart contracts). is a distributed software that contains any sets of agreements represented in any high-level languages like Solidity [48]. It also is defined as a computer program stored on the blockchain and is honestly executed by the peer-to-peer network. The most crucial element in the second generation of blockchain Blockchain 2.0 is the decentralized application (i.e., the smart contract). To solve problems that are common and to reach agreements within a minimal trust, the smart contract is deployed and executed on the blockchain and is used by connected components [20].

Smart Contract is a technology that allows end users to build a self-executing contract on the blockchain. The scripting language used in the Bitcoin network does not support complex control flow, and it has limited expressiveness. Hence, smart contracts in this network are straightforward. A blockchain-based platform called Ethereum was proposed to address the issue of supporting complex contracts. Ethereum started from scratch to build its blockchain and to introduce a scripting language to write complex smart contracts.

2.3 Bitcoin

Blockchain was first utilized and introduced in the Bitcoin network. The Bitcoin network is a peer-to-peer decentralized electronic cash system created by Satoshi

Nakatomo [75]. It validates transactions without the need for a trusted third party [27]. To validate transactions between the networks participants and to ensure the integrity of transactions, public key cryptography and digital signature have been utilized. The use of these cryptographic mechanisms provides high data security principles (e.g. Integrity, Confidentiality, and non-repudiation) [17]. The public key in Bitcoin is considered as participant addresses of where they can receive transactions, and the private key is used as ownership credentials. Private keys are stored in digital wallets for each participant, and their coins are represented as digital signatures.

In this section, we present an overview of Bitcoin inner details. The summary includes the Bitcoin address, which is the users pseudonymous identity, Bitcoin transaction, which allow users to exchange their assets over the blockchain. Then, we explain the consensus algorithm that power the Bitcoin network (i.e., PoW).

2.3.1 Bitcoin Address

Every node in the Bitcoin network has an account that contains their address and their balance. The address is public/private keys that are created by the user as a keys file and stored on the users local hard drive, not on the blockchain [14].

The keys are used to sign transactions that transfer Bitcoin from one account to another as well as to proof the ownership of a number of Bitcoin on a users balances. The private key is a 256-bit generated randomly. This private key is used to create a 512-bit public key using Elliptic Curve Digital Signature Algorithm. To generate a transaction, an account creates a signature using their private key, then broadcast their public key. A miner who maintains the ledger can verify the authority of this signature using the accounts public key. To receive an amount of bitcoin, the hash value of the public key which calculated as $(RIPEMD_{160}(SHA_{256}(K_{pub})))$ is used as account number by the sender of the bitcoin [14].

2.3.2 Bitcoin Transaction

A transaction is a data structure that is used to exchange bitcoins between users. It has one or more inputs and one or more outputs. The inputs refer to previous transactions outputs called UTXO (Unspent Transaction Output). UTXO is a bitcoins owned by a specific user and stored on the blockchain. The wallet software [14] is used to calculate the balance of a user by counting the number of UTXOs.

A Forth-like scripting language is used to control the input and the output that enforce the conditions needed to claim the bitcoins. There are two approaches used as

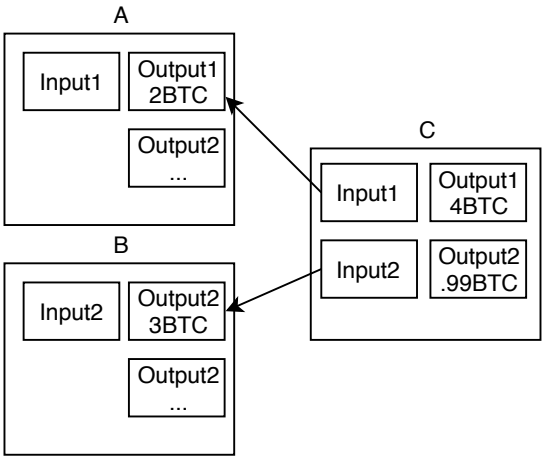

Figure 2.2: Bitcoin Transaction Overview.

scripting languages [2]. The *pay-to-pubkey-hash* script and *pay-to-script-hash* script. The former requires a single signature to authorize the payment. The later was introduced as a soft-fork in 2016 which enables a range of transactions types. The later is the most common used [2].

In the example depicted in Figure 2.2, if Alice wants to send four bitcoins to Bob, she needs to have a UTXO with four bitcoins or number of UTXOs that their sum is equal or more than four bitcoins. As can be seen in the Figure, Alice has two UTXO their sum is five bitcoins so that she can send four bitcoins to Bob. In this example, Alice creates a transaction that contains two inputs A and B, which outputs a single UTXO with two recipients, Bob and herself. The difference between the inputs and the outputs values are the transaction fee paid to the miners.

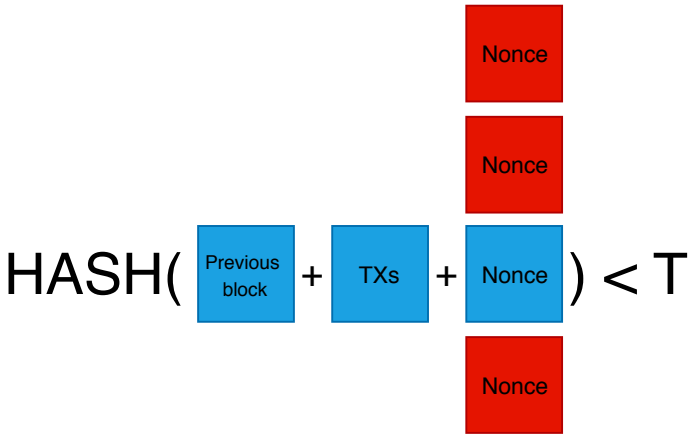

Figure 2.3: Schematic representation of PoW.

2.3.3 Proof-of-Work (PoW) Algorithm

The blockchain network synchronizes a set of a distributed databases where each node distrusts others without a trusted authority. This is achieved by using the PoW. PoW is a certain computational cryptographic hash puzzle, which requires a computational resource to be solved. First, miner collects all transactions created after the last block and validates these transactions. Transaction validation is to check the balance of the sender and the correctness of the signature. Then, miners repeatedly generate a random number (nonce) till the hash value of the nonce together with the block header and a hash of the Merkle tree root of the transactions that have been included in the block results in a hash value that is less than a target (T) value see Figure 2.3. The target value depends on the total hashing power of the network and is adjusted dynamically such that the block creation time on average is 10 minutes. The node that finds the nonce first includes the hash in to the block header and broadcasts the block to the network. The first node to solve this puzzle is rewarded both the transaction fees and the block reward.

It is important to realize that by design no miner can influence their chances to *win* the PoW competition, other than by adding computational resources. It can be shown that the probability of a miner to win the next block is equal to its share of the overall hashing power of all miners.

2.3.4 Bitcoin Mining

Mining in the Bitcoin network is the process of validating transactions as well as creating new blocks. Each node in the Bitcoin network can play the role of miner. The mining process involves validating a transaction by checking the correctness of its structure, which includes validating the signature of the sender and their ownership of the spent value and solving the PoW puzzle. As a result, miners create network-wide distributed consensus. In return for that, miners are rewarded a sum of transactions fee as well as a fixed amount of Bitcoin. All peers on the network verify the PoW solution before appending the created block to the blockchain and broadcasting it to other peers.

2.3.5 Applications of Bitcoin

Bitcoin was designed to provide a secure medium to exchange assets between parties. The success of Bitcoin inspired the community to explore applications for both

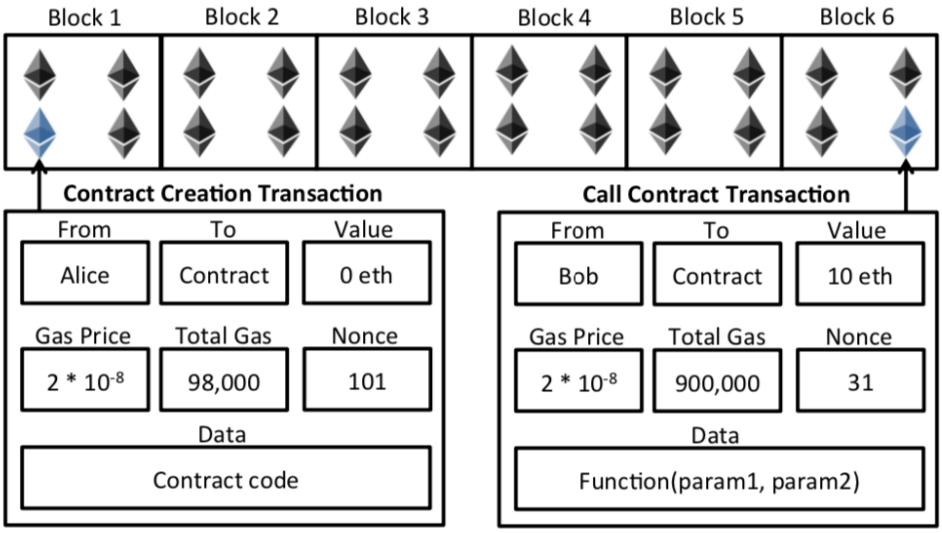

Figure 2.4: Ethereum's contract creation and call-contract transactions [72].

blockchain and Bitcoin such as exchanging assets [62], carbon dating [26] and authenticating devices in the Internet of Things [61]. These applications use the blockchain as a shared database to store data. The Bitcoin network itself cannot be used to validate transactions that contain application-specific data, and instead, it relies on an external third party. Therefore, Bitcoin is not the ideal platform for such applications. The following sections introduce Ethereum, which can be seen as a global computer that can be used to deploy applications with complex transactions.

2.4 Ethereum

The main idea of the Ethereum project is to introduce a global machine able to run and execute a distributed application build from Smart contracts over the blockchain. Ethereum blockchain is similar to Bitcoin, in that it operates as a distributed ledger, in which all transactions history is recorded and stored in every node in the network. Additionally, in Ethereum, every node also stores the most recent state of smart contracts.

Ethereum's cryptocurrency is referred to as Ether, while the operational execution amount is identified as Gas. That is, computation takes an amount of Gas, which cost an amount of Ether. Several programming languages can be used to develop smart contracts, the most prevalent being Solidity, but Serpent and LLL are also common. Whatever language is deployed, the source code of the smart contract will

be transformed into bytecode so that the Ethereum Virtual Machine (EVM) can interpret it.

Ethereum Accounts. There are two user types for Ethereum, these types also being denoted as accounts: externally owned accounts and contract accounts. A balance is maintained for each account; should it be a contract account, it will additionally have a code and storage facility associated with it. The blockchain stores the code, which is expressed in bytecode, to enable the EVM to interpret it. In order for a contract to be executed, a signed transaction with fields such as Receiver, Data and Gas Price is forwarded on to the blockchain. The collection of transactions is by miners (whose duty is to keep the blockchain operable), who then use the EVM to execute the code. Should there be a fruitful execution, the value of variables used in the smart contract will alter, as will the balance and storage, but if the execution is unsuccessful, no alteration will occur.

Ethereum Transaction. There are two forms of transactions in Ethereum: financial and contract-creation/call-contract transactions. The former is to transfer currency between accounts, while the latter is either to attach to the blockchain a fresh smart contract (contract-creation), or to engage a contract that already exists (call-contract transaction). Figure 2.4 highlights the Ethereum transactions that are structured as follow [94].

- **From**: A signature form an externally owned account to authorize the transaction.
- **To**: The receiver (contract or externally owned).
- **Data**: Contains either the contract's code or a function's identifier and its arguments.
- **Total Gas**: Is the maximum amount of gas that transaction's submitters commit to the transactions
- **Gas Price**: A value equal to the number of ether to be used to purchase the unit of gas.
- **Nonce**: An incremented value equal to the number of transactions sent by the sender.
- **Value**: A number of ether the sender is willing to send.

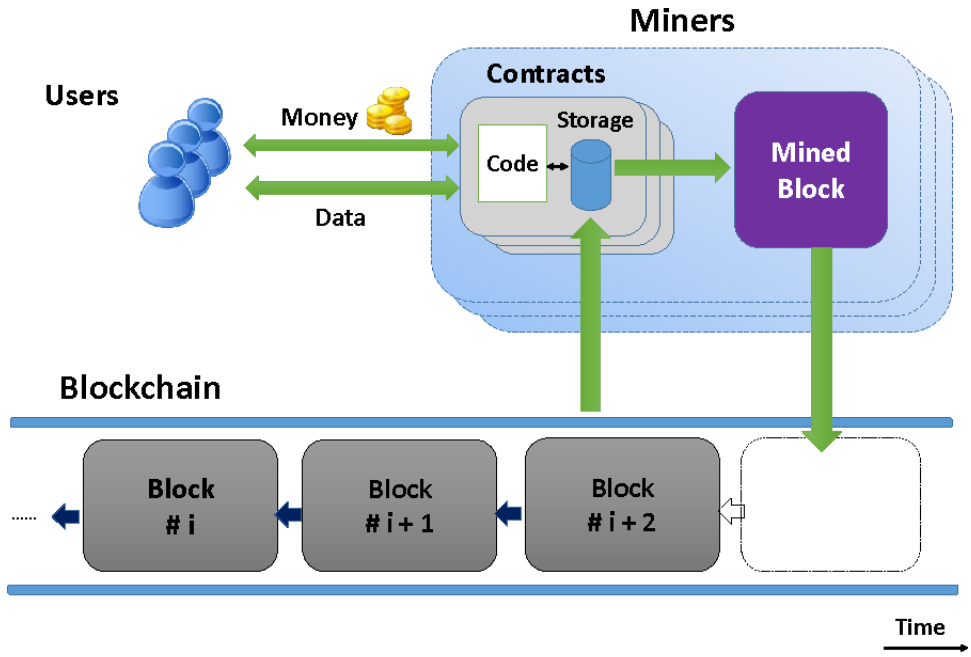

Figure 2.5: How Ethereums users interact with the smart contract on the blockchain [28].

For the contract-creation transaction, one needs to attach the compiled bytecode for the smart contract that needs to be deployed. Upon successful execution of the transaction, the contract will be assigned to a unique 160-bit identifier address. Later on, the contract can be invoked by submitting a call-contract transaction that specifies the address of the contract, the functions to be executed, and possibly all input data required by those functions.

2.4.1 Smart Contracts

A smart contract is a computer program that is retained and executed on a blockchain like Ethereum, with the accuracy of execution being guaranteed by the consensus protocol [20]. The contract can include any type of agreements that can be defined in whatever high-level language is relevant. A range of applications can be realized by smart contracts; these include financial applications, such as saving wallets or wills, or computer cloud functions, for example, [34]. The quantity of verified Ethereum smart contracts at the time of writing is around 50,000[1].

Smart contracts exist as bytecode on the blockchain, and are called by their 160-bit identifier address, then deployed on submission of a Contract-creation transaction. Upon acceptance and incorporation into the blockchain, the invoking of the contact

[1]https://etherscan.io/contractsVerified

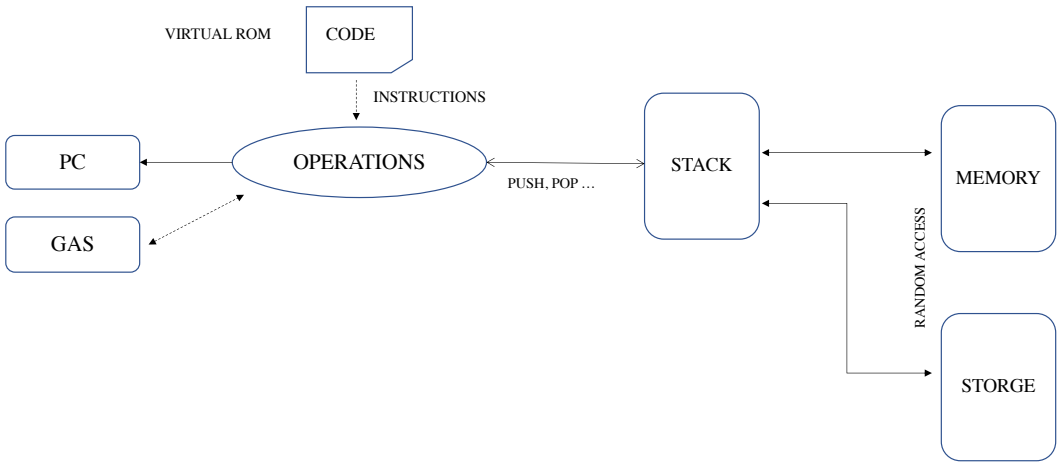

Figure 2.6: The Execution model of the Ethereum Virtual Machine's (EVM) parts and their interactions.

and its uses can be undertaken by any Ethereum account. More precisely, when a new transaction with a contract address as recipient is accepted by the blockchain, then the smart contract is executed by all miners, with the blockchain current state and the transaction payloads as input. The results of transaction executions are permanently recorded in the blockchain..

Figure 2.5 shows how users interact with smart contracts. The codes and states of smart contracts are stored on the blockchain. Miners execute the smart contract's code and reach a consensus of the execution outcome. Then, accordingly, the state of the smart contract is updated. Users can send and received money and data from the smart contract.

The contract called by a transaction that invokes a function in the contract. The transaction can have a value in ether, function parameters or both specified in the value field and the data field of the transaction respectively. The contract can receive as well as sent ether to a user or a contract. The variables of the contract are stored in a persisted place called storage. The intermediate result of contract execution is stored in a non persisted memory as byte-array. The contracts code and variables are public and stored in the blockchain.

2.4.2 Ethereum Virtual Machine (EVM)

The EVM is stack-based Turing Complete machine with a predefined set of opcodes (instructions) that handles the smart contract execution and their state. Hence, in general, the contract is basically a series of opcodes, executed on the EVM.

The architecture and the execution model of the EVM are presented in Figure 2.6. As can be seen, the smart contracts bytecode is stored in an immutable virtual ROM. This means once a smart contract is deployed to the blockchain, it cannot be updated. The EVM has five data values: a program counter (PC), a Gas counter (GAS), a stack, a memory and a storage. The PC is a value that points to which opcode is to be executed. Then the Gas cost of each executed opcode is stored in the Gas entry. The stack is where most opcodes consume their parameters. The operations over the stack are `PUSH, SWAP, DUP` and `POP`. The `PUSH` opcode adds one element to the stack, the `SWAP` opcode exchanges an index of a value in the stack for another. The `DUP` opcode duplicates an item in the stack, and `POP` removes one item from the stack. The memory space is used to store data during the execution. Finally, the storage is the space where the smart contract's variables are stored.

Every instruction the EVM runs has a cost related with it, measure with gas, to ensure a precise resource handling of the EVM. Based on the complexity of the computational resources, each opcode has a different cost [24]. Opcodes that require complex computation cost more than the one require fewer. The Ethereum foundation sets the gas cost of these instructions [94]. For example, the `ADD` instruction costs three units of gas while `MUL` instruction costs five units of gas, which more complex than the previous instruction [94]. The EVM keeps a record of the instructions being executed and cumulates their associated gas costs.

2.4.3 Ethereum Layers

As mentioned in Section 2.7, a smart contract is executed on the Ethereum EVM, which implemented in every Ethereum's machine connecting to the Ethereum network. Figure 2.7 shows where the EVM fits in the Ethereum blockchain system and where the EVM is embedded within each machine. As can be seen in Figure 2.7, the Ethereum machine consists of four layers. In the first layer, the smart contract code is encoded to bytecode (i.e., EVM code) using a proper compiler and deployed to the blockchain with a unique given address. The given address is then used to trigger the bytecode. Every time the given address is triggered, the bytecode is then executed on the EVM. The EVM is based on the second layer, and it runs the smart

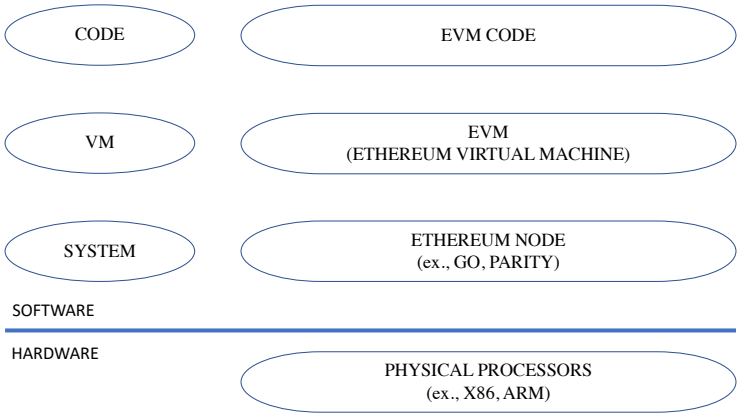

Figure 2.7: The basic architecture of the Ethereum system and where the Ethereum Virtual Machine (EVM) fits into the system.

contract bytecode each time the system receives a transaction. The EVM is essential for both the Ethereum system and protocol. It allows peers safely and in a trustless ecosystem to execute codes where the outcome of the execution is guaranteed. These layers are embedded in the Ethereum full node, such as Go [44] and Parity [45], which is depicted in the third layer of the system. The final layer is the physical processor of the machine that runs the Ethereum node.

2.4.4 Ethereum Execution Model

The execution of a smart contract is done in a serialized manner, one by one, where every opcode is allocated a byte (eg., `ADD` is `0x01`). In the beginning, the PC points to the first byte to be executed, and then its value is incremented by one to point to the next byte. Each opcode retrieves its parameter either from the stack or from memory. In some cases, opcodes write the execution results onto the stack, which can be read by other opcodes. The EVM reads from and writes onto the storage by taking the parameters from the stack and performing an update to the storage using the opcode `SSTORE`.

The following example depicted in Figure 2.8 simulates the execution of a simple, smart contact complied using the Solidity compiler to this hexadecimal bytecode `0x6001600101`. The contract adds two numbers and stores the result in the persistent storage. During the execution the compiled bytecode is divided into two hexadecimal values except for `PUSHs` opcodes that treated differently. `PUSHs` opcodes are divided

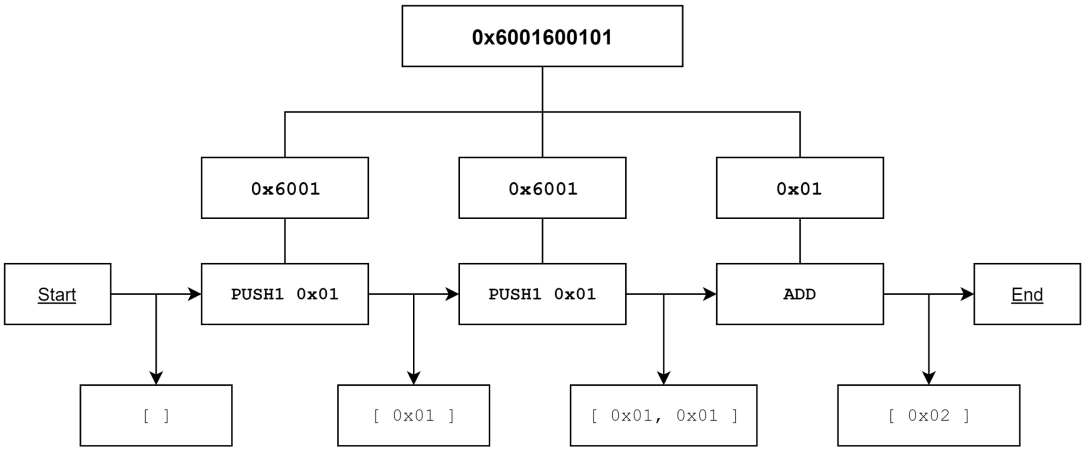

Figure 2.8: Simulating the addition operation of two numbers on the EVM.

into four hexadecimal values because these opcodes push values into the stack. In our example the first opcode is `0x60` that translated to `PUSH` see [94] for all hexadecimal representation of all EVM opcodes. Hence, the following 1 byte (`0x01`) is added to the stack "[0x01]". The PC counter is incremented by one to points to the next opcode, which is `0x60`. Similar to the previous opcode, the stack is filled with the second value `0x01` "[0x01, 0x01]". The stack now contains two values `0x01` and `0x01`. The final opcode is `0x01` which according to [94], means `ADD`. It retrieves two items from the stack and pushes the sum of these items to the stack. The final state of the stack is `0x02`. Finally, the execution is halt and contracts state is updated.

2.4.5 Ethereum Clients

The Ethereum client is a software application that implements both specification and communication of Ethereum protocol through the peer-to-peer network. The Ethereum project has multiple client implementations across a different range of operating systems (OSs). The reason for this varieties of implementations is to verify the correctness of the project protocol by testing it on a range of OSs as well as to find consensus problems [63]. As of the time of writing, eight clients implement the Ethereum protocol as presented in Table 2.2. As of June 2019, according to [52], the leading implementations are Parity and Go-Ethereum. An Ethereum client interacts with the blockchain by verifying and creating transactions as well as mining blocks. Note, the Ethereum client is the software application that implements the Ethereum specifications as mentioned above, whereas the Ethereum node is the machine that in-

stalls and runs the Ethereum client (see next the Subsection). Thus, many Ethereum clients can be installed on a single node.

Client	Programming language
PyEthApp	Python
Go-Etereum	Golang
Parity	Rust
Ethereumj	Java
Ethereumjs	Javascript
Ciri	Ruby
Aleth	C++
ethereumH	Haskell

Table 2.2: EVM client implementations.

2.4.6 Ethereum Peer-to-Peer Network

The Ethereum network is similar to the Bitcoin network in that anyone can join and leave at any time, this is known as open-membership. Ethereum has multiple different implementations that validate and run the network as Full Ethereum nodes or Light Ethereum Subprotocol (LES) nodes. the Ethereum node is the machine that installs and runs the Ethereum client.

Full Ethereum nodes download, validate and store a full copy of the blockchain. This includes downloading and validating each block contents, verifying its PoW and downloading the global state of the Ethereum, which includes all accounts/contracts balances, storage, and codes. In addition, the global states hash is verified against a specific hash header within a particular block as seen in Figure 2.9. Note, all newly created blocks are stored and validated once the node is fully synchronized to the network.

Light Ethereum Subprotocol nodes (LES) involve downloading and validating only block header. Transactions and receipts are requested from the network later. Checking and verifying transactions and their receipts in a block is done by checking the block headers cryptographic commitments. According to [39] a node requires storage of around 10MB, for bandwidth while idling about 1MB/h, and for stage/storages request from the network roughly 2-3kB. Although this node in its infancy, it could be improved to be used by mobile devices in the future.

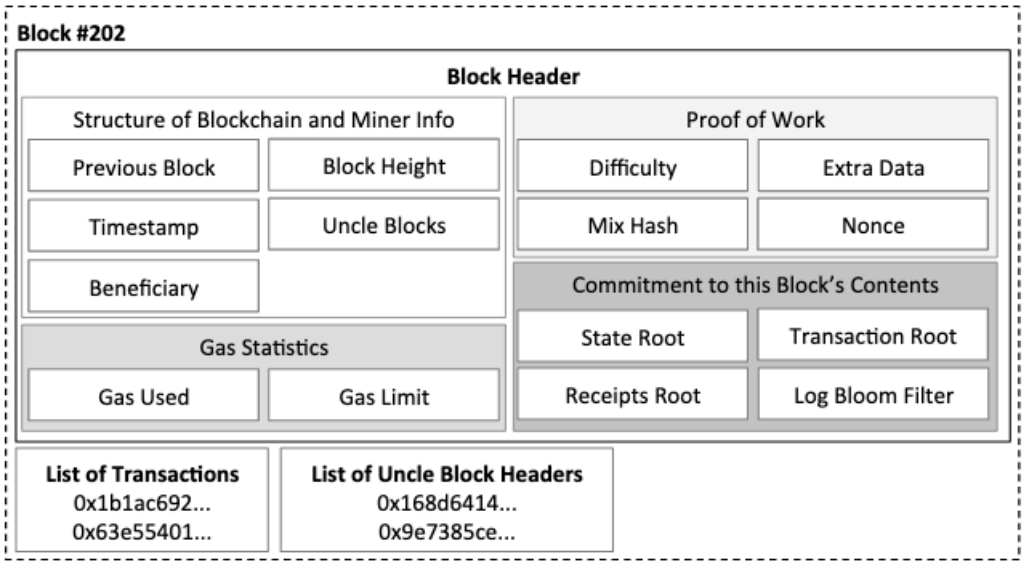

Figure 2.9: Ethereum's block header and transactions [72].

2.4.7 Ethereum Mining

Similar to Bitcoin mining, Ethereum uses an improved PoW mechanism to validate a created block called Ethash [93]. A block is only valid if it has a PoW of a specific difficulty. The Ethereum mining difficulty is automatically adjusted; therefore, a new block is mined every twelve seconds.

Ethash [93] is developed to be memory hard. Therefore, it is impossible to implement on ASIC(Application-Specific Integrated Circuit) approaches. In order to calculate a PoW, a large amount of data is required to be stored in RAM, organized as a Directed Acyclic Graph (DAG) and are collected from the block headers and the nonces. The size of this data is 1GB, and it is changed every 30000 blocks. The verification of this data requires a low CPU power and memory[50].

Miners are rewarded for the efforts they invest in the network. The rewards are 5 ether for each block included to the main branch of the blockchain. They also collect transactions fee specified by the creator of the transaction. In the future, the transaction fee will surpass the fixed reward and will be the only incentive reward for the mining [50].

Unlike Bitcoin, Ethereum rewards miners who create a stale block. A stale block is a block that is an ancestor of the created block, but is not on the main chain. The reward for a stale block is 7/8 of the fix block rewards.

Nowadays, Mining difficulty of the Ethereum network is high; hence, only GPUs can perform the mining process [49].

2.4.8 Ethereum Gas Mechanism

We observed both transaction types earlier in Section 2.4: Financial and Contract-Creation or call-contract. Financial transactions, deployed in the cross-account transference of Esther, involve a cost of 21,000 gas units. However, Contract-creation or call-contract transaction costs are subject to whatever level of used gas the EVM determines, as well as the 21,000 gas units transaction cost.

The *Operations that the bytecode conducts* and *data inherent in the blockchain* are accounted for by the EVM. The former relates to the opcodes that the transaction executes, in which the costs of all opcodes are pre-set, while the latter levies a charge of 20,000 gas units for zero to non-zero storage levels; otherwise the fee is 5,000 gas units.

Transactions in Ethereum also include *Gas Price* and *Total Gas* [94]. Executing transactions consumes gas that can be converted into Ether using the *Gas Price*, and the ether is charged to the transaction submitters. Moreover, the higher the *Gas Price*, the more chance to execute the transaction faster. *Total Gas* is the maximum amount of gas that transaction submitters commit to the transactions. If the *Total Gas* is less than the execution needs, the transaction will fail with an exception out-of-gas, and the full amount of gas will be paid to the miners. Note that all unused gas is returned to the transaction submitters. All submitted transactions are located in the memory pool of each node, miners collect these transactions and execute them to include them in their block. Miners tend to select transactions that look more profitable to them based on the *Gas Price*. In addition to the *Gas Price*, miners prefer transactions that involve deploying new contracts to gain more gas.

There are several reasons why the fee framework is considered as necessary, as was noted earlier; in the interests of completeness, these reasons are discussed here. If the submitter places a limit on the quantity of gas transacted, the execution is sure to terminate, as when any opcode executes, a positive quantity of gas is added to the total of used gas. This resolves the difficulty of Turing complete programmes halting, as the termination is certain. Another positive consequence of the fee framework is the avoidance of attacks from denial of service [54], which would ensue if miner resources were executed on worthless contracts at zero cost, thereby preventing the miners carrying out tasks necessary for the blockchain to operate reliably. The attacks sustained by Ethereum in 2016, for example, the `EXTCODESIZE` attack [19] were able to happen because particular opcodes needed a lot of computation but would only involve a small fee for the submitter. The attack resulted in Ethereum altering the `EXTCODESIZE` fee to reduce the effects of an attack.

Thus, the introduction of the gas in the Ethereum network is to encourage developers to avoid writing a wasteful code, as well as to ensure that miners are compensated for their resources contribution.

2.4.9 Solidity

Solidity [48] is the official high-level programming language that is utilized to develop a smart contract. It is maintained by the Ethereum foundation, which suggests using it as the main contract language. It is a JavaScript-like language; thus, it is an Objective Oriented language. It supports different data types like any traditional language, such as integer, and boolean.

Moreover, Solidity also supports structs, enumerations and byte-array data structures. Mapping, which can be seen as a key-value or a hash-table [65] data structure, is also supported by Solidity. In order to access the transactions log, which is a special data structure in the blockchain that stores the outputs of a contract function, Solidity provides Events. Users who initiate transactions can see events. Thus, the contract outputs can be stored into events to maintain the sequences of functions execution. Although there are other programming languages, Solidity is the most commonly used language for developing smart contracts. Using Solidity, developers can write a smart contract that enables, e.g., voting, blind auction, and more. In chapters 6 and 7, we show how we used Solidity to develop smart contracts in different domains like cloud computing and distributed database management systems.

2.5 Benchmarks

This section provides a general introduction to the field of performance benchmarks, which we use in Chapter 3 and 4. It introduces common definitions of the term Benchmark in 2.5.1. It presents a common requirements for significant benchmarks in 2.5.2. Followed by the most common types and classifications of the benchmark as we shall see in 2.5.3. Finally, in 2.5.4, it defines the difference between the workload and the metrics.

2.5.1 Definition for Benchmarks

Benchmark is the evaluation process used in experiments to compare different tools, platforms and/or techniques [16]. Benchmark in computer science is defined as a tool to compare, for example, the performance CPUs, systems or algorithms [89].

In addition to performance evaluation, benchmarks can also measure things such as the number of false negatives or positives in an algorithm. The term *benchmark* is defined by several organizations and standards which provide different definitions. The following are common definitions from the literature.

The ISO/IEC25010 [40] standard for software quality provides a very general definition of the term benchmark.

> **Definition: Benchmark**
>
> *A standard against which results can be measured or assessed.*

IEEE systems and software engineering vocabulary [64] provided similar definitions to the previous definition by ISO/IEC25010:

> **Definition: Benchmark**
>
> 1. *A standard against which measurements or comparisons can be made.*
> 2. *A procedure, problem, or test that can be used to compare systems or components to each other or a standard.*

The glossary of the Standard Performance Evaluation Corporation (SPEC)[55] provides a definition that is more focused on the performance of computer systems:

> **Definition: Benchmark**
>
> *A benchmark is a test, or set of tests, designed to compare the performance of one computer system against the performance of others.*

In this PhD research, we define the benchmark as *a performance measurement of a smart contract code against a computer's CPU at both smart contract and operations codes (opcode).* We compare the CPU performance of one Ethereum's client against the others at smart contracts level. We also compare the CPU performance of a computer system (i.e., smart contracts at opcodes level) against others.

2.5.2 Benchmarks Requirements

According to [36], the benchmarking objective is twofold: Firstly, to improve and enhance the system design and identify performance bottleneck. Secondly, to propose a standard for systems comparison, e.g., TPC-B [77].

In [74], authors presents benchmarks' properties and requirements that a significant benchmark must follow:

- Repeatable: benchmarks can be used by others to check the results and apply them to different systems.
- Comparable: benchmarks can be used to compare different results for a certain task.

- Relevant: benchmarks can be used to predict the behavior of real-life applications.

Based on these requirements, the benchmark's metrics should have the following features: Firstly, using a proper scientific process is necessary to obtain a realistic result. Secondly, an agreement between researchers on the set of metrics used on the benchmark. Finally, practical and widely used metrics must be used. Thus, the used metrics are applicable to both research and real-life domains.

2.5.3 Classifications of Benchmarks

According to [82], *micro-benchmark* and *macro-benchmark* are the most common categories that benchmark systems can be classified into.

Micro-benchmark is the performance evaluation of various small and specific parts of the software system. Most *micro-benchmarks* are written to test a particular type of operation in a large system such as performing a single type of CPU instructions. *Micro-benchmark* should be quickly repeatable across different systems, because the variation from other components of the system is factored out as much as possible.

Macro-benchmark, on the other hand, is designed to evaluate the performance of large and even complex system as well as to simulate a real system. It also can be as a representation of a real system. In abstract, *macro-benchmarks* is the evaluating of the overall functions of a large system, such as an accounting system, by simulating its task before releasing the final version. The simulating system involves all the functionality that the real system has.

In this thesis, for our experiments that shall be explained in details in the following two chapters, we used these two categories of benchmarks. The *micro-benchmarks* is used in chapter 4 to evaluate and to compare the CPU performance against the gas consumed for each instruction on the EVM (the opcodes). Also, the *macro-benchmarks* is utilized in chapter 3 to evaluate the overall performance of a smart contract and compare the CPU overhead with the gas collected.

2.5.4 Workload and Metrics

Benchmarks are classified into two types based on workload or metrics. For workload, it is classified into either homogeneous or heterogeneous based on the workload they run on the under-test systems. On the other hand, benchmarks classified based on

metrics types that the systems are characterized. For workload, the following are the most commonly used [36]:

- Latency: the interval between the request and response.
- Throughput: the amount of data processed per unit of time.
- Utilization: utilizing the computing time on the CPU in percentage.

Certain aspects of performance are represented by metrics. For a given combination of user requirements and workload, some aspects might be more important than the others. For instance, to characterize the performance of an interactive system, throughput might not be the appropriate metric.

2.6 Conclusion

There is considerable interest from both industry and academia for the success of the technology of blockchain. Both Bitcoin [75] and Ethereum [94] are blockchain technology-based applications. Ethereum is adopting the smart contract technology using its Turing-complete execution machine, the EVM, that executes distributed smart contracts.

This chapter provided essential background information for blockchain in general, and some applications build on it, such as Bitcoin and Ethereum. We summarize Bitcoin, PoW, Bitcoin mining, Bitcoin transactions and finally, applications that based on the Bitcoin. Ethereum, which is the most popular platform form smart contract system, is explained in details as well as the programming language for smart contracts, Solidity. Finally, this chapter introduced a brief background of the term benchmark as well as its classifications. The next chapters are focused on both blockchain applications and benchmark techniques.

Chapter 3

Performance Benchmark of Blockchain Smart Contracts

3.1 Summary

A defining feature of the Ethereum blockchain is its ability to execute smart contracts, providing a Turing complete programming model for distributed applications in non-trusted environments. The successful operation of the Ethereum blockchain depends on whether the miners' incentives (in the form of fees) to execute contracts is proportional to the miners' cost (in terms of energy usage, and thus CPU usage). In general, if the received fee is not proportional to the computational cost, miners would prefer some tasks over others, therefore potentially adversely affecting the continuing dependable operation of the blockchain. In this chapter, we design a benchmark to compare smart contract execution time with the award a miner would receive, to determine if incentives align.

We present the design of the benchmarking approach and provide results for the Python Ethereum client running on a Mac. The results indicate that for functions in Ethereum's most popular contracts, the difference of reward per CPU microsecond can be up to a factor of almost 50. Besides, contract creation, which is done once for each new contract, can be up to 6 times more lucrative than the regular execution of contract functions. Potentially, these discrepancies result in misaligned incentives that impact the dependable operation of the blockchain.

The structure of the chapter is as follows. Section 3.3 describes in detail the process of collecting the data needed for the experiment. The experiment design and procedure are presented in Section 3.4. Section 3.5 shows our benchmarking approach results and observations. Finally, Section 2.6 concludes the chapter.

3.2 Introduction

Permissionless blockchains, such as Bitcoin [75] and Ethereum [94], rely on miners for their successful operation. Miners invest computational resources and the energy to run them, and in return, receive a fee, expressed in the unit of cryptocurrency belonging with the blockchain. By establishing the right incentives, the miners are sufficiently motivated to keep operating the blockchain in a dependable manner. Recent literature investigates the alignment of incentives, for instance in relation to the long-term development of block and transaction rewards [22], in the context of miner pool strategies [96] and associated with denial of service attacks [54].

In this chapter, we consider incentives for miners when executing smart contracts in Ethereum. Ethereum contains a cryptocurrency, called Ether, but the defining feature of Ethereum is the ability to execute smart contracts, providing a Turing complete programming environment for distributed applications. Smart contracts are executed by the Ethereum Virtual Machines (EVM) of miners, who earn Ether depending on the executed machine language operations (called opcodes). In particular, when executing a contract, the EVM keeps track of the amount of 'used gas', based on the amounts of gas specified per opcode by the Ethereum foundation [43]. To determine the fee, the used gas is multiplied by the 'gas price' offered by the submitter of a transaction.

The fee structure for contract execution is essential for several reasons, including assuring termination of execution and avoiding denial of service attacks. We discussed this in more detail in Subsection 2.4.8. In this chapter, we are particularly interested in a less commonly researched reason why the fee structure is essential, namely that the fee structure determines the incentives required to assure the dependable long-term operation of the blockchain. Specifically, if there is no clear relationship between the computational effort needed and the fee awarded, miners cannot rely on a reasonable award for their energy investment. [8] demonstrates that under such uncertainty, miners could not optimize their profits. Moreover, if specific smart contracts are known not to be attractive, transactions using that smart contract would not be executed by miners. Therefore, to assure the dependable operation of blockchains with smart contracts, miners and submitters of transactions need to be confident that the fee structure correctly incentivizes miners.

This chapter aims to describe a benchmark for Ethereum smart contracts that evaluates whether the fee awarded for the execution of smart contracts is proportional to the computational effort required. We envisage that such a benchmark could be

run periodically, on a variety of software and hardware platforms, to demonstrate to the community if and how well costs and benefits are aligned within Ethereum. Users, as well as miners, would adjust their confidence in the dependable operation of Ethereum based on such a benchmark. We note that the performance of smart contracts execution becomes even more critical once Proof of Work will be abandoned by Ethereum [60]. Currently, the hashing for Proof of Work dominates the computational effort of miners, but in the future effort to compute smart contracts will be increasingly important, mainly if applications rely on increasingly complex contracts such as in [37, 34].

In the design of the smart contract benchmark, there are a number of issues we consider. The granularity of measurement can be of two types, either based on the overall contract or based on individual functions, and we propose that a benchmark should do both. (Note that this is different from benchmarking opcode execution time, as discussed in the next Chapter, which is meant to assess the correctness of the gas per opcode in [43].) Within current implementations of EVM, contracts are also executed once before they are added to the blockchain. Since the fee, as well as the functions executed, differs the benchmark should measure both such contract creation and normal contract execution. In terms of the contracts to be considered in a benchmark, we propose to use the most commonly used ones on current-day Ethereum, and we describe a procedure to obtain these. We use CPU usage as our main measure, as a first approximation of energy consumption and thus cost to miners.

Our experiments in this chapter concern a single hardware platform and a single EVM, the Python Ethereum virtual machine (PyEthApp)[46]. Although preliminary in terms of coverage of software and hardware platforms, the results are quite striking. We conclude that used gas is not always well aligned with the computational effort for either creating or executing smart contracts. There is a factor of almost 50 difference in gas reward per CPU second between varying contract functions as we shall see in Subsection 3.5.2. Also, the amount of used gas per CPU consumption for creating contracts is significantly more than that for contract execution (on average six times more) see Section 3.5.1.

3.3 Smart Contracts Selection

As discussed in [90], among other properties, a well-designed benchmark needs to be representative of the system at hand. Our approach is to select the most commonly

used smart contracts available on Ethereum by reducing the set of all existing smart contracts based on exclusion criteria.

To start, we use EtherScan[1] to identify all the available verified Ethereum smart contracts. EtherScan provides for all contracts in Ethereum the following: contract address, contract name, compiler, balance, transaction count, settings and verified date. At the time of collecting this data (10 May 2018), there were over 28000 verified contracts. Since EtherScan contains a large number of verified contracts distributed into many HTML pages, we had to write JavaScript code that fetches the contents from all those HTML pages. We were then able to produce a single HTML page that contains a single table with all verified contracts.

To establish a reasonable set of contracts for our study, we decided to select the first 100 most commonly used smart contracts. These most commonly used contracts were ranked by the number of transactions each contract experiences (transaction count). For instance, EtherDelta was the most widely used contract in Ethereum as it was invoked by about 9.6 million transactions.

For each contract, we manually collected the bytecode for the contract creation as well as analyzed the source code. The analysis process is vital in order to know what functions each contract is using and what possible inputs (if any) each function required. We also went through the transactions that were sent to invoke each contract in order to collect a transaction example for each function. This is to allow us to test and benchmark real Ethereum transactions instead of creating our transactions. Finally, we excluded all contracts that have not either source code or bytecode available.

Thus, we ended up with 76 different contracts. 24 contracts were excluded for the aforementioned reason. For the functions in each contract, we excluded some functions that cannot be obviously benchmarked without further assumptions about input parameters, such as transfer, transferFrom, withdraw, depositToken and some other token related functions. These functions require deployment assumptions and parameters that cannot be changed or updated within the available byte codes, so we excluded them.

For each contract, we look at the source code and received transactions to collect the most commonly used functions. We found that most of contracts have a single function, and some have two. Thus, we ended up with 77 different functions to be considered in our benchmark.

[1]https://etherscan.io/contractsVerified

3.4 Design of Benchmark Measurement System

It is possible to measure the computational efforts of both creation and execution of transactions in the Ethereum test-net or a private network, but it is not straightforward to control all aspects that may impact measuring CPU usage. In this section, we describe our system design to measure CPU usage for smart contracts.

Specifically, smart contracts are 'created' by deploying them on the blockchain through submitting a *Contract Creation* Transaction that contains the correct bytecode (Creation Code) for a smart contract. After the creation of the contract, this contract has an address at which it can be called. Once deployed, contracts can be executed by submitting transactions that refer to the contract's address. However, there is potential interference of a number of factors, including transaction validation overhead, signature validation overhead and proof or work computation. Therefore, we propose an alternative approach that isolates the execution of the smart contract from other computation. We measure the execution time in a single client EVM and simulate any of the other aspects as needed, for instance, the submission of transactions.

For the smart contracts selected as explained in Section 3.3, we investigate in detail the code of each collected smart contract at a function level, to identify how many functions each contract has, what inputs these need, and how to call the function. We initialized a set of Ethereum accounts with available balances to launch the transactions as well as to prepare the inputs for each function. For all functions, we use real transaction inputs submitted to the Ethereum, as mentioned in Section 3.3.

The next phase is to execute transactions. To that end, we create transactions, submit them using the accounts we set up and then execute each transaction in the local machine. For instance, to measure CPU use for a function within some contract, we first launch a transaction to deploy the smart contract containing this function and then start another transaction with all associated inputs to trigger that function. We record the timer for each of the described steps and store the results in a separate file. To compare different contracts, we calculate used gas per second of CPU use, so that one knows how much reward one can gain per unit of CPU time invested. For some contracts and functions, we were unable to execute transactions successfully. For instance, if a contract calls other unverified contracts, or because of source code issues. Eventually, we managed to execute 76 contracts (out of 100 contracts) and 21 functions (out of 77 functions) in total. All results and observations are presented in Section 3.5.

3.5 Results and Discussion

This section presents and discusses the first set of results for benchmarking both contract creation and function execution transactions. We conduct the experiment on a single machine that submits and mines all experiment transactions. The machine is a MacBook Pro and equipped with a 2.8 GHz Intel i5 CPU and 8 GB RAM. All transactions are executed 100 times, and the average time is calculated as well as the confidence interval. We use the Python PyEthApp[46] client. For each contract creation and function execution we read the data from the first phase and run the experiment 100 times, which provides 95% confidence intervals width with a half-width of less than 0.2 times the average.

3.5.1 Contract Creation Transactions

First, we measure the CPU usage for contract creation, for all 76 contracts. Figure 3.1 shows the amount of used gas that can be collected by miners per microsecond of CPU usage for contract creation transactions. The main curve presents the amount of used gas for creating each contract, while the straight line presents the average used gas for all the tested contracts. There is quite some difference between the return on investment for miners when creating contracts, roughly up to a factor of 5. The amount of used gas for creating contracts varies between 51 and 269 units of gas with an average of 182. Contracts such as Nagacoin, StatusContribution, MatchingMarket and Dragon are the most profitable ones with over 260 units of used gas awarded per microsecond of CPU usage. However, contracts such as TronToken, Controller, EKT, CybereitsToken and GnosisToken are more costly, with an amount of used gas awarded less than 100 units of gas. From these early experiments, we see that the profit (amount of used gas) miners can get from contract creation transactions is not particularly well aligned with their computational effort.

Table 3.1 shows the CPU usage (execution time in microseconds) and the amount of used gas for the most profitable and expensive contract creation transactions. The profitability is based on the amount of used gas offered to miners per time unit of CPU execution. It is clear that the CPU usage is not proportional to the amount of used gas offered by contract creation transactions. For example, both EKT and NAGACoin contracts consume roughly the same amount of CPU time, while the latter offers over three times as much used gas, and thus, offers more profit to the miners. As already mentioned, the profitability to miners can differ more than a factor of 5, for the contracts MatchingMarket and GnosisToken, respectively.

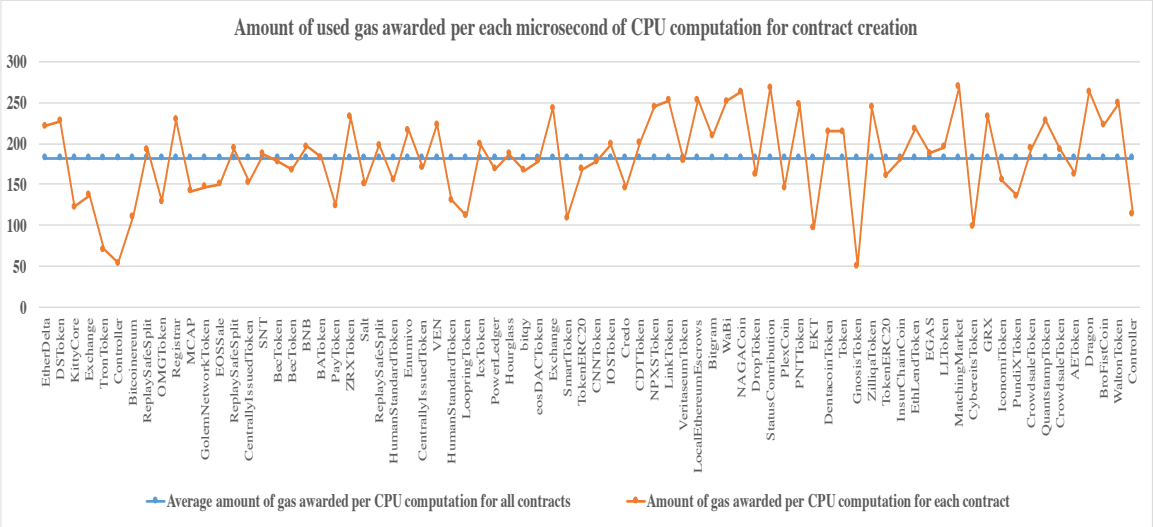

Figure 3.1: Amount of used gas awarded per each microsecond of CPU usage for contract creation.

Contract Name	CPU Usage (μs)	Used Gas	Gas/CPU
MatchingMarket	13193	3544767	269
StatusContribution	7918	2120935	268
NAGACoin	8491	2231105	263
Dragon	6226	1634521	263
EKT	7316	712297	97
TronToken	12476	891859	71
Controller	20160	1094303	54
GnosisToken	20419	1036626	51

Table 3.1: The most profitable and expensive contract creation transactions.

3.5.2 Function Execution Transactions

When executing a smart contract, often only a single function is called, and therefore we compare the execution phase based on functions, as opposed to full contracts. Figure 3.2 shows the amount of used gas collected by miners per microsecond of CPU usage for function execution transactions. For each function, we provide the contract they belong with, labeled C1 to C17. The main line presents the amount of used gas for executing each function, while the straight line presents the average used gas for all the tested functions. Some contracts have multiple functions that we call (e.g., contract C10 has two functions, namely, makeWallet and logSweep). In addition, some contracts share the same function name (e.g., both C1 and C2 have a function called deposit). However, the source code for those functions that have the same name is not necessarily identical, and thus, the benchmarking results may differ accordingly

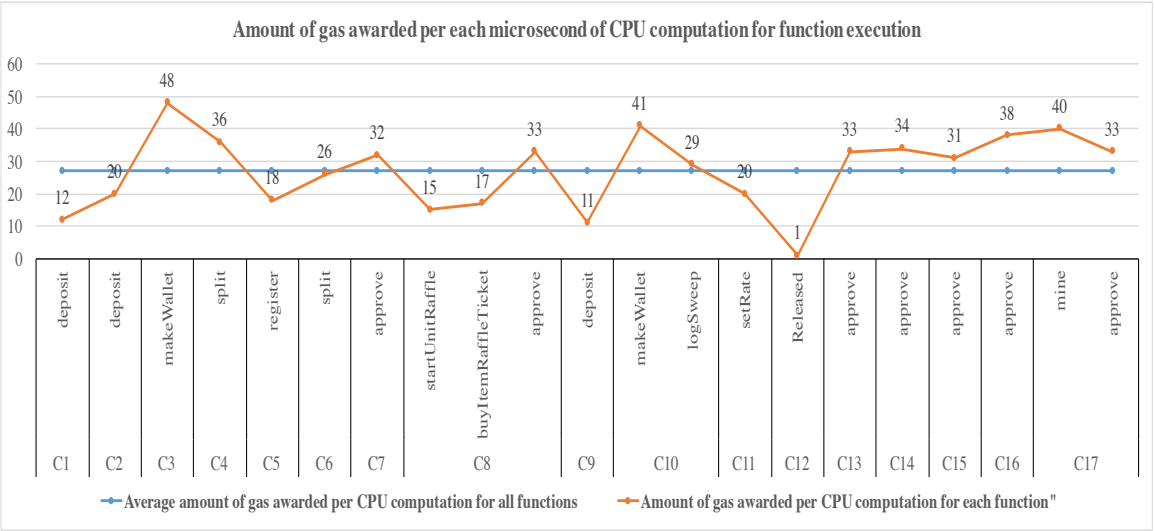

Figure 3.2: Amount of used gas awarded per each microsecond of CPU usage for function execution.

(see later in this section for a more detailed comparison for functions with identical names).

The amount of used gas collected from executing functions varies between 1 and 48 units of gas per CPU microsecond with an average of 27. This presents a considerable difference in terms of profitability to the miners. Functions such as makeWallet (in both C3 and C10) and mine (in C17) are the most profitable functions with 40 units of gas or more awarded for each microsecond of CPU usage. However, functions such as deposit (in C1), deposit (in C9), released (in C12) are more costly since the amount of used gas awarded is less than 15 units of gas. From these early experiments, we see that the profit (amount of used gas) miners can collect from function execution transactions is not well aligned with their computational effort. Miners could gain a factor up to 50 times more profit if they selected profitable function execution transactions such as makeWallet compared to the costly ones such as released.

Table 3.2 shows the CPU usage (the average execution time in microseconds), the amount of used gas and the used gas collected per CPU usage for all function execution transactions for more details. It is clear that the CPU usage is not proportional to the amount of used gas offered by function execution transactions to the miner. For instance, released function (in C12) is the most expensive function as it consumes much more CPU time compared to the used gas offered. Functions such as makeWallet (in both C3 and C10) are more profitable than the released function since they consume by far less CPU time while they offer more used gas.

Contract	Function Name	CPU Usage	Used Gas	Gas/CPU
C1	deposit	2365	29223	12.3
C2	deposit	2472	49265	19.9
C3	makeWallet	3861	184563	47.8
C4	split	2097	75753	36.12
C5	register	5072	89734	17.6
C6	split	1895	49664	26.2
C7	approve	1420	45677	32.16
C8	startUnitRaffle	1492	22899	16.12
	buyItemRaffleTicket	1297	22643	17.4
	approve	1387	45423	32.7
C9	deposit	2594	29411	11.3
C10	makeWallet	3484	142501	40.9
	logSweep	925	26509	28.6
C11	setRate	1406	28109	19.9
C12	Released	97669	116182	1.18
C13	approve	1451	47174	32.5
C14	approve	1345	45167	33.5
C15	approve	1487	45384	30.5
C16	approve	1190	45677	38.3
C17	mine	2131	84605	39.7
	approve	1378	45381	32.9

Table 3.2: The average execution time (in microsecond) and the amount of used gas for all function execution transactions.

We looked in more detail at the benchmarking results for functions with identical names (such as deposit, approve, split and makeWallet) used by different contracts. For the deposit function, which is used by three contracts, namely, C1, C2 and C9, we found the benchmarking results for this function in C2 is significantly different from that for other functions. Deposit function in C2 offers about 70% more used gas per each microsecond of CPU usage compared to deposit function in both C1 and C9.

We inspected the source code for the deposit functions and found that the deposit function in C2 has slightly different code with an extra line of computation, which takes the current block number and stores it inside a specific mapping to keep tracking of the last active transaction. This extra line of computation results in a significant increase in the amount of used gas, while the computational efforts were increased by less than 5%. Therefore, miners could gain more profit be selecting deposit function in C2 instead of other deposit functions.

Similarly, for the split function, which is used by both C4 and C6. Split function in C4 offers 38% more used gas compared to split function in C6. We inspected the source code for both functions and found that the latter has two extra IF checks. These checks resulted in a significant increase in the amount of used gas, while the computational efforts were only increased by about 10%. Therefore, miners could gain more profit by selecting the split function in C4 instead of C6. We finally note that, on the contrary, the results for makeWallet functions in both C3 and C10 are almost identical. Similar for approve functions in C7, C8, C13, C14, C15, C16 and C17.

3.5.3 Comparison Between Contract Creation and Function Execution Transactions

It is important to point out the significant difference in terms of the amount of used gas that can be collected by miners per each microsecond of CPU usage between contract creation and function execution transactions. The average amount of used gas awarded for contract creation transactions is nearly six times more than that for function execution transactions see Figures 3.1 and 3.2. That means miners could collect more profit by selecting and including contract creation transactions.

In conclusion, we have identified two types of discrepancies in terms of the reward for computational effort. First, both for contract creation and for function execution, the amount of used gas awarded is not consistently proportional to the CPU usage, there can be a factor of 5 difference in the creation of different contracts and a factor of almost 50 difference in the execution of different functions. Secondly, between contract

creation and function execution, there is a considerable difference in reward for CPU usage, roughly a factor 6. Miners gain more profit if they select transactions based on these two observations. We believe that this implies a threat to the dependable operation of the blockchain, for which it is important to ensure that the used gas awarded is proportional to the CPU usage for all types of operations that are required. The results in this chapter indicate that it would require modifications for Ethereum to establish a fair incentive model for miners as well as an appropriate cost model for the submitter of the transactions.

3.6 Conclusion

This chapter proposes a benchmarking approach to assess whether the fees miners receive from creating and executing smart contracts is proportional to the cost as expressed in terms of CPU usage. To illustrate our approach, we conducted a benchmarking study to investigate whether the used gas for creating smart contracts and executing contract functions is proportional to the computational effort. We created a benchmark of the 100 most commonly used smart contracts in Ethereum. Due to some technical limitations, we managed to benchmark the creation of 76 contracts and the execution of 21 functions.

Our results show that the used gas is not always proportional to the computational effort for both creating and executing smart contracts. More specifically, contract creation is about a factor 6 more profitable to miners than the execution of contract functions. Besides, some functions can be up to almost 50 times more profitable than others. This indicates that miners could gain more profit when creating and deploying new contracts instead of executing existing ones, and by being selective about which contract functions to execute. Potentially, this forms a threat to the dependability of the overall blockchain, in that some important computational tasks may be not sufficiently attractive for miners to dedicate their resources to.

Vice versa, transaction submitters, may offer different gas prices to compensate for the observed discrepancies, thus incentivizing miners more reasonably again. How to achieve this in practice is a topic for further research. With respect to our proposed benchmarking approach, additional refinement of the proposed methods can be envisaged, for instance considering computing effort beyond CPU usage (e.g. storage), and relating computational effort more directly to actual energy costs. Also, we thus far only conducted experiments on a single platform, and additional experiments are required to cover more contracts, functions, types of client codes, operating systems

and hardware. In the next chapter, we extend this work by looking more in-depth in terms of what is causing the high (or low) CPU overheads at a finer granularity within the specific functions by proposing a new performance benchmark system, which is the fist of its kind, named "OpBench " that assesses the CPU overheads for the EVM opcodes.

Chapter 4

Performance Benchmark of Blockchain Smart Contract Operation Code (Opcode)

4.1 Summary

Ethereum and other blockchains rely on miners to contribute computational power to execute tasks such as the proof of work consensus mechanism and the execution and validation of smart contracts. Miners receive a fee for their efforts, and for the correct operation of the blockchain, rewards should be proportional to the required investment (equipment, energy use, etc.). In Ethereum, the reward obtained for executing smart contracts is set statically, associating a fee with each operation code (opcode) in the smart contract. To determine whether fees are aligned with investments made, we propose OpBench, a platform-independent benchmarking approach.

OpBench measures the CPU time required to execute opcodes in the EVM. We implemented OpBench for the PyEthApp, the Go-Ethereum and the Parity clients running on Windows, Linux and Mac operating systems.

This chapter is organized as follows. In Section 4.2 we provide an introduction. The related work is presented in Section 4.3. Section 4.4 offers a detailed design framework for our proposed OpBench system. In Section 4.5, we present the implementation of our system in the PyEthApp, the Go-Ethereum and the Parity clients, and we conclude the chapter in Section 4.6.

4.2 Introduction

The proper functioning of permissionless blockchains such as Bitcoin [75] and Ethereum [94] depends on miners contributing their computing resources to the operation of the blockchain. In exchange for a fee (in the form of the cryptocurrency associated with the blockchain) miners are willing to invest, specifically, the electricity required to operate the blockchain. If the fee structure is fair, miners are incentivized to run the blockchain correctly and efficiently. However, if fees are not set fairly, miners may alter their behaviour, possibly to the detriment of the blockchain's operation [10]. In the worst case, fees that are set inappropriately may be vulnerable to misuse, for instance, exemplified by the denial of service attacks on Ethereum in 2016 [54, 19]. Therefore, for miners and for the successful operation of the blockchain, it is critical to understand the relation between the fee received and the cost incurred.

In Ethereum, when executing a smart contract, the fee a miner receives is determined by the *Gas* required to execute operation code (opcode), multiplied by a price the submitter of a transaction pays per unit of Gas. The Ethereum client (more precisely, the Ethereum Virtual Machine (EVM)) tallies the total Gas as it executes a smart contract. It does this based on values specified in a table in the Ethereum yellow paper [94], which statically associates an amount of Gas to each opcode. That means that the Gas used (and therefore the fee received) per smart contract is independent of the hardware or software used by the client; it directly follows from the opcodes in the smart contract.

From the miners' perspective, there are several implications of this static approach. The cost of executing smart contracts can be expected to be different on different computing platforms since the execution time of individual opcodes is likely to be different across platforms. As a consequence, a miner would want to choose a platform that optimizes the reward for the used energy. The benchmark presented in this chapter, when carried out for different platforms, will *help select the best platform.* Miners may also want to mine transactions based on optimizing the trade-off between the cost and reward. Our opcode benchmark would assist in deciding *which smart contracts to execute:* smart contracts with opcodes that add more Gas to the tally per CPU cycle are preferred since these result in higher rewards for the same investment.

Equally important is the perspective of the dependable operation of the blockchain: if reward and cost are not proportional across opcodes it could result in misalignment of mining incentives, e.g., [10] . Especially when Proof of Stake replaces Proof of

Work, the profit miners make will depend strongly on the investment made in executing smart contracts. Misalignment of incentives may result in miners operating the blockchain in a manner that is not optimal for the fair and effective long-term operation of the system. By conducting benchmarks on various platforms, one can adapt the value of the static fees and relate fees better to the CPU cost across most platforms.

For the above reasons, we introduce OpBench, the first CPU performance benchmark system for Ethereum smart contract opcodes. OpBench is the first systemic benchmark solution for opcodes we are aware of, and it provides a benchmark approach and benchmark results for all CPU-sensitive opcodes in Ethereum. The chapter introduces the design of OpBench, which is independent of the Ethereum client's language or operating system and is integrated with the Ethereum Virtual Machine architecture of Ethereum clients. To demonstrate the utility of the design, we developed three implementations of OpBench: for the PyEthApp client (in Python), the Parity client (in Rust) and the Go-Ethereum client (in Go).

Several challenges needed to be addressed in the design and implementation of OpBench. In particular, since individual opcodes take very little time to execute, OpBench executes opcodes repeatedly, taking care of stack management challenges that result from the small size EVM stack. We show in our implementations how to leverage the EVM and Python/Go/Rust libraries to measure performance accurately. To be of practical use, OpBench runs independently from the live Ethereum blockchain.

4.3 Related Work

Smart contract systems and its underlying technology, the blockchain, have been studied in depth for the last four years. A recent systematic survey [8] states that the most researched aspects of smart contract-based systems are in new applications and software engineering approaches, while performance and scalability are relatively less explored. To the best of our knowledge, there is no prior systematic approach suggested for performance benchmarking of Ethereum opcodes. The fee schedule in [94], which assigns Gas to operation codes, is based on classifying opcodes in categories of high, medium, etc., but does not provide a basis for that classification.

Related work includes the work of Dinh et al. [31], which proposes an evaluation framework (BLOCKBENCH) to measure the latency, throughput, fault-tolerance, and scalability of the private blockchain. BLOCKBENCH allows for performance

comparison of diverse blockchains, including Hyperledger and Ethereum, but it does not provide a detailed performance benchmark at the granularity of opcodes. Another benchmark approach has been suggested by [6], proposing a performance benchmark for Ethereum smart contracts. They found that used Gas is often not proportional to the computational effort for both contract execution and contract creation. Their performance benchmark was designed for the smart contract level, as opposed to the opcode level. A limitation of benchmarking at the level of smart contracts is that results for one contract do not extend to others and that therefore, every contract needs to be benchmarked separately. Chen et al. [24] conducted an experiment that records the time consumption of a CPU against executing the opcodes. Their results show that the consumed Gas is not proportional to the CPU usage, but they did not attempt to create a benchmark from their work; instead, their interest is in adaptive schemes to deal with fees that are not proportional to the computing requirements. Chen et al. [24] then proposed a tool called GASPER that automatically locates Gas-costly patterns by analyzing the smart contract bytecode. Their tool analyses the bytecode and reports the miss-programming patterns that cause a high Gas cost, such as unused code patterns and loop patterns. In [97], the authors introduced a tool called (GasReducer) that detects sub-optimal code in the smart contracts' bytecodes and replaces it with sufficient bytecodes to reduce unnecessary Gas.

In conclusion, although several interesting efforts in understanding and improvement of smart contract performance exist, none of these efforts proposes an opcode level benchmark as we do in this chapter.

4.4 Design of OpBench System

In this section, we describe the design of our benchmark system, OpBench. This proposed design can be implemented in various languages depending on the client's specifications. First, we present the workflow of our system. Next, we categorize all the opcodes, and discuss how we propose to conduct experiments for different opcode categories e.g. how to deal with parameter dependence. In the rest of this section, we present the details of the benchmark operations that we run for various opcodes. These includes how to deal with the limited stack size and how to calculate the execution time per opcode.

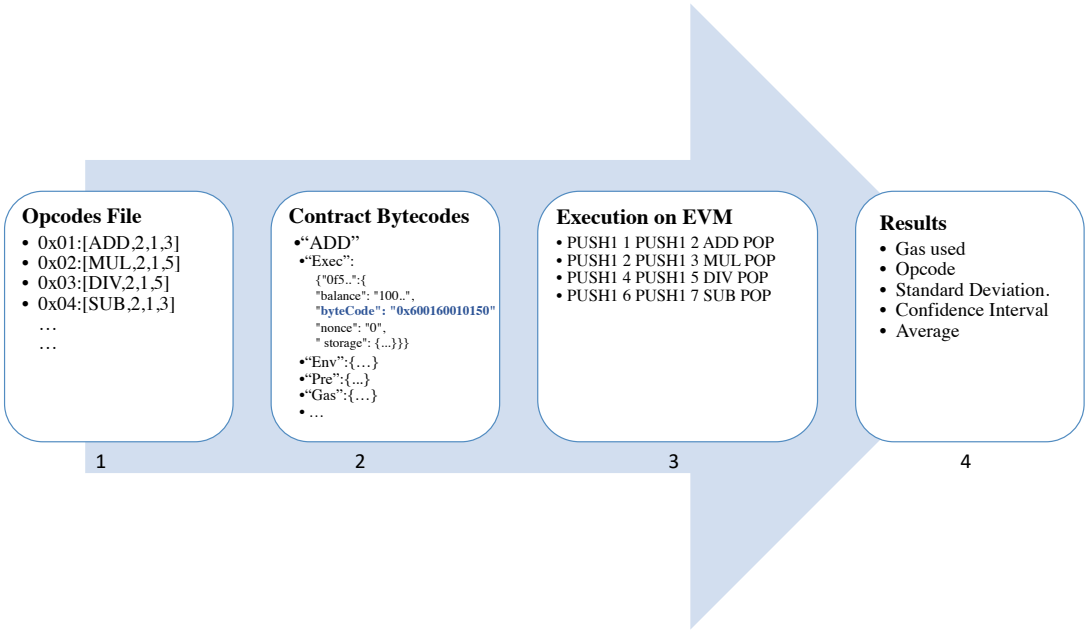

Figure 4.1: OpBench overview.

4.4.1 OpBench Overview

The usage of OpBench can be divided into four Phases, as depicted in Figure 4.1. In the first phase, we utilize PyEthApp [46] client to identify all opcodes, associated Gas prices, as well as the number of required input and output parameters for executing the opcode. For example, the `ADD` opcode ([0x01:[`ADD,2,1,3`]]) requires two inputs and one output, and it costs 3 unit of Gas. The $0x01$ represents the value of the opcode in the hexadecimal representation which can only be interpreted by the EVM. Note that also if the EVM is in a different language like Go for Go-Ethereum client or Rust for Parity client, this phase can be done in Python because the generated bytecode can be executed in any Ethereum clients. In the official implementation of the PyEthApp [33], all available EVM's opcodes are located in a single file called *opcode*[1].

In phase 2, for each opcode from phase 1, we generate the bytecode for a fully executable smart contract, which contains repeated bytecode instances of the opcode intended to be measured, as well as the required `PUSHs` and `POPs` opcodes to successfully manipulate the EVM stack. This is depicted under "Contract Bytecode" in Figure 4.1, showing generated contract account's state, including account information, environment information as well as execution information and extra information

[1] `https://github.com/ethereum/py-evm/tree/master/eth/vm`

such as the block Gas limit.

```
1
2     from Ethereum import opcodes
3     ...
4     push_params = [range(1, 3), range(4, 8), range(9, 16), range(17, 32)]
5     codesize_params = [[100000, 100000]]
6
7     def generate_op_tests():
8         out = {}
9         for opcode, (name, inargs, outargs, _) in opcodes.items():
10            _subid = 0
11            for push_depths in push_params:
12                for jump_num, code_size in codesize_params:
13                    if name in ['DELEGATECALL', 'LOG0', ..]:
14                        continue
15                    if name[:4] == 'PUSH':
16                        if push_depths != push_params[0]:
17                            continue
18                        for i in range(code_size):
19                            v = int(name[4:])
20                            w = random.randrange(256**v)
21                            c += chr(0x5f + v) + utils.zpad(utils.encode_int(w), v)
22                    ....
23                    o = o = {
24                    "Exec": {
25                        "0f572e5295c57f15886f9b263e2f6d2d6c7b5ec6": {
26                            "balance": "1000000000000000000",
27                            "byteCode":"0x"+c.encode('hex'),
28                            "nonce": "0",
29                            "storage": {}
30                        }
31                    },
32                    "Env": {..},
33                    "Pre": {..},
34                    "Gas": "1000000000",
35                    "Logs": [],
36                    "Out": "0x"}
37                    ...
38    open('ByteCode.json', 'w').write(json.dumps(generate_op_tests(), indent=4))
```

Listing 4.1: Generating Ethereum smart contract bytecode (Phase 2 in Figure 4.1).

Listing 4.1 shows a code snippet used to generate bytecode for each opcode of the EVM. In the first line we import the file that contains these opcodes then we loop 100k times (Line 12) to create bytecode for each actual opcode. 100k time was chosen because it provides a tight confidence interval (95%-Confidence Interval $\approx$ 0.005). Hence, the bytecode contains 100k actual opcodes. As mentioned above, the generated bytecode has its all required input parameters and other related opcodes (i.e., in most cases `PUSHs` and `POPs`) (Lines 15-21). Later in (Lines 24-36), we generate the executable test that consists of the contract account configuration such as address (Line 25), balance (Line 26), the generated and the executable bytecode (Line 27), nonce (Line 28) and finally the contract's storage (Line 29). Finally, in (Lines 32-36), we define the EVM's state configurations that are needed to generate transactions to deploy and execute the bytecode (more details in Phase 3).

For opcodes for which the size of their parameters could impact the computation overhead, we generate different versions of the bytecode, each with different size (64-bits, 128-bits, and 256-bits). This is due to the fact that depending on the implementation the smaller sizes can be computed more quickly and the stack limit size is 256-bit (1 word) [94] [68]. These opcodes belong to the first and second categories in Table 4.1. For opcodes that are not sensitive to the input parameter,

we only create one version based on 256-bits, unless otherwise specified. An example is `BALANCE`, which returns the Ether balance of an address, where the size of that address is always 32 bytes.

Phase 3 is the heart of OpBench system. In this phase, we setup the blockchain configuration, account details and the Ethereum state configurations. Then, we deploy the smart contract's bytecode on the blockchain by submitting an Ethereum transaction. Next, we initiate a transaction that executes the deployed bytecode on the EVM, playing the roles of both the sender and miner. In the example (`PUSH1 1 PUSH1 2 ADD POP`), after setting-up the blockchain, the state and the account, the EVM runs the `ADD` opcode by pushing two input parameters (i.e., 1 and 2) onto the stack (`PUSH1 1 PUSH1 1`) then adding these numbers (`ADD`), and finally popping the result (`POP`) from the stack. Moreover, the execution times as well as the Used Gas of each opcode are collected and exported as *json* files. Later in Section 4.5, we will show in details the real implementation of this phase on the three EVM clients.

The computation time of each bytecode is recorded in Phase 4. Listing 4.2 presents the final results of the phase 4 of OpBench in *json* file. We collect all the results and calculate the average, standard deviation and confidence interval for each opcode to be able to report the final results and their accuracy. In Figure 4.1, the result of the execution of an opcode is reported as a *json* formatted file that shows the used Gas, the name of the opcode standard deviation and confidence interval. The confidence interval is calculated by collecting each run's execution time and storing it in a list data type. Then, we used the build-in methods for each client, *SiPy*[30] in Python, for instance. We are only concerned with the Gas used and the execution time average. The other statistics is presented to check the correctness of OpBench execution.

```
[ {     "Used Gas": 3.0,
        "Opcode": "ADD",
        "Appearance": 100000,
        "Upper bound": 0.8092592672921111,
        "Standard Deviation": 0.26486772279105025,
        "Lower bound": 0.7988753863939606,
        "95% Confidence Interval": 0.0051919404490752274,
        "Mean": 0.8040673268430183
    }
]
```

Listing 4.2: The Final output of OpBench (Phase 4 in Figure 4.1).

4.4.2 Workload: Classification of Opcodes

Different opcodes require different treatment when benchmarking their performance. Following the approach of [94], we distinguish between computation-based opcodes and formula-based ones. The computation-based opcodes have a static constant

amount of Gas as the fee, as defined in [94]. The formula-based opcodes have more intricate performance dependencies and therefore are more involved when designing the benchmark. We describe our approach for both types of opcodes in the following sections.

A third type of the opcodes have an associated fee that is not motivated by CPU usage, and it is therefore not suitable for OpBench. These opcodes fall in the system and log opcode categories (see last two categories in Table 4.1), as described in [94]. For example, `CREATE`, which creates a new account with an associated code, has a fee of 32k Gas is for creating (not running) the bytecode. This is not bound to the CPU usage. Similarly, log opcodes append a log record to the blockchain, and the cost associated with these opcodes is motivated by the disk usage.

4.4.2.1 Computation-based Opcodes

This class of opcodes includes but not limited to the ones in the Stop and Arithmetic operations, Comparison & Bitwise Logic Operations, using the categories identified in [94]. Table 4.1 provides a list of all opcodes, the category names associated with each opcode, the Gas cost, the required inputs and outputs, the classification of each opcode either formula or static, their hexadecimal representation and their descriptions. OpBench runs the experiments with three different sizes of parameters (64-bits, 128-bits, and 256-bits). For instance, consider the `ADD` opcode, which has associated Gas of 3. Our benchmark includes three entries for `ADD`, with three different sizes of parameters: `ADD64, ADD128` and `ADD256`. This is because depending on the implementations the smaller sizes can be computed more quickly [68].

In the opcode categories for which inputs do not impact the computation overhead (*Push Operations, Exchange Operations, Duplication Operations*), each opcode is benchmarked with a single size of parameter (256-bits). For these opcodes, in the generated bytecode in phase 1, we push the required parameters onto the stack using `PUSH` opcodes, and then we execute the actual opcode. Some opcodes such as `ADD` pushed the execution results onto the stack as well. In these cases, we use the `POP` opcode to maintain the stack's size of at most 1024 bytes, to be able to execute a high numbers of opcodes.

4.4.2.2 Formula-based Opcodes

The formula-based opcodes are more intricate to benchmark, so we describe these in more detail in this section. 11 out of 150 opcodes belong to this type [94] of which

we explain six representative ones here.

$$Gas = \begin{cases} 10, & \text{if } EXP = 0 \\ 10 + 50 \times (1 + \log 256(EXP)), & \text{if } EXP > 0 \end{cases} \tag{4.1}$$

The `EXP` opcode is the exponential operation, and its formula is shown in Equation (4.1). `EXP` opcode pops two values from the stack, the base and the exponent and calculates the exponential. Then, it pushes the result onto the stack. The Gas cost of the `EXP` is zero if the exponent zero, other wise it is $10 + 50\times$ a factor related to the size of the log of the exponent (see Equation 4.1). For this opcode, we repeat the experiments with different exponent sizes (64-bits 128-bit and 256-bits).

$$Gas = \begin{cases} 30, & \text{if } input = 0 \\ 30 + 6 \times (sizeofinputinwords), & \text{if } input > 0 \end{cases} \tag{4.2}$$

The Gas price for the `SHA3` opcode which computes the Keccak-256 hash for any input is calculated according to Equation 4.2. The `SHA3` opcode's input parameters are the memory offsets and size of the value intended to be hashed.

Note that the memory in the EVM architecture refers to a special memory area where the contract gets fresh instances for variables. The `SHA3` opcode is the only opcode that retrieves its input parameters from both the stack and the memory [94]. We store the value we wish to hash in the memory and push its location and size onto the stack, and then we perform the `SHA3` opcode. In OpBench design, we benchmark this opcode with various parameters sizes that are 1, 2, 3 and 4 words, and each word is 32-bytes.

$$Gas = \begin{cases} 20,000, & \text{if } input \neq 0 \ \&\& \ storage = \emptyset \\ 5000, & \text{otherwise} \end{cases} \tag{4.3}$$

For the `SSTORE` opcode that stores a vale of word size into the contact's storage. The `SSTORE` is the only opcode that update the storage. We benchmark this opcode by setting the contract's storage to a non-zero value from a zero value and updating the current storage as well as setting the contract's storage to a zero value from a non-zero value. The storage in the Ethereum refers to a persistent memory area. The Gas cost of the `SSTORE` opcode is $20k$ units if the storage is set to a non-zero value from a zero value, and $5k$ units for storage update as shown in Equation (4.3).

$$Gas = \begin{cases} 2, & \text{if } \#word = 0 \\ 2 + 3 \times (\# \ words), & \text{if } \#word > 0 \end{cases} \tag{4.4}$$

The `CALLDATACOPY` and `CODECOPY` opcodes copy the the transaction's input data of as explained in Figure 2.4 in the current environment and the current running

code in the EVM to the memory, respectively. The `CALLDATACOPY` opcode pops the transaction's input size out from the stack and copies the transaction's input to the memory. For this opcode, two sizes of parameters are used 1 and 2 words. The `CODECOPY` opcode also pops the current executing code from the stack and copies the popped code into the memory. Similar to `CALLDATACOPY`, we execute this opcode with different transaction's input data and different code sizes (i.e. 1 and 2 words). The Gas calculation formulas for these opcodes are calculated according to Equation (4.4).

$$Gas = \begin{cases} 700, & \text{if } \#word = 0 \\ 700 + 3 \times (\#\ words), & \text{if } \#word > 0 \end{cases} \tag{4.5}$$

The `EXTCODECOPY` opcode is to copy an account's code to the memory. The main difference between this opcode and `CODECOPY` is that this opcode copy the contract's code from the the Ethereum's state and the later from the EVM. It pops the account's address, the code's start point, and the size of that code out from the stack and searches on the Ethereum's state (i.e. blockchain database), then copies the code to the memory. To benchmark this opcode, we push the required parameters into the stack and execute the actual opcode. This opcode is executed in three different sizes (1, 2 and 3 words). The Gas cost formula is shown in 4.5. We run this opcode with different code's size to report the results.

4.4.3 Manipulating the Stack

OpBench needs to determine the execution time for a single execution of each opcode from the execution of a smart contract that repeatedly executes the opcode. When executing each opcode many times, we need to resolve two challenges: a) the stack is limited, and b) `POP` and `PUSH` operations are needed to manipulate the stack, and these need to be removed from the total execution time.

Limited Stack Size. The EVM stack has a maximum size of 1024 [94] and a stack limit exception occurs when the stack size reaches 1024. Figure 4.2 illustrates an example of executing the `ADD` opcode on the EVM and how the stack is utilized. To execute the `ADD` opcode, two values are pushed onto the stack, then the `ADD` opcode is executed by popping the values from the stack. The result of the execution is pushed back onto the stack (see step 1 in the Figure 4.2). If we want to repeat the `ADD` until the stack becomes full (i.e., more than 1024 times), it would result in an EVM exception. We overcome this by utilizing the `POP` opcode that pops the result of the execution from the stack (see step 2 in the Figure 4.2 [`POP 3`]). Therefore, the stack

Category	Operation Code	Gas Cost	Value (hex)	Add to Stack	Remove from Stack	Description
Arithmetic & Stop Operations	STOP	0	0x00	0	0	Halts execution.
	ADD	3	0x01	1	2	Addition operation
	MUL	5	0x02	1	2	Multiplication operation.
	SUB	3	0x03	1	2	Subtraction operation.
	DIV	5	0x04	1	2	Integer division operation.
	SDIV	5	0x05	1	2	Signed integer division operation (truncated).
	MOD	5	0x06	1	2	Modulo remainder operation
	SMOD	5	0x07	1	2	Signed modulo remainder operation.
	ADDMOD	8	0x08	1	3	Modulo addition operation.
	MULMOD	8	0x09	1	3	Modulo multiplication operation.
	EXP	FORMULA	0x0a	1	2	Exponential operation.
	SIGNEXTEND	5	0x0b	1	2	Extend length of two's complement signed integer.
Comparison & Bitwise Logic Operations	LT	3	0x10	1	2	Less-than comparison.
	GT	3	0x11	1	2	Greater-than comparison.
	SLT	3	0x12	1	2	Signed less-than comparison.
	SGT	3	0x13	1	2	Signed greater-than comparison.
	EQ	3	0x14	1	2	Equality comparison.
	ISZERO	3	0x15	1	1	Simple not operator.
	AND	3	0x16	1	2	Bitwise AND operation.
	OR	3	0x17	1	2	Bitwise OR operation
	XOR	3	0x18	1	2	Bitwise XOR operation.
	NOT	3	0x19	1	1	Bitwise NOT operation.
	BYTE	3	0x1a	1	2	Retrieve single byte from word
SHA3	SHA3	FORMULA	0x20	1	2	Compute Keccak-256 hash.
Environmental Information	ADDRESS	2	0x30	1	0	Get address of currently executing account.
	BALANCE	400	0x31	1	1	Get balance of the given account.
	ORIGIN	2	0x32	1	0	Get execution origination address.
	CALLER	2	0x33	1	0	Get caller address.
	CALLVALUE	2	0x34	1	0	Message funds in wei.
	CALLDATASIZE	2	0x36	1	0	Message data length in bytes.
	CALLDATACOPY	FORMULA	0x37	0	3	Copy message data.
	CODECOPY	FORMULA	0x39	0	3	Copy executing contract's bytecode.
	GASPRICE	2	0x3a	1	0	Get price of Gas in current environment.
	EXTCODESIZE	700	0x3b	1	1	Get size of an account's code.
	EXTCODECOPY	FORMULA	0x3c	0	4	Copy an account's code to memory.
Block Information	BLOCKHASH	20	0x40	1	1	Hash of a specific block.
	COINBASE	2	0x41	1	0	Get the block's beneficiary address.
	TIMESTAMP	2	0x42	1	0	Get the block's timestamp.
	NUMBER	2	0x43	1	0	Get the block's number.
	DIFFICULTY	2	0x44	1	0	Get the block's difficulty.
	GASLIMIT	2	0x45	1	0	Get the block's Gas limit.
Stack, Memory, Storage and Flow Operations	POP	2	0x50	0	1	Remove item from stack.
	MLOAD	3	0x51	1	1	Load word from memory.
	MSTORE	3	0x52	0	2	Save word to memory.
	MSTORE8	3	0x53	0	2	Save byte to memory.
	SLOAD	200	0x54	1	1	Load word from storage.
	SSTORE	FORMULA	0x55	1	1	Save word to storage.
	JUMP	8	0x56	0	1	Alter the program counter.
	JUMPI	10	0x57	0	2	Conditionally alter the program counter.
	PC	2	0x58	1	0	Program counter.
	MSIZE	2	0x59	1	0	Get the size of active memory in bytes.
	GAS	2	0x5a	1	0	Remaining Gas.
	JUMPDEST	1	0x5b	0	0	Mark a valid destination for jumps.
Push Operations	PUSH*	3	0x60 – 0x7f	1	0	Place * byte item on stack. $0 < * \leq 32$
	DUP*	3	0x80 – 0x8f	* + 1	*	Duplicate *th stack item. $0 < * \leq 16$
	SWAP*	3	0x90 – 0x9f	* + 1	* + 1	Exchange 1st and (* + 1)th stack items.
Logging Operations	LOG0	FORMULA	0xa0	0	2	Append log record with no topics.
	LOG1	FORMULA	0xa1	0	3	Append log record with one topic.
	LOG2	FORMULA	0xa2	0	4	Append log record with two topics.
	LOG3	FORMULA	0xa3	0	5	Append log record with three topics.
	LOG4	FORMULA	0xa4	0	6	Append log record with four topics.
System operations	CREATE	32000	0xf0	1	3	Create a new account with associated code.
	CALL	FORMULA	0xf1	1	7	Message-call into an account.
	CALLCODE	FORMULA	0xf2	1	7	Call a method in another contract.
	RETURN	0	0xf3	0	2	Halt execution returning output data.
	DELEGATECALL	FORMULA	0xf4	1	6	calls a method in another contract, using this contract's storage.
	INVALID	NA	0xfe	NA	NA	Designated invalid instruction.
	SELFDESTRUCT	FORMULA	0xff	0	1	Halt execution.

Table 4.1: List of all operation codes (opcodes) in the Ethereum Virtual Machine [94].

is empty after each execution. The top part of Figure 4.2 shows that without using the `POP` opcode, we would end up with a stack limit exception after executing 1024 opcodes. The bottom part illustrates that with our proposed technique (i.e., with `POP`) we are able to execute any desired number of opcodes.

Removing `POP` and `PUSH` Overhead. Almost all the EVM's opcodes require at least one element to be retrieved from the stack. That is, one or more `PUSHs` are needed to fill the stack and one or more `POPs` to retrieve parameters. The execution of `PUSH` and `POP` affects the overall execution time, so we need to differentiate between the CPU time used by the opcode of interest and that for the stack operations. However, the EVMs provide very high granularity timing support allowing to set a timer before and after the execution of each opcode on the EVM. The output result of OpBench is a list of opcodes and their execution times (see Listing 4.2). Hence, in this case the execution times of `PUSH` and `POP` are separated from the actual opcode, in this

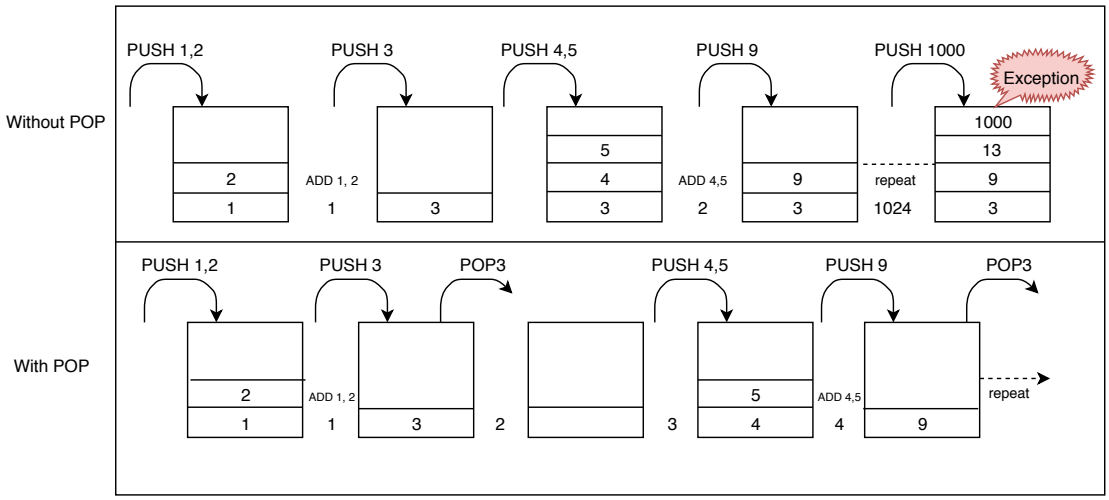

Figure 4.2: Utilizing the `POP` opcode to overcome the stack size limitation.

case the `ADD` opcode. The accuracy of setting the timer is achieved by utilizing the *Profile* module in Python, which uses the platform-specific time function to provide the most accurate time calculation possible [59]. In this way, we get the most accurate measurement of the execution time. In Golang, we make use of the build-in *benchmark* module in [32], with the same effect.

4.5 Implementation

In this section, we present the implementations of our OpBench system for three Ethereum clients: PyEthApp [46], Go-Ethereum [44] and Parity [45]. We choose Go-Ethereum and Parity clients since they are the most popular platforms [52] and PyEthApp because the initial Gas allocation for opcodes is based on a benchmark performed on the PyEthApp client "[personal communication with Vitalik Buterin, 2016]".

4.5.1 PyEthApp

OpBench system in PyEthApp has two phases. In the first phase, we collect all *json* files generated by phase 1 of OpBench. In the second phase, presented in Listing 4.3, the contract's bytecodes are executed on the PyEthApp client. The execution process of the contract's bytecode involves setting up the blockchain configurations (Line 11-16), state configurations (Line 17-21), the contract's account configuration (Line 22-30) and the contract deployment transaction (Line 34). All the configurations data are retrieved from the *json* files generated by phase 1.

Next, after preparing the blockchain and Ethereum's state as well as deploying the contract's bytecodes in the blockchain, we execute the contract's bytecodes on the EVM by submitting a transaction (Line 45). The execution of the bytecodes is done in a serialized manner. Thus, the EVM loops over the bytecodes and execute each opcode separately. By utilizing the *Profile* module provided by Python, which according to [59] provides a statics analysis about the the CPU performance, we store each loop's execution time in a data type list (Line 48). Finally, we calculate the upper bounds, the lower bounds, the confidence interval and the standard deviation for each opcode using *Statistics, SciPy* modules [29, 30] and export the final results (Line 52). The final results are exported as a *json* file that looks like Listing 4.2.

```
1  import transactions
2  import block
3  from vm import vm_execute, Message, CallData
4  ...
5  def profile_vm_test(params, _):
6      # file = open("Extime.csv", "a")
7      pre = params['pre']
8      exek = params['exec']
9      env = params['env']
10     # Setting up the Blockchain configurations i.e., Genesis block, difficulty, gas limit,
       coinbas, etc. lines 11 to 30
11     blkh = block.BlockHeader(prevhash=env['previousHash'].decode('hex'), number=int(env['
       currentNumber']),
12                              coinbase=env['currentCoinbase'],
13                              difficulty=int(env['currentDifficulty']),
14                              gas_limit=int(env['currentGasLimit']),
15                              timestamp=int(env['currentTimestamp']))
16     block.Block(blkh, db=_env)
17     state = State(env=_env, block_number=int(env['currentNumber']),
18                   block_coinbase=env['currentCoinbase'],
19                   block_difficulty=int(env['currentDifficulty']),
20                   gas_limit=int(env['currentGasLimit']),
21                   timestamp=int(env['currentTimestamp']))
22     for address, h in pre.items():
23         state.set_nonce(address, int(h['nonce']))
24         state.set_balance(address, int(h['balance']))
25         state.set_balance("cd1722f3947def4cf144679da39c4c32bdc35681", int(h['balance']))
26         state.set_code(address, h['code'][2:].decode('hex'))
27         for k, v in h['storage'].iteritems():
28             state.set_storage_data(address,
29                                    u.big_endian_to_int(k[2:].decode('hex')),
30                                    u.big_endian_to_int(v[2:].decode('hex')))
31     #  setting up and signing the Ethereum transaction to deploy the byteCode.
32     sender = exek['origin']  # a party that originates a call
33     recvaddr = exek['address']
34     tx = transactions.Transaction(
35                 nonce=state.get_nonce(exek['caller']),
36                 gasprice=int(exek['gasPrice']),
37                 startgas=int(exek['gas']),
38                 to=recvaddr,
39                 value=int(exek['value']),
40                 data=exek['data'][2:].decode('hex'), r=1, s=2, v=27)
41     tx._sender = sender
42     ext = pb.VMExt(state, tx)
43     msg = Message(tx.sender, tx.to, tx.value, tx.startgas, CallData([ord(x) for x in tx.data])
       )
44     # Blockchain transaction to execute the byteCode..
45     success, gas_remained, comStack, ListOp = vm_execute(ext, msg, exek['code'][2:].decode('
       hex'))
46     state.commit()
47     # Opcodes and their executing times for each
48     time, ops = [x['Time'] for x in ListOp if x['Time'] == 'T'], [x['op'] for x in ListOp if x
       ['OpC'] == 'Op']
49     ...
50 # Write the output results to finalResult file
51 open('finalResult.json', 'w').write(json.dumps(prepare_files(recursive_list(sys.argv[1])),
       indent=4))
```

Listing 4.3: Executing the contract's bytecode generated by phase 2 in PyEthApp.

4.5.2 Go-Ethereum

Listing 4.4 shows a part of OpBench implementation in Golang. In order to benchmark opcodes in Golang, a separate method for each opcode must be implemented. This is not the case in Python where we can benchmark all opcodes in a single run by reading bytecodes from the generated *json* files (Phase 2 in Figure 4.1). In Listing 4.4, we show the benchmark method for the `ADD256` opcode line 32. Hence, for opcodes which require inputs with different sizes, we need to implement a separate method for each of them (i.e., methods for `ADD64` and `ADD128`, etc.).

In the Go-Ethereum benchmark implementation, as mentioned above, we use the contract's bytecodes created by the Python script (Phase 2 of Figure 4.1) (Line 13), and then utilize the *benchmark* module provided by Golang. The *benchmark* module take all the required parameters and execute the opcode on the EVM (Lines 22-28). In the Listing (Lines 12-15), we generate the transactions that deploy and execute the bytecodes by retrieving the contract account state as well as the EVM state configurations from the *json* files. Finally, the execution transaction is executed in (Line 28).

The execution of the opcode is repeated until the desired benchmark runtime is reached (default one second) [21]. We set the time high enough (10ns) so that opcodes are executed sufficiently often and we check the accuracy by calculating a confidence interval. Golang provides statistical results about the execution time and the confidence interval as well as the standard deviation [21].

```
1  package vm
2  import (
3    "github.com/ethereum/go-ethereum/core"
4  )
5  func opBenchmark(bench *testing.B, op func(pc *uint64, evm *EVM, contract *Contract, memory *
       Memory, stack *Stack) ([]byte, error), args ...string) {
6      memory.store = common.Hex2Bytes("0x..")
7      contract := NewContract(AccountRef(common.HexToAddress("05600160010150..")), AccountRef(
       common.HexToAddress("x33333322")), new(big.Int), 1000)
8    env.StateDB.AddBalance(common.HexToAddress("1233"), big.NewInt(1000))
9    env.StateDB.AddBalance(common.HexToAddress("0x1000000"), big.NewInt(1000))
10   byteArgs := make([][]byte, len(args))
11   for i, arg := range args {byteArgs[i] = common.Hex2Bytes(arg)}
12   pc := uint64(0)
13     bench.ResetTimer()
14     for i := 0; i < bench.N; i++ {
15         for _, arg := range byteArgs {
16             a := new(big.Int).SetBytes(arg)
17             stack.push(a)
18     }
19     bench.StartTimer()
20     op(&pc, env, contract, memory, stack)
21     bench.StopTimer()
22     stack.pop()
23     }
24 }
25 func BenchmarkOpAdd256(b *testing.B) {
26   x := "0802431afcbce1fc194c9eaa417b2fb67dc75a95db0bc7ec6b1c8af11df6a1da9"
27   y := "a1f5aac137876480252e5dcac62c354ec0d42b76b0642b6181ed099849ea1d57"
28   opBenchmark(b, opAdd, x, y)
29 }
```

Listing 4.4: The Golang implementation of OpBench (part of it).

4.5.3 Parity

As stated in Subsection 2.4.5 , the Parity client implements the Ethereum protocol in Rust [87]. Similar to Python, benchmarking in Rust is not as simple as it is in Golang. Consequently, our benchmark implementation uses the bytecodes created by phase 1 for each actual opcode and using the Solidity compiler, we create for each opcode file a rust executable file. The solidity compiler takes the bytecode of each actual code as inputs and outputs an executable bytecode for the EVM. For each test, we loop over 2 million times of a group of 160 the actual opcode, along with will some required `PUSHs` and `POPs`. So the actual opcode is executed 320,000,000 times. As discussed in Subsection 4.4.3, the reason for introducing the `PUSHs` is to get a value on the stack to execute, the `POPs` empty the stack and start a new run after executing the previous run. Similar to the other clients, the numbered being benchmarked keep the inputs executed on with 64, 126 and 256-bits. We benchmark each opcode with different sizes because depending on the implementation the smaller sizes can be computed more quickly [68].

All files generated by the Python script (Phase 1 in Figure 4.1) are compiled using the Solidity compiler, then are executed on the Parity client as a smart contract bytecode. Thus, the generated bytecode can be easily deployed and run on the Ethereum blockchain. The contract's bytecodes are deployed and executed utilizing the debugging and testing tool (EVMBIN) that is provided by Ethereum[2]. Acconding to [42], the EVMBIN tool takes the Ethereum state file (*json*) generated by phase 1 and similar to the other clients it configures the blockchain, state, and the contract's account. Then it deploys and executes the bytecodes. The EVMBIN reports the execution time and the Gas used. To remove the overhead of `PUSH` and `POP` opcodes, We use some Python scripts [3] to analysis the result and to isolate the overhead of `PUSHs` and `POPs` from the actual opcodes. The scripts subtract the overhead of the start-up and the shutdown and the overhead of both `PUSHs` and `POPs` opcodes from the overall execution time. Therefore, the results of the benchmark are processed to be accurate.

Listing 4.5 depicts the implementation snippet of OpBench system in the Parity client. In (Lines 8-12), we utilize the Solidity compiler to compile phase 1 outputs into an executable file that can be run on parity by the EVMBIN. The outputs of the Solidity compiler are listed in (Lines 17-21). The contract's bytecode deployment and execution are presented in (Line 3). The execution outputs, which has the execution

[2] `https://github.com/paritytech/parity-ethereum/tree/master/evmbin`
[3] `https://github.com/Amjad13?tab=repositories`

time for each opcode and its Used Gas , are export as a log file. We use a Python script that converts the log file into a *csv* file. Thus, the final execution outputs of the Parity client implementation looks like Listing 4.2.

```
1
2  ifdef PARITY
3    PARITY_ = $(call STATS,parity) $(PARITY) --gas 1000000000000000 --code `cat $*.bin`; touch $
       *.ran
4  endif
5  STATS = time -p
6  %.ran : %.bin
7    $(call PARITY_)
8  %.ran : %.c
9    gcc -O0 -S $*.c
10   gcc -o $* $*.s
11   $(call STATS,C) ./$*
12   touch $*.ran
13 .PRECIOUS : %.bin
14 %.bin : %.sol
15   $(call SOLC_SOL_)
16 all : ops programs
17 ops : \
18   add256.ran \
19     mul256.ran \
20     sub256.ran \
21   div256.ran \
22   ...
23 rerun :
24   rm *.ran
```

Listing 4.5: The Parity implementation of OpBench (part of it).

4.6 Conclusion

In this chapter we presented OpBench, an Ethereum performance benchmark system for smart contract operation code and, to the best of our knowledge, the first of its kind. OpBench assesses, for each opcode, the CPU effort required by the EVM for its execution. We implemented OpBench for three different clients, Python-based PyEthApp, Rust-based Parity and Go-based Go-Ethereum.

The work demonstrates the feasibility to benchmark opcodes, discussing both design and implementation. The results obtained from OpBench, presented in the following chapter, establish if the award received when executing smart contracts is proportional to the cost of executing. This is important for several reasons described in the chapter's introduction: possible future adjustment of advertised fees associated with opcodes, selection of contracts based on opcodes present within the smart contract and to help select hardware and software configurations that optimize the return on investment.

Chapter 5

Experimental Results and Discussion

5.1 Summary

In the previous chapter, we have presented a details design framework and the implementations for our proposed OpBench system in three different Ethereum clients, Python-based PyEthApp, Rust-based Parity and Go-based Go-Ethereum. As we know, OpBench is the first of it kind that assesses the invested CPU overheads required by the EVM for its execution for each opcode.

In this chapter, we present two experimental results, one for six system configurations, running two different implementations using three different hardware platforms (and operating systems), and the other for six system configurations, running three different implementations using the *same* hardware platform. These experiments demonstrate the validity of OpBench approach across platforms. It also allows us to compare PyEthApp, Go-Ethereum and Parity, with respect to CPU usage for various opcodes as well as for the fee rewarded per unit of CPU time and it allows us to compare operating systems, in particular, Linux and Windows.

The results show that the static fees set by Ethereum are not always proportional to the invested CPU time, with up to an order of magnitude difference across opcodes. The results also show a markable difference in performance between clients, with the Parity client outperforming the other clients across machines and OS configurations. Moreover, the results show that the Windows systems outperform the other system in all clients.

The structure of this chapter is as follow. In Section 5.2, we present the first set of experiments for two Ethereum clients on different hardware platforms. The second

set of experiments for three Ethereum clients is presented in Section 5.3. Section 5.4 provides the validation of the results, and we conclude the chapter in Section 5.5.

5.2 First Experimental Results

We conduct our experiments using three different machines listed in Table 5.1. Having three different clients (Go-Ethereum, Parity and PyEthApp), our results concern nine different platforms, denoted as Windows Go-Ethereum, Windows Parity, Windows PyEthApp, Linux Go-Ethereum, Linux Parity, Linux PyEthApp, Mac Go-Ethereum, Mac Parity, and Mac PyEthApp. Note that our experiments aim to compare the Go-Ethereum, Parity and PyEthApp clients on different platforms, we do not aim to compare operating systems. In Section 5.3, we consider operating systems by comparing three clients running on the same hardware platforms.

Machine	MacBook Pro 2.8GHz	Desktop 3.2GHz	Desktop 3.6GHz
CPU	Intel i5 2.8 GHz	Intel i7 3.20GHz	Intel i7 3.60GHz
Cores	2	4	6
Memory	8GB	16GB	32GB
OS	MacOS 10.14.6	Ubuntu 16.04.3 LTS	Windows 10

Table 5.1: Experimental 1 platforms.

The performance benchmark results for all opcodes are provided in Table 5.2, in microseconds (μs). We note that all confidence intervals are exceedingly tight (95%-Confidence Interval $\approx 0.005\mu s$ in average), and are not provided in the table. As explained in Section 4.5, for PyEthApp, we executed all opcodes 100k times, collecting the average time and calculating a confidence interval. In Go-Ethereum, each opcode is executed until the desired benchmark time is reached, typically resulting in yet more samples than in PyEthApp, and again, a very tight confidence interval.

We will discuss the benchmark results in stages. First, we consider the CPU time itself, in absolute value and relative to the fastest platform, respectively. Then we discuss the ratio of the Used Gas and CPU usage in order to identify the reward for the invested CPU time. We summarize the main insights that follow from our discussion up front:

- Parity and Go-Ethereum clients are generally considerably faster (more profitable) than the PyEthApp client, regardless of the machine.

- The performance varies across the clients and the operating systems. For example, the Linux 3.2 GHz machine outperforms the Windows 3.6 GHz machine on the PyEthApp client, while the reverse is true for the Go-Ethereum client.
- There is a considerable difference between the fee obtainable per CPU time unit for different opcodes and this difference can be more than an order of magnitude.
- PyEthApp clients are able to gain higher fees for opcodes in the Arithmetic category compared to Go-Ethereum, and Go-Ethereum performs better across opcodes in the Environment category.

Category	Opcode	PyEthApp			Go-Ethereum			Parity			Used Gas
		Windows	Linux	MAC	Windows	Linux	MAC	Windows	Linux	MAC	
Arithmetic Operations	ADD	0.787	0.509	2.061	0.168	0.296	0.347	0.61	0.665	0.659	3
	ADDMOD	1.179	0.680	1.704	0.435	0.632	0.762	1.032	1.2	1.504	8
	DIV256	0.912	0.555	1.405	0.375	0.487	0.581	2.283	2.331	2.792	5
	EXP64	6.558	5.055	9.473	16.664	20.407	11.902	1.347	1.453	1.746	19.961
	EXP128	26.010	11.070	20.803	17.447	21.675	24.410	1.947	2.054	2.547	50
	EXP265	118.880	41.033	59.171	20.834	25.217	29.909	5	5.099	6.783	170
	MOD	0.921	0.563	1.490	0.304	0.447	0.578	0.725	0.799	0.882	5
	MUL	0.998	0.495	1.320	0.226	0.373	0.436	0.627	1.24	0.651	5
	MULMOD	1.856	1.064	3.049	0.644	0.834	0.988	1.075	0.809	1.811	8
	SDIV	1.734	1.095	4	0.521	0.651	0.787	0.774	0.763	0.885	5
	SIGNEXTEND	0.923	0.588	1.463	0.195	0.329	0.384	0.699	0.763	0.788	5
	SMOD	1.535	1.026	3.217	0.542	0.708	0.996	0.509	0.5	0.512	5
	SUB	0.810	0.444	1.467	0.170	0.298	0.355	0.606	0.658	0.646	3
Comparison & Bitwise Logic Operations	AND	0.807	0.476	1.369	0.150	0.276	0.328	0.703	0.777	0.796	3
	BYTE	0.919	0.579	1.363	0.146	0.274	0.319	0.715	0.777	0.809	3
	EQ	0.765	0.427	1.103	0.148	0.275	0.328	0.604	0.647	0.649	3
	GT	0.720	0.389	1.673	0.148	0.274	0.326	0.615	0.663	0.659	3
	ISZERO	0.589	0.360	0.899	0.074	0.135	0.160	0.665	0.706	0.72	3
	LT	0.678	0.382	1.063	0.148	0.274	0.328	0.612	0.662	0.655	3
	OR	0.863	0.489	1.286	0.152	0.276	0.326	0.701	0.77	0.788	3
	SGT	1.042	0.658	2.267	0.159	0.292	0.340	0.719	0.785	0.808	3
	SLT	1.043	0.630	1.610	0.160	0.288	0.342	0.72	0.79	0.81	3
	XOR	0.857	0.527	1.853	0.152	0.286	0.331	0.58	0.6	0.601	3
SHA3 Operations	SHA3-1	12.260	18.307	37.026	1.305	1.584	1.876	1.183	1.262	1.379	36
	SHA3-2	15.623	21.712	65.446	2.085	2.417	2.884	1.163	1.266	1.44	42
	SHA3-3	23.273	28.827	67.432	3.539	4.106	4.950	1.168	1.267	1.46	54
	SHA3-4	40.199	41.915	102.093	6.470	7.478	8.963	1.585	1.699	1.95	78
Environmental Information	ADDRESS	3.491	2.767	5.833	0.097	0.134	0.169	0.608	0.647	0.661	2
	BALANCE	7.234	5.268	10.931	0.813	0.971	1.132	1.175	1.264	1.455	20
	CALLDATACOPY1	1.793	1.337	3.225	0.287	0.314	0.376	0.766	0.807	0.964	3
	CALLDATACOPY2	42.565	33.863	56.395	0.290	0.314	0.376	0.742	0.811	0.851	72
	CALLER	4.927	3.635	8.320	0.071	0.109	0.137	0.614	0.646	0.66	2
	CALLVALUE	0.429	0.283	0.646	0.034	0.041	0.047	0.604	0.642	0.649	2
	CODECOPY1	13.029	10.567	17.838	0.358	0.420	0.509	0.738	0.804	0.926	9
	CODECOPY4	23.653	19.391	32.267	0.359	0.421	0.507	0.738	0.804	0.882	15
	EXTCODECOPY1	10.027	7.368	19.287	0.640	0.741	0.887	0.85	0.952	1.097	20
	EXTCODECOPY4	22.430	17.630	29.730	0.639	0.743	0.893	1.89	2.31	2.44	26
	EXTCODECOPY8	33.930	27.429	45.075	0.639	0.742	0.892	2.84	3.28	3.66	32
	EXTCODESIZE	8.128	6.203	11.984	0.996	1.319	1.503	0.637	0.658	0.689	20
	GASPRICE	0.593	0.410	0.859	0.033	0.040	0.046	0.599	0.64	0.648	2
	ORIGIN	4.874	3.627	8.116	0.069	0.110	0.136	0.612	0.648	0.661	2
Block information	BLOCKHASH	6.895	12.297	27.462	0.110	0.126	0.149	0.659	0.737	0.725	20
	COINBASE	3.633	2.778	7.876	0.072	0.107	0.133	0.616	0.649	0.66	2
	DIFFICULTY	0.430	0.267	0.799	0.045	0.051	0.061	0.597	0.64	0.65	2
	GASLIMIT	0.439	0.285	0.654	0.072	0.081	0.096	0.598	0.641	0.649	2
	NUMBER	0.404	0.245	0.597	0.045	0.051	0.063	0.6	0.643	0.65	2
	TIMESTAMP	0.387	0.238	0.540	0.045	0.051	0.061	0.545	0.55	0.567	2
Stack, Memory, Storage and Flow Operations	GAS	0.552	0.377	0.812	0.059	0.069	0.081	0.599	0.644	0.65	2
	MLOAD	8.819	6.952	15.716	0.365	0.589	1.036	0.666	0.73	0.735	3
	MSIZE	0.543	0.366	0.774	0.059	0.070	0.082	0.601	0.639	0.649	2
	MSTORE	3.703	2.834	7.861	0.344	0.518	0.853	0.684	0.704	0.727	3
	MSTORE8	0.804	0.594	1.583	0.250	0.335	0.772	0.668	0.721	0.733	3
	PC	0.519	0.347	1.132	0.041	0.045	0.061	0.6	0.641	0.65	2
	POP	0.346	0.224	0.607	0.081	0.123	0.135	0.605	0.655	0.644	2
	SLOAD	2.374	1.994	4.002	0.616	0.723	1.171	0.701	0.771	0.789	50
	SSTORE1	8.915	8.809	15.443	3.687	5.682	7.745	0.824	0.885	1.048	5000
	SSTORE2	10.394	9.496	16.926	7.169	12.518	14.625	0.834	0.9	1.201	20000
Push, Dup, Swap Operations	SWAP1	0.341	0.312	0.605	0.022	0.025	0.024	0.55	0.568	0.572	3
	DUP1	0.340	0.241	0.524	0.078	0.094	0.093	0.594	0.641	0.634	3
	PUSH1	0.376	0.259	0.594	0.098	0.117	0.116	0.64	0.66	0.69	3

Table 5.2: The average CPU time for each of the opcodes for all platforms in μs. The right-most column provides the Used Gas.

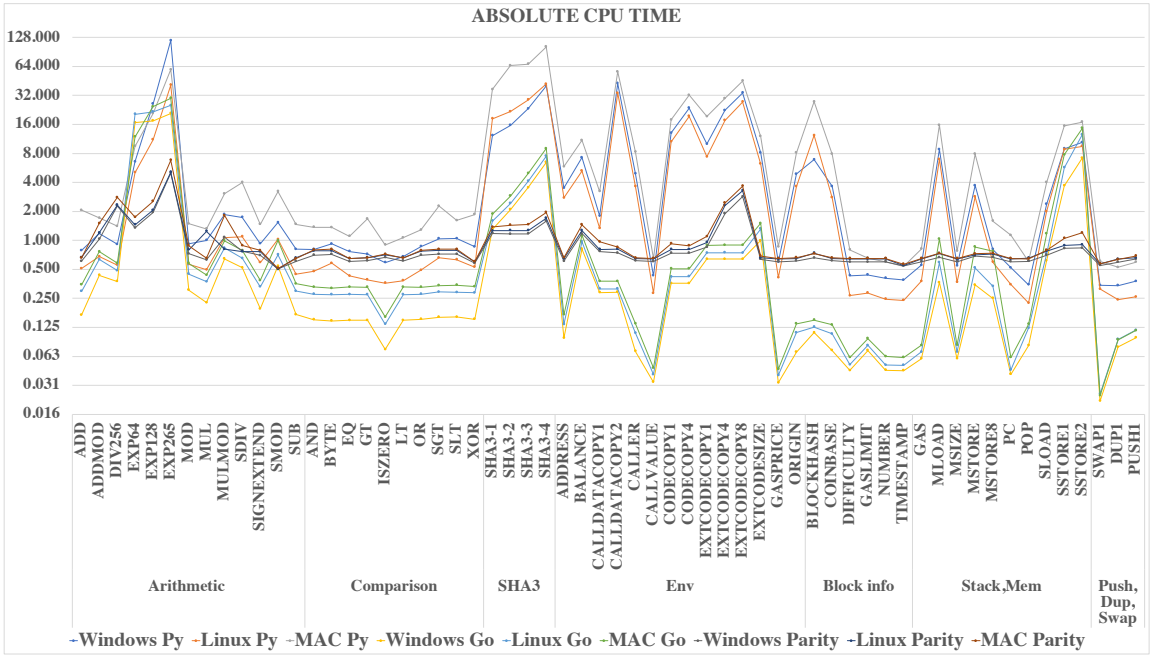

Figure 5.1: CPU time (in μs) for each opcode on a logarithmic scale.

5.2.1 Absolute CPU time

Figure 5.1 shows the graph with the CPU time required for each of the opcodes on all the nine platforms, as given in Table 5.2. The x-axis shows the same opcodes as in the table, and note that the CPU time on the y-axes is depicted on a logarithmic scale. Many of the opcodes take in the order of 1 μs or less to execute, but some take considerably longer, in the order of 0.1 milliseconds. There are several examples of opcodes that take more than 0.1 μs in Figure 5.1, e.g., the exponential (`EXP`) and the hash (`SHA3`).

From Figure 5.1 it is clear that the Go-Ethereum client outperforms both the Parity and PyEthApp clients on all three machines (see the green, blue and yellow lines): Mac 2.8Ghz, Linux 3.2GHz and Windows 3.6GHz. This gap is particularly clear in opcodes belonging to Comparison, Environment, and Block information categories. For example, the three machines consume between 4 and 8 μs to execute `ORIGIN` on the PyEthApp client, and between 0.6 and 0.66 μs on the Parity client, while they consume less than 0.14 μs on the Go-Ethereum client. Similarly, the machines consume between 30 and 60 μs to execute `CALLDATACOPY2` on the PyEthApp client, and on Parity client consumes between 0.74 and 0.85 μs, while they consume less than a half executing on the Go-Ethereum client.

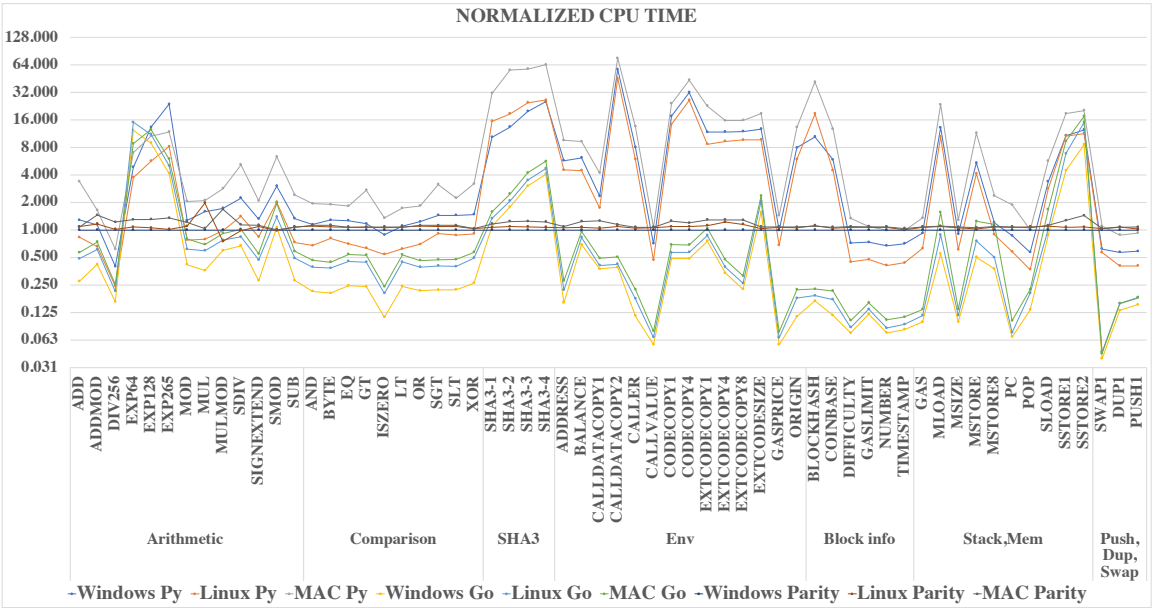

Figure 5.2: CPU time for each opcode, relative to the fastest platform (Windows Go-Ethereum 3.6GHz).

5.2.2 Relative CPU time

In order to better compare the CPU usage results for different platforms, Figure 5.2 shows the CPU time relative to the fastest platform, Windows Go-Ethereum 3.6Hz. In other words, we divided the results of each individual platform by the results of Windows Go-Ethereum, and, as a consequence, Windows Go-Ethereum shows as a straight line at 1 in Figure 5.2. Note that the Windows Go-Ethereum platform was not the fastest for some `EXP` opcodes, as is visible on the left side of the range in Figure 5.2.

The Go-Ethereum client generally outperforms the other clients: the three higher lines in Figure 5.2 correspond to the PyEthApp clients and the three lower ones to the Go-Ethereum clients, while the three middle ones to the Parity clients. The exceptions are `EXP`, the `SHA3` and `SSTORE` opcodes, where the Parity client performs better. The Mac 2.8 GHz machine is slower than other machines on both clients and this is expected since it has lower specifications. However, the Mac machine does outperform the Go-Ethereum client in some of the `EXP` opcodes (e.g., `EXP64` on the Go-Ethereum client). In addition, the Mac machine is as fast as the Linux machine in the Go-Ethereum client for `DUP`, `PUSH` and `SWAP` opcodes.

There is a very interesting difference between the PyEthApp and both Go-Ethereum and parity clients in terms of the OS on which they perform best. For PyEthApp,

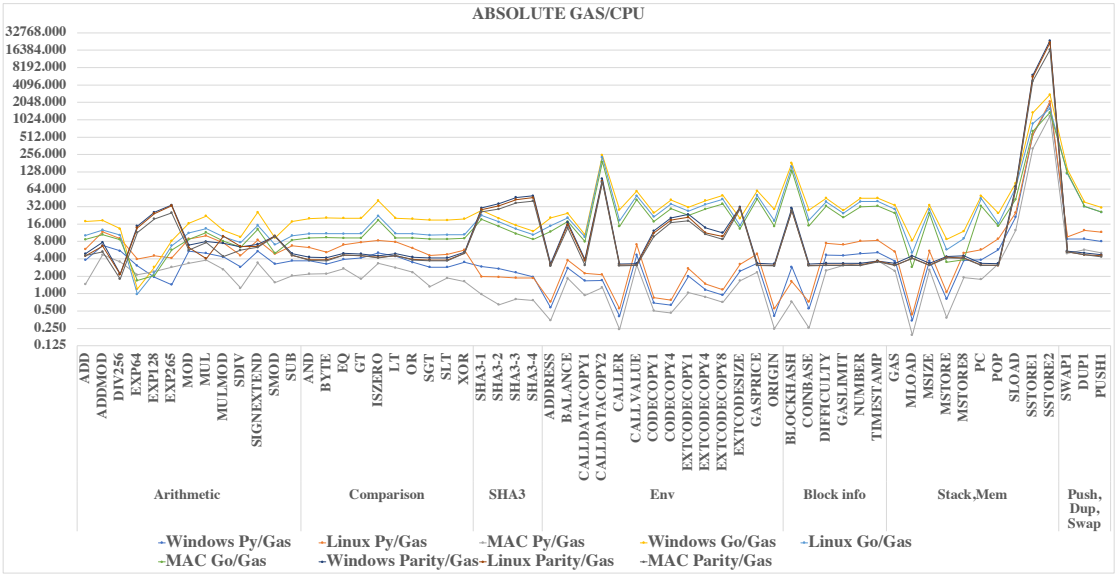

Figure 5.3: Used Gas (per [94]) per CPU time unit (in Gas/μs). Reward and cost are proportional for a platform if the lines are straight.

the Linux 3.2 GHz machine performs better than the Windows 3.6 GHz machine, whereas, for the other clients, the Windows 3.6 GHz machine outperforms the Linux 3.2 GHz. The only exception is for the `SHA3` opcodes, where the Windows machine is better on both clients.

5.2.3 Absolute gas/CPU

The critical issue for the successful operation of Ethereum is not the CPU time required for an opcode, but the ratio between the fee obtained and the CPU invested. This is displayed in Figure 5.3, which shows the amount of Used Gas per CPU μs for all opcodes. Recall that the Used Gas is given in the rightmost column of Table 5.2 and is calculated based on [94]. The higher the Used Gas, the more profit a miner makes.

Ideally, the curves in Figure 5.3 for each platform would be straight lines to have the fee and cost proportional to each other across all opcodes. Unfortunately, there is a considerable deviation between the best and worst return for various platforms, in the extremes more than two orders of magnitude.

Comparing the nine platforms, the conclusions from the previous two graphs remain valid, since all nine curves are scaled in the same manner, so Go-Ethereum outperforms the other clients. The range of values for the respective clients are significantly different. For the Go-Ethereum client, the average collected gas per μs of

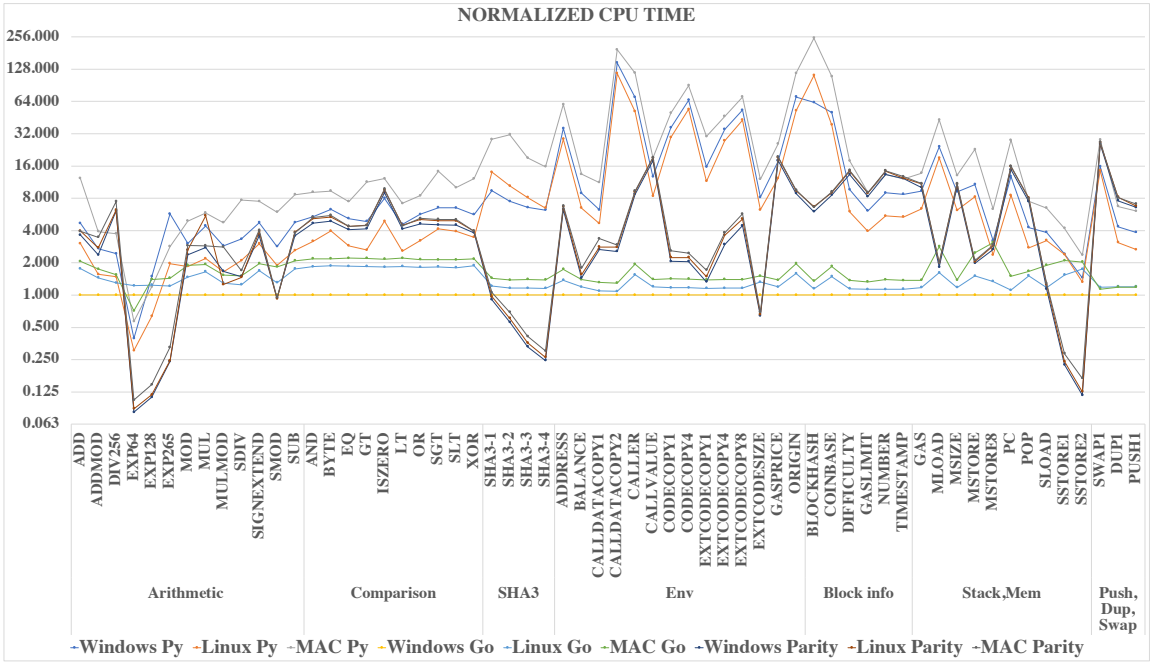

Figure 5.4: Normalized Used Gas per CPU time unit (results in 5.7 divided by the platform's result for opcode `Byte`). Reward and cost are proportional if the lines are straight at value 1.

CPU usage varies between 54 and 98 units of gas, while it ranges from 26 and 49 in the PyEthApp client and in the Parity client it range from 367 to 513. Therefore, the average amount of the awarded gas in the Parity client is about six time that of the Go-Ethereum client and a bout fourteen time that of the PyEthApp client.

In all clients, `SSTORE` opcodes are the most profitable ones since the awarded gas is higher than the required computation time. `SSTORE` is the only opcode available to modify storage. Therefore, the cost to the miner is in terms of the storage access, and it is priced based on storage access, not CPU use. For this performance benchmark study, it should be considered an outlier. Two other opcodes return a high fee per CPU time unit, namely `CALLDATACOPY2` and `BLOCKHASH`. In the PyEthApp client and the Parity client, opcodes such as `CALLER, ORIGIN` and `MLOAD` return the least value per CPU time unit, with less than one unit of gas return per CPU μs. In the Go-Ethereum client, opcodes such as `EXP64` and `EXP128` are the most expensive ones.

5.2.4 Normalized gas/CPU

To remove the platform-specific element from the results, we introduce normalized results for the gas used per μs in Figure 5.4. In this figure, the ideal behaviour of any of the six platforms would imply the result is a straight line with value 1. To

obtain this graph, we selected the 'median' opcode, namely `BYTE` (second left in the Comparison & Bitwise Logic Operations). For each platform, we took the Gas/CPU value of `BYTE` and divided all other opcode results for that platform by this value.

None of lines stays close to the value 1, which means that in all client combinations, there is considerable different between the opcodes in term of the fee rewarded per CPU μs. In fact, for all clients, the curves for the three machines follow a very similar pattern across the opcodes. Particularly the opcodes in the Arithmetic category provide a higher fee with the Parity client than with the other clients. At the same time, with respect to the Environment category, Go-Ethereum receives the higher fee. This implies that miners who run Parity client get better profit than the other clients when executing smart contracts that have opcodes in the Arithmetic category, while miners that use Go-Ethereum client perform better than other clients miners if the smart contract has more opcodes in the Environment category.

5.3 Second Experimental Results

This section presents and discusses the results of the second experiments with the implementations of OpBench run on identical hardware. We do this for three clients: Go-Ethereum, PyEthApp and Parity. We conduct our experiments using machines shown in Table 5.3. Therefore, our results in this section concern six different platforms denoted as Windows Go, Windows PyEthApp, Windows Parity, Linux Go, Linux PyEthApp and Linux Parity.

Machine	Desktop 3.5GHz	Desktop 3.5GHz
CPU	Intel i7 3.50GHz	Intel i7 3.50GHz
Cores	6	6
Memory	32GB	32GB
OS	Ubuntu 18.04.3 LTS	Windows 10

Table 5.3: Experimental 2 platforms.

The performance benchmark results for all opcodes are provided in Table 5.4, in microseconds (μs). We note that all confidence intervals are exceedingly tight, and are not provided in the table. As explained in Section 4.5, for PyEthApp, we executed all opcodes 100k times, collecting the average time and calculating a confidence interval. In Go-Ethereum, each opcode is executed until the desired benchmark time is reached, typically resulting in yet more samples than in PyEthApp, and again, a very tight confidence interval. In Parity, the actual opcode is executed 320,000,000 times, with

similar to the other clients, a tiny confidence interval (95%-Confidence Interval $\approx$ 0.005 μs in average).

The left-most column in Table 5.4 provides the categories into which Ethereum opcodes are typically classified, as explained in Subsection 4.4.2. We successfully benchmarked the opcodes in six out of eleven categories. As stated in Subsection 4.4.2, the two categories that we could not benchmark are *Logging* and *System* opcodes. For convenience of presentation, we merged `PUSH`, `DUP`, and `SWAP` into a single category, thus resulting in seven categories. Note that, the results in Table 5.4 do not include all results of OpBench system, we remove some opcodes that their computation are affected by the input size parameters such as `ADD64` and `ADD128`. Full results are available in [4].

We will discuss the benchmark results in stages. First, we consider the CPU time itself, in absolute value and relative to the fastest platform, respectively. Then we discuss the ratio of the Used Gas and CPU usage in order to identify the reward for the invested CPU time. Finally, we compare Linux with Windows using the same hardware platform running three different clients with each other.

We summarize the main insights that follow from our discussion up front:

- The Parity client is generally considerably faster (and thus more profitable) than the other clients, regardless of the type of the machine.

- The performance varies across the clients and the OSs. For example, the Linux 3.2 GHz machine outperforms the Windows 3.2 GHz machine on the PyEthApp client, while the reverse is valid for the Go-Ethereum and Parity clients.

- There is a considerable difference between the fee obtainable per CPU time unit for different opcodes. This difference between the profit made per time unit from different opcodes can be more than an order of magnitude.

- PyEthApp clients are able to gain higher fees for opcodes in the Arithmetic category compared to Go-Ethereum and Go-Ethereum performs better across opcodes in the Environment category. Also, Parity clients perform better than the other clients in most of the categories.

Category	Opcode	PyEthApp		Go-Ethereum		Parity		Used Gas
		Windows	Linux	Windows	Linux	Windows	Linux	
Arithmetic Operations	ADD	0.79	0.51	0.60	0.62	0.61	0.67	3
	ADDMOD	1.18	0.68	1.12	1.33	1.03	1.20	8
	DIV	0.91	0.56	1.67	1.74	2.28	2.33	5
	EXP64	6.56	5.05	13.86	14.16	1.35	1.45	19.9607
	EXP128	26.01	11.07	26.47	26.97	1.95	2.05	50
	EXP265	118.88	41.03	56.34	57.56	5.00	5.10	170
	MOD	0.92	0.56	0.72	0.74	0.73	0.80	5
	MUL	1.00	0.50	1.18	1.09	0.63	1.24	5
	MULMOD	1.86	1.06	1.11	1.75	1.08	0.81	8
	SDIV	1.73	1.09	1.76	1.15	0.77	0.76	5
	SIGNEXTEND	0.92	0.59	1.22	1.29	0.70	0.76	5
	SMOD	1.53	1.03	0.50	0.52	0.51	0.50	5
	SUB	0.81	0.44	0.61	0.64	0.61	0.66	3
Comparison & Bitwise Logic Operations	AND	0.81	0.48	0.64	0.66	0.70	0.78	3
	BYTE	0.92	0.58	0.65	0.67	0.72	0.78	3
	EQ	0.77	0.43	0.57	0.59	0.60	0.65	3
	GT	0.72	0.39	0.58	0.58	0.62	0.66	3
	ISZERO	0.59	0.36	0.59	0.62	0.67	0.71	3
	LT	0.68	0.38	0.58	0.59	0.61	0.66	3
	OR	0.86	0.49	0.65	0.67	0.70	0.77	3
	SGT	1.04	0.66	0.65	0.68	0.72	0.79	3
	SLT	1.04	0.63	0.66	0.67	0.72	0.79	3
	XOR	0.86	0.53	0.55	0.56	0.58	0.60	3
SHA3 Operations	SHA3-1	12.26	18.31	2.08	2.08	1.18	1.26	36
	SHA3-2	15.62	21.71	2.34	2.19	1.16	1.27	41.99994
	SHA3-3	23.27	28.83	2.44	2.55	1.17	1.27	53.99982
	SHA3-4	40.20	41.91	3.08	3.30	1.59	1.70	77.99958
Environmental Information	ADDRESS	3.49	2.77	1.17	1.16	0.61	0.65	2
	BALANCE	7.23	5.27	1.28	1.28	1.18	1.26	20
	CALLDATACOPY1	1.79	1.34	2.18	2.19	0.77	0.81	3
	CALLDATACOPY2	42.57	33.86	2.48	2.59	0.74	0.81	72
	CALLER	4.93	3.64	1.14	1.12	0.61	0.65	2
	CALLVALUE	0.43	0.28	0.56	0.57	0.60	0.64	2
	CODECOPY1	13.03	10.57	1.95	1.94	0.74	0.80	9
	CODECOPY4	23.65	19.39	1.91	1.93	0.74	0.80	15
	EXTCODECOPY1	10.03	7.37	1.00	1.00	0.85	0.95	20
	EXTCODECOPY4	22.43	17.63	2.22	2.24	1.89	2.31	26
	EXTCODECOPY8	33.93	27.43	3.33	3.36	2.84	3.28	32
	EXTCODESIZE	8.13	6.20	0.97	0.95	0.64	0.66	20
	GASPRICE	0.59	0.41	0.56	0.57	0.60	0.64	2
	ORIGIN	4.87	3.63	1.14	1.11	0.61	0.65	2
Block Information	BLOCKHASH	6.89	12.30	0.62	0.64	0.66	0.74	20
	COINBASE	3.63	2.78	1.13	1.11	0.62	0.65	2
	DIFFICULTY	0.43	0.27	0.57	0.58	0.60	0.64	2
	GASLIMIT	0.44	0.28	0.58	0.58	0.60	0.64	2
	NUMBER	0.40	0.25	0.57	0.58	0.60	0.64	2
	TIMESTAMP	0.39	0.24	0.53	0.54	0.55	0.55	2
Stack, Memory, Storage and Flow Operations	GAS	0.55	0.38	0.56	0.57	0.60	0.64	2
	MLOAD	8.82	6.95	1.84	1.83	0.67	0.73	3
	MSIZE	0.54	0.37	0.56	0.57	0.60	0.64	2
	MSTORE	3.70	2.83	1.73	1.71	0.68	0.70	3.00027
	MSTORE8	0.80	0.59	2.14	2.21	0.67	0.72	3.00003
	PC	0.52	0.35	0.57	0.57	0.60	0.64	2
	POP	0.35	0.22	0.57	0.59	0.61	0.66	2
	SLOAD	2.37	1.99	0.69	0.69	0.70	0.77	50
	SSTORE1	8.91	8.81	0.52	0.52	0.82	0.89	5000
	SSTORE2	10.39	9.50	0.53	0.53	0.83	0.90	20000
Push, Dup and Swap Operations	SWAP1	0.34	0.31	0.53	0.53	0.55	0.57	3
	DUP1	0.34	0.24	0.56	0.58	0.59	0.64	3
	PUSH1	0.38	0.26	0.60	0.63	0.64	0.66	3

Table 5.4: The average CPU time for each of the opcodes for all platforms, by category. The right-most column provides the Used Gas with the opcode. All results in (μs).

5.3.1 Comparison of Platforms Absolute CPU Time

Figure 5.5 shows the graph with the CPU time required for each of the opcodes on all the six platforms, as given in Table 5.4. The x-axis shows the same opcodes as in the table, and note that the CPU time on the y-axes is depicted in a logarithmic scale. Many of the opcodes take in the order of one microsecond or less to execute,

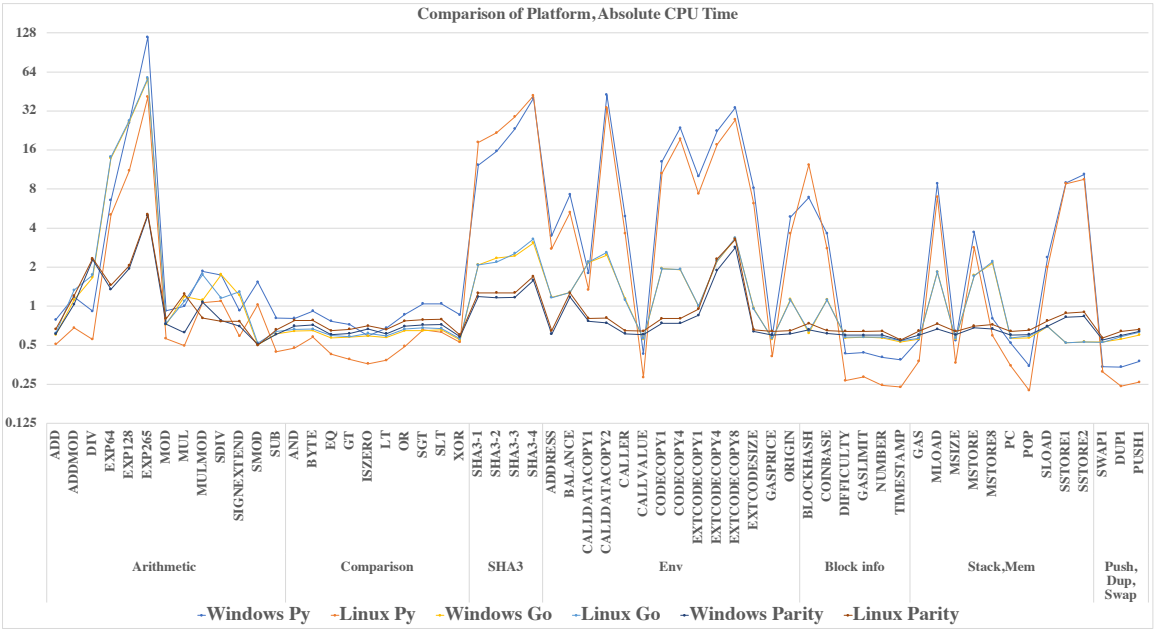

Figure 5.5: CPU time (in microseconds) for each opcode.

but some take considerably longer, in the order of 0.1 milliseconds. There are several examples in Figure 5.5, e.g., the exponential opcode (`EXP`) and the hash operation opcode (`SHA3`).

From Figure 5.5, it is clear that the Go-Ethereum client outperforms the PyEthApp client on all two machines: Linux 3.5GHz and Windows 3.5GHz. This gap is particularly clear in opcodes belonging to the SHA3, Environment and Block information categories. For example, the three machines consume between 40 and 102 μs to execute the `SHA3` on the PyEthApp client, while they consume less than 10 μs on the Go-Ethereum client. Similarly, the machines consume between 30 and 60 μs to execute the `CALLDATACOPY2` on the PyEthApp client, while they consume less than half a microsecond running on the Go-Ethereum client. The gap between these clients is expected since the performance of Golang language is faster than the Python language [66].

Furthermore, from the Figure, the Parity client outperforms the Go-Ethereum client on all opcodes but the Comparison & Bitwise Logic operations on the two machines. Also, as shown in the Figure, the Parity client outperforms the PyEthApp client in some categories such as Arithmetic, SHA3 and Environment on all the three machines. The most likely cause of the Parity outperforming the other clients is according to [67] the Rust language is faster than both the Python and the Golang languages.

There is a very interesting difference between the three clients in terms of the operating system on which they perform best. For the PyEthApp client the Linux 3.5GHz machine always performs better than the Windows 3.5GHz machine. The only exception is for `SHA3` and `BLOCKHASH` opcodes where the Windows 3.5GHz is faster. However, for the Go-Ethereum and the Parity clients, the Windows 3.5GHz machine performs better for all opcodes but the `SDIV` and the `SHA3-2` opcodes in the Go-Ethereum client and `MULMOD` opcode in the Parity client, see Table 5.5, which provides a selected slowest opcodes and compares in which clients and operating systems they are faster in, for more details.

Client	**Opcode**	**Machines**	
		Windows	Linux
PyEthApp	EXP64	6.56	5.05
	EXP128	26.01	11.07
	EXP256	118.88	41.03
	BLOCKHASH	6.89	12.3
Go-Ethereum	SDIV	2.34	2.19
	SHA3-1	1.76	1.15
Parity	MULMOD	1.08	0.81

Table 5.5: Comparison of CPU time (in microseconds) between different clients and operating system for selected slowest opcodes.

5.3.2 Sensitivity of Platform Speed to Relative CPU Time

In order to better compare the CPU usage results for different platforms, Figure 5.6 shows the CPU time relative to the fastest platform, the Windows Parity 3.5Hz. In other words, we divided the results of each platform by the results of the Windows Parity, and, as a consequence, the Windows Parity shows as a straight line at 1 in Figure 5.6. This allows us to identify opcodes that are particularly fast or slow on specific platforms. Note that the Windows Parity platform was not the fastest for some opcodes, as is visible on the left, the middle and the right sides of the range in Figure 5.6.

As mentioned, the Parity client generally outperforms the other clients: the two higher lines in Figure 5.6 correspond to the PyEthApp clients, the two middle lines to the Go-Ethereum clients and the two lower ones to the Parity clients, for most cases. The exceptions are listed in Table 5.6, where the PyEthApp client performs better than Go-Ethereum and Parity clients.

Table 5.6, shows a list of these opcodes where parity is not the fastest platform. Figure 5.6 and Table 5.6 reveal that the line for all platforms are not straight and therefore, on different platforms, different opcodes are faster. Thus, the results in the Table and the Figure help to decide which machine to use to execute a smart contract with these opcodes faster and to receive higher fees.

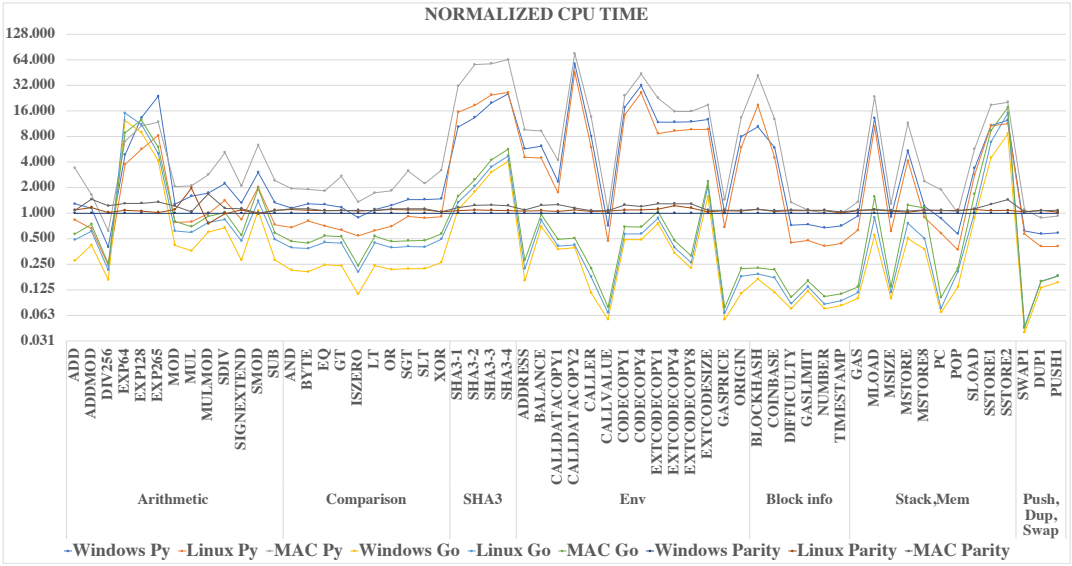

Figure 5.6: CPU time for each opcode, relative to the fastest platform (Windows Parity 3.5GHz).

5.3.3 Platforms Comparison for Gas/CPU Ratio

The critical issue for the successful operation of Ethereum is not the CPU time required for the opcode, but the ratio between the fee obtained and the CPU invested [81]. This is displayed in Figure 5.7, which shows the amount of Used Gas per CPU microsecond for all opcodes. Recall that the Used Gas is given in the rightmost column of Table 5.4 and is calculated by the EVM from values set in [94]. The higher the Used Gas, the more profit a miner makes.

Ideally, the curves in Figure 5.7 for each platform would be straight lines, since then fee and cost are proportional to each other across all opcodes. Unfortunately, there is a considerable deviation between the best and worst return for the various platform, in the extremes more than two orders of magnitude.

Comparing the six platforms, the conclusions from the previous two graphs remain valid, since all six curves are scaled in the same manner, so the Parity client outperforms the Go-Ethereum and the PyEthApp clients. The range of values for the

Category	Opcode	PyEthApp		Go-Ethereum		Parity	
		Windows	Linux	Windows	Linux	Windows	Linux
Arithmetic Operations	DIV	0.91	0.56	1.67	1.74	2.28	2.33
	MOD	0.92	0.56	0.72	0.74	0.73	0.8
	MUL	1	0.5	1.18	1.09	0.63	1.24
	MULMOD	1.86	1.06	1.11	1.75	1.08	0.81
	SIGNEXTEND	0.92	0.59	1.22	1.29	0.7	0.76
	SUB	0.81	0.44	0.61	0.64	0.61	0.66
Comparison & Bitwise Logic Operations	AND	0.81	0.48	0.64	0.66	0.7	0.78
	BYTE	0.92	0.58	0.65	0.67	0.72	0.78
	EQ	0.77	0.43	0.57	0.59	0.6	0.65
	GT	0.72	0.39	0.58	0.58	0.62	0.66
	ISZERO	0.59	0.36	0.59	0.62	0.67	0.71
	LT	0.68	0.38	0.58	0.59	0.61	0.66
	OR	0.86	0.49	0.65	0.67	0.7	0.77
	SGT	1.04	0.66	0.65	0.68	0.72	0.79
	SLT	1.04	0.63	0.66	0.67	0.72	0.79
	XOR	0.86	0.53	0.55	0.56	0.58	0.6
Environmental Information	CALLVALUE	0.43	0.28	0.56	0.57	0.6	0.64
	GASPRICE	0.59	0.41	0.56	0.57	0.6	0.64
Block Information	DIFFICULTY	0.43	0.27	0.57	0.58	0.6	0.64
	GASLIMIT	0.44	0.28	0.58	0.58	0.6	0.64
	NUMBER	0.4	0.25	0.57	0.58	0.6	0.64
	TIMESTAMP	0.39	0.24	0.53	0.54	0.55	0.55
Stack, Memory, Storage and Flow Operations	GAS	0.55	0.38	0.56	0.57	0.6	0.64
	MSIZE	0.54	0.37	0.56	0.57	0.6	0.64
	MSTORE8	0.8	0.59	2.14	2.21	0.67	0.72
	PC	0.52	0.35	0.57	0.57	0.6	0.64
	POP	0.35	0.22	0.57	0.59	0.61	0.66
	SSTORE1	8.91	8.81	0.52	0.52	0.82	0.89
	SSTORE2	10.39	9.5	0.53	0.53	0.83	0.9
Push, Dup and Swap Operations	SWAP1	0.34	0.31	0.53	0.53	0.55	0.57
	DUP1	0.34	0.24	0.56	0.58	0.59	0.64
	PUSH1	0.38	0.26	0.6	0.63	0.64	0.66

Table 5.6: Selected opcodes where other clients outperform the Windows Parity clients.

respective clients is significantly different. For the Parity client, the average collected gas per microsecond of CPU usage varies between 367 and 513 units of gas, while it ranges from 344 and 500 in the Go-Ethereum client and 26 to 47 in the PyEthApp client. In other words, the average amount of the awarded gas in the Parity and the Go-Ethereum clients is about fourteen times that of the PyEthApp client.

In all clients, `SSTORE` opcodes are the most profitable ones since the awarded gas is higher than the required computation time (see Subsection 4.4.2). The `SSTORE` opcode is the only opcode available to modify storage. Therefore, the cost to the miner is in terms of the storage access, and it is priced based on storage access, not CPU use. For this performance benchmark study, it should be considered an outlier. In the Go-Ethereum and the Parity clients, two other opcodes return a high fee per CPU time unit, namely `CALLDATACOPY2` and `BALANCE`. This is generally caused by the fact that the client must either load state objects from disk or a memory, which require high CPU usage and therefore, the assigned Gas costs are higher than the required CPU overheads. In the PyEthApp client, opcodes such as `CALLER, GAS` and `GASPRICE` return the least value per CPU time unit, with less than one unit of gas return per

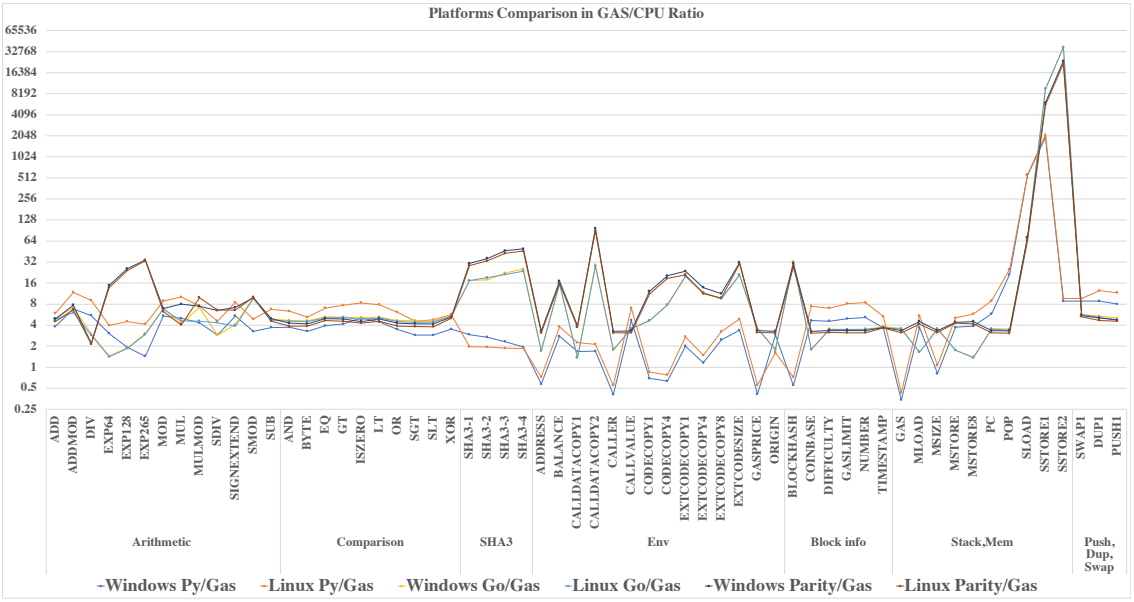

Figure 5.7: Used Gas (per [94]) per CPU time unit (in Gas/microsecond). Reward and cost are proportional for a platform if the curve is a straight line.

CPU microsecond. All these opcodes obtain their values from the transaction, which seems expensive in the PyEthApp client due the implementation process flow. In the Go-Ethereum client, opcodes such as `EXP64` and `EXP128` are the most expensive ones. However, the Used Gas in these opcodes are roughly proportional to CPU usage.

On the other hand, the same opcodes in the Parity client opcode return fifteen times higher Used Gas value per CPU time unit than the Go-Ethereum client, so, from miners' perspective, smart contacts that have these are preferable to be selected. We also note that in the Parity client, formula based opcodes return higher value per CPU than the computationally based ones. The main reason for this particular variation is caused by the fact that the Rust is faster than the others languages [67]. Table 5.8 provides the most profitable opcodes in each client.

To remove the platform-specific element from the results, we introduce normalized results for the gas user per microsecond in Figure 5.8. In this Figure, typical behavior of any of the six platforms would imply the result is a straight line with value 1. To obtain this graph, we selected the 'median' opcode, namely `BYTE` (second from the left in the Comparison & Bitwise Logic Operations). For each platform, we took the Gas/CPU value of the `BYTE` opcode and divided all other opcode results for that platform by the value for the `BYTE` opcode.

Particularly, some opcodes in the Arithmetic category provide a higher fee per CPU in microseconds with the PyEthApp client than with the Go-Ethereum client

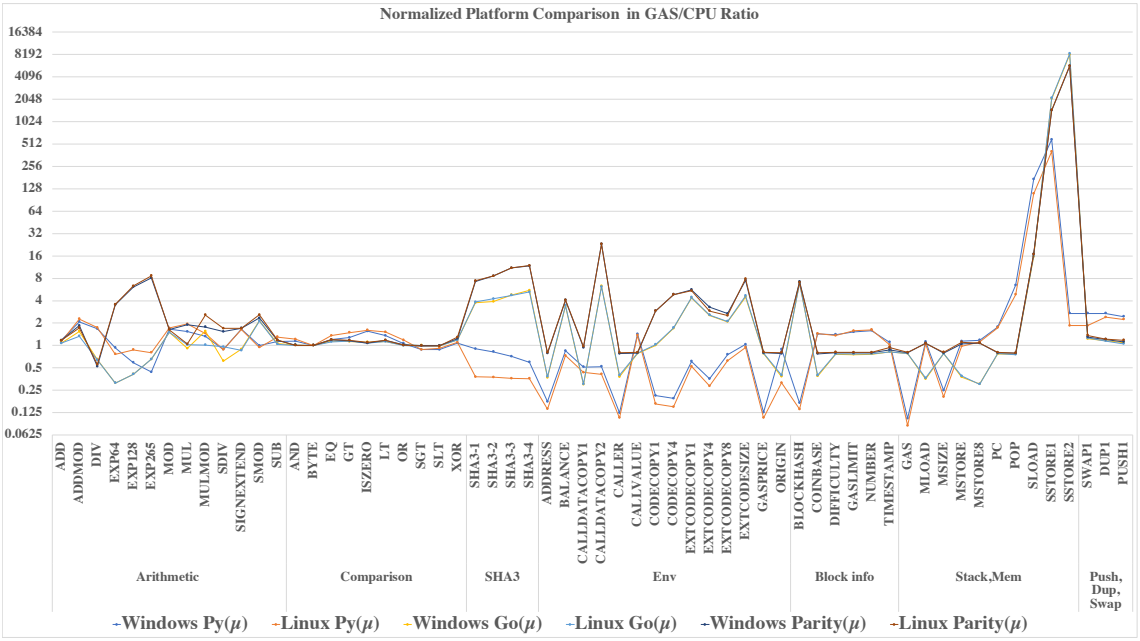

Figure 5.8: Normalized Used Gas per CPU time unit (results in 5.7 divided by the platform's result for opcode `BYTE`). Reward and cost are proportional if the curve is a straight line at value 1.

and less than the Parity client. At the same time, with respect to the Stack, Memory and Storage category, the Go-Ethereum client receives the higher fee compared to the PyEthApp and the Parity clients. Table 5.7 presents selected opcodes with highest Gas/CPU for each client. It is clear from the Table, the Parity client has the more highest opcodes than the other clients.

However, the Go-Ethereum provide higher fee for the `SSTORE` opcode than the other clients, whereas the PyEthApp outperforms the others in the `PC` and the `SLOAD` opcodes. This implies that miners who run the Parity or the PyEthApp clients get better profit than the Go-Ethereum client when executing smart contracts that implement opcodes belong to the Arithmetic category, while miners who use the Go-Ethereum client perform better than the Parity and the PyEthApp clients miners if the smart contract has more opcodes belong to the Block information category. Moreover, miners who run the Parity client and execute smart contracts that have more `SHA3` opcodes get a better profit than the other clients. Also, the Parity client is more profitable than the other clients in most of the Environment category.

Client	Opcode	Machines	
		Windows	Linux
Parity	EXP64	14.82	13.74
	EXP128	25.68	24.34
	EXP265	34.00	33.34
	SHA3-1	30.43	28.53
	SHA3-2	36.11	33.18
	SHA3-3	46.23	42.62
	SHA3-4	49.21	45.91
	CALLDATACOPY2	97.04	88.78
	CODECOPY4	20.33	18.66
	EXTCODECOPY1	23.53	21.01
	EXTCODECOPY4	13.76	11.26
	EXTCODECOPY8	11.27	9.76
	EXTCODESIZE	31.40	30.40
	BLOCKHASH	30.35	27.14
	SSTORE1	6067.96	5649.72
	SSTORE2	23980.82	22222.22
Go-Ethereum	SHA3-1	17.29	17.31
	SHA3-2	17.92	19.14
	SHA3-3	22.09	21.20
	SHA3-4	25.34	23.66
	CALLDATACOPY2	29.04	27.76
	EXTCODECOPY1	20.06	19.94
	EXTCODESIZE	20.73	21.01
	BLOCKHASH	7	6.88
	SSTORE1	9578.54	9578.54
	SSTORE2	37664.78	37878.79
PyEthApp	PC	5.77	8.93
	SLOAD	560.85	567.62
	SSTORE1	1924.15	2106.14
	SSTORE2	8.80	9.61

Table 5.7: Selected opcodes by highest Gas per CPU on the three clients.

5.3.4 Comparison of Clients Absolute CPU Time

In this Section, we remove the machine specifications, and we compare the CPU overhead times for the three clients. Figures 5.9, 5.10 and 5.11 depict the CPU overhead times for each of the three clients (i.e., PyEthApp, Go-Ethereum and Parity), respectively and Figure 5.5 combines them all in one figure for convenience. According to the Figures, it is clear that the Parity client is faster than the other client in almost all categories but the Comparison categories where the PyEthApp client is the fastest. In this category, i.e., the Comparison & Bitwise Logic Operations, the PyEthApp client outperforms all clients. The reason for that is based on [85], which compare Python with other programming languages at some regular expressions and bitwise opcodes, Python is performing better than the other languages. Moreover, the PyEthApp client only outperforms the Go-Ethereum client in the `EXP` opcode in the Arithmetic category, whereas, the Parity client outperforms the other two clients in this opcode.

We also observe that the formula-base opcodes, presented in Table 5.8, are executed faster on the Parity client. This is due to the high-performance of Rust over the

others. However, opcodes that manipulate the stack such as PUSHs and POPs consume higher CPU overhead times in the Parity client compared to the PyEthApp and the Go-Ethereum clients.

In summary, the Parity client compared to the other clients, is faster for almost all opcodes, followed by the Go-Ethereum client then the PyEthApp client, which is the slowest. Thus, based on the results, the best client to execute most of the EVM's opcodes faster is the Parity client.

Opcode	Windows Py	Linux Py	Windows Go	Linux Go	Windows Parity	Linux Parity	Used Gas
EXP64	6.56	5.05	13.86	14.16	1.35	1.45	19.96
EXP128	26.01	11.07	26.47	26.97	1.95	2.05	50
EXP265	118.88	41.03	56.34	57.56	5.00	5.10	170
SHA3-1	12.26	18.31	2.08	2.08	1.18	1.26	36
SHA3-2	15.62	21.71	2.34	2.19	1.16	1.27	41.99
SHA3-3	23.27	28.83	2.44	2.55	1.17	1.27	53.99
SHA3-4	40.20	41.91	3.08	3.30	1.59	1.70	77.99
CALLDATACOPY1	1.79	1.33	2.18	2.192	0.766	0.807	3
CALLDATACOPY2	42.56	33.86	2.479	2.594	0.742	0.811	72
CODECOPY1	13.03	10.57	1.95	1.94	0.74	0.80	9
CODECOPY4	23.65	19.39	1.91	1.93	0.74	0.80	15
EXTCODECOPY1	10.03	7.37	1.00	1.00	0.85	0.95	20
EXTCODECOPY4	22.43	17.63	2.22	2.24	1.89	2.31	26
EXTCODECOPY8	33.93	27.43	3.33	3.36	2.84	3.28	32
SSTORE1	8.91	8.81	0.52	0.52	0.82	0.89	5000
SSTORE2	10.39	9.50	0.53	0.53	0.83	0.90	20000

Table 5.8: Comparison between the PyEthApp, the Go-Ethereum, and the Parity clients on formula-based opcodes.

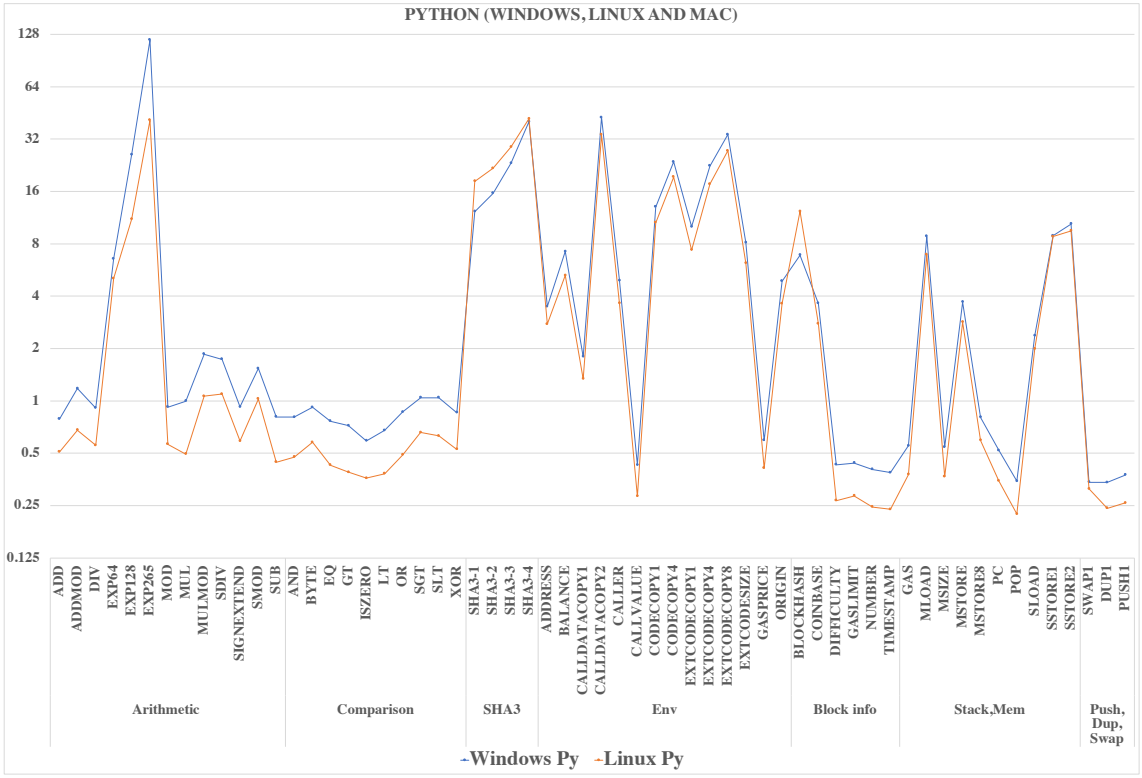

Figure 5.9: CPU time (in microseconds) for each opcode in the PyEthApp client.

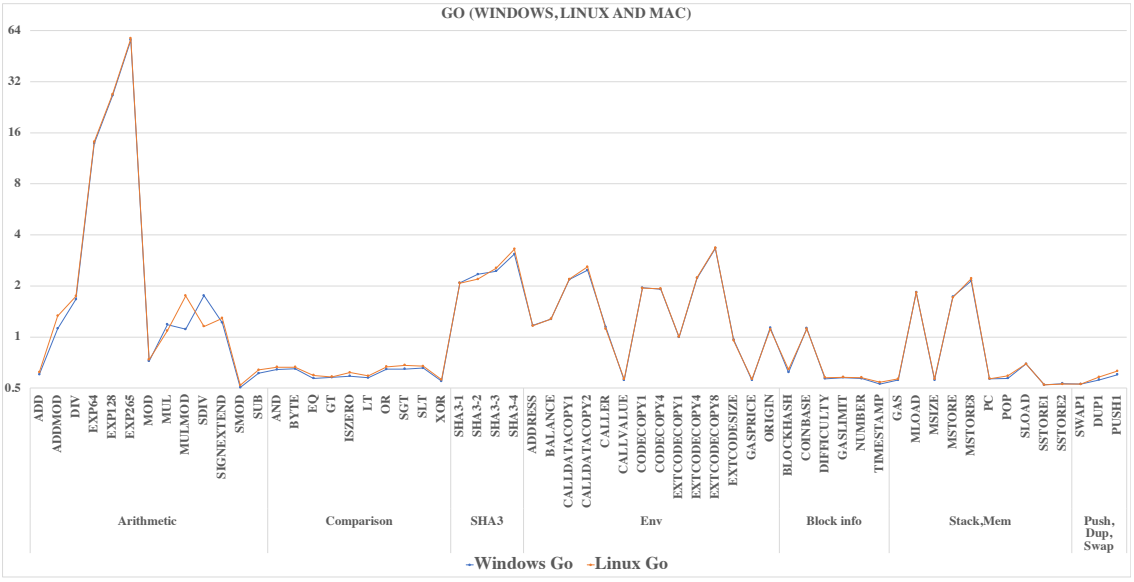

Figure 5.10: CPU time (in microseconds) for each opcode in the Go-Ethereum client.

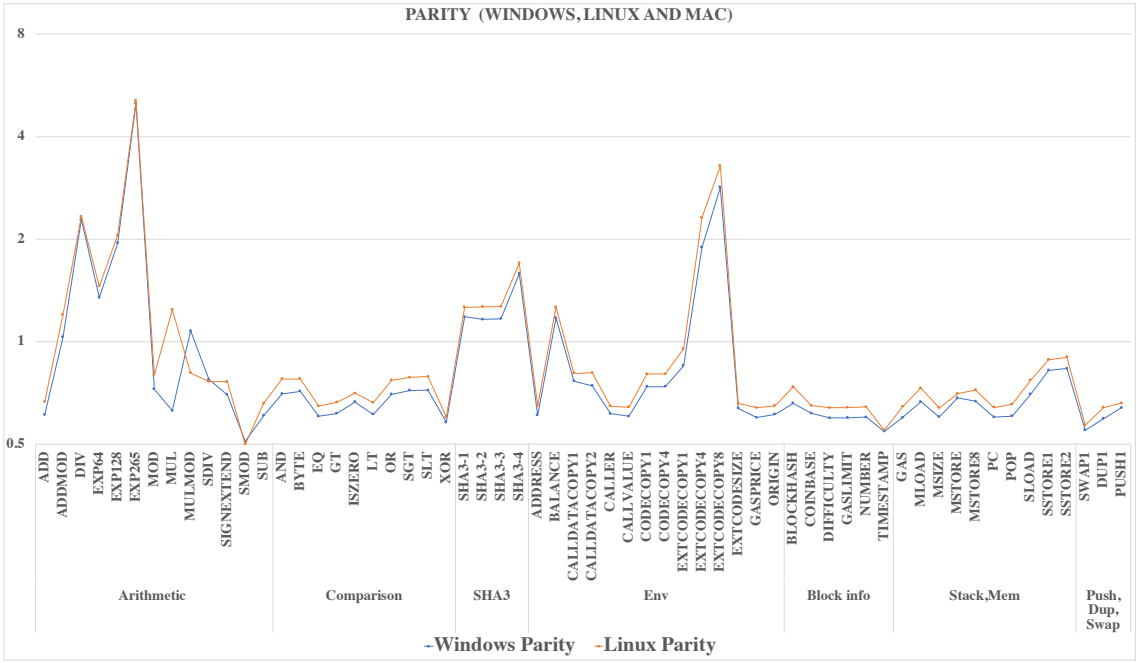

Figure 5.11: CPU time (in microseconds) for each opcode in the Parity client.

5.3.5 Comparison of Different Operating Systems

In order to better compare different operating systems (OSs), we should collect same experiment's results from the three clients on same hardware platforms running over two different OSs, in this case Windows and Linux. For both Windows and Linux, in Figures 5.12 and 5.13, we give the absolute CPU time for the three clients PyEthApp, Go-Ethereum and Parity. In Figure 5.14, the two figures are combined for convenience. An identical experiment setup is conducted with the identical inputs for the opcodes for two OSs.

Interestingly in Figure 5.14, in the Parity and the Go-Ethereum clients, the Windows machines are on average 8.20% and 31.12% faster than the Linux machine, respectively, whereas, the Linux machine is about 33.56% faster in the PyEthApp client on average for all opcode categories.

Additionally, from Figure 5.14, it is clear that in all six combinations the curve of all machines follow a very similar pattern across all opcodes. The main difference is the CPU overhead variations. Furthermore, Linux machine running the PyEthApp client is faster than the Linux Parity, the Windows Parity and the Windows PyEthApp in all Comparison & Bitwise Logic opcodes and some Arithmetic opcodes as well as some Block information opcodes (see the blue line in Figure 5.13).

To sum up, from the miners' perspective, the ideal combination to collect more gas units and thus, more profits against relatively cheap CPU consumption's would be Windows running the Parity client. This would include all smart contracts that implement all opcodes but less `DIV`, `CALLVALUE` and `SSTORE` opcodes. Additionally, form users' perspectives, the ideal platform to execute their transactions and therefore, execute their contracts, if they have the choice, is the Parity on either OSs.

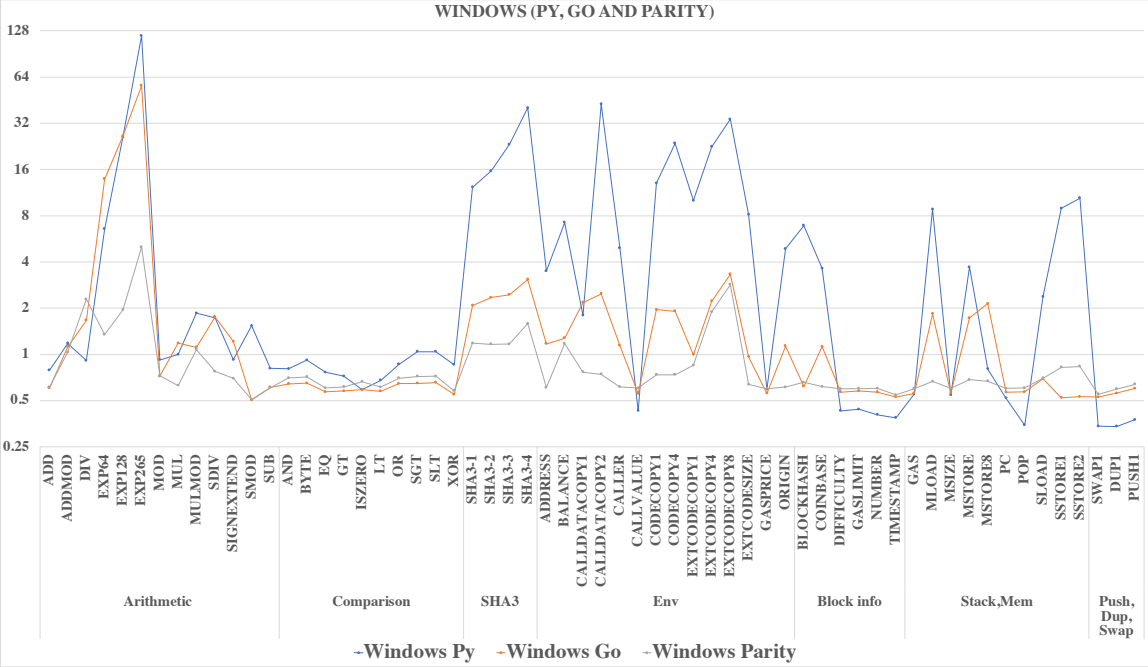

Figure 5.12: Absolute CPU time for Windows machine for all clients.

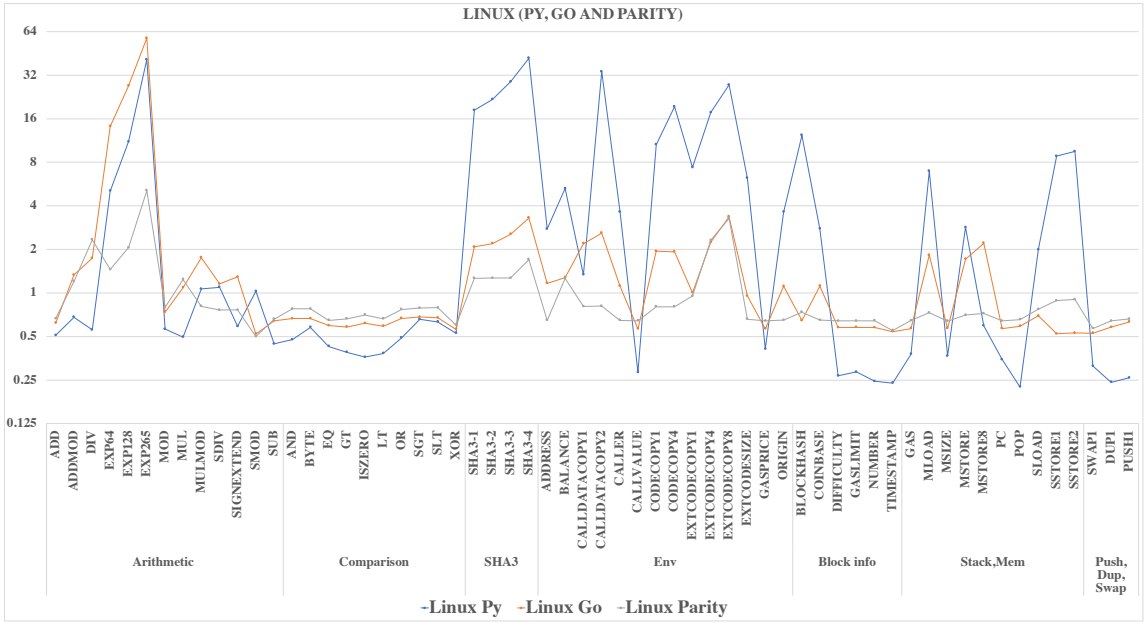

Figure 5.13: Absolute CPU time for Linux machine for all clients.

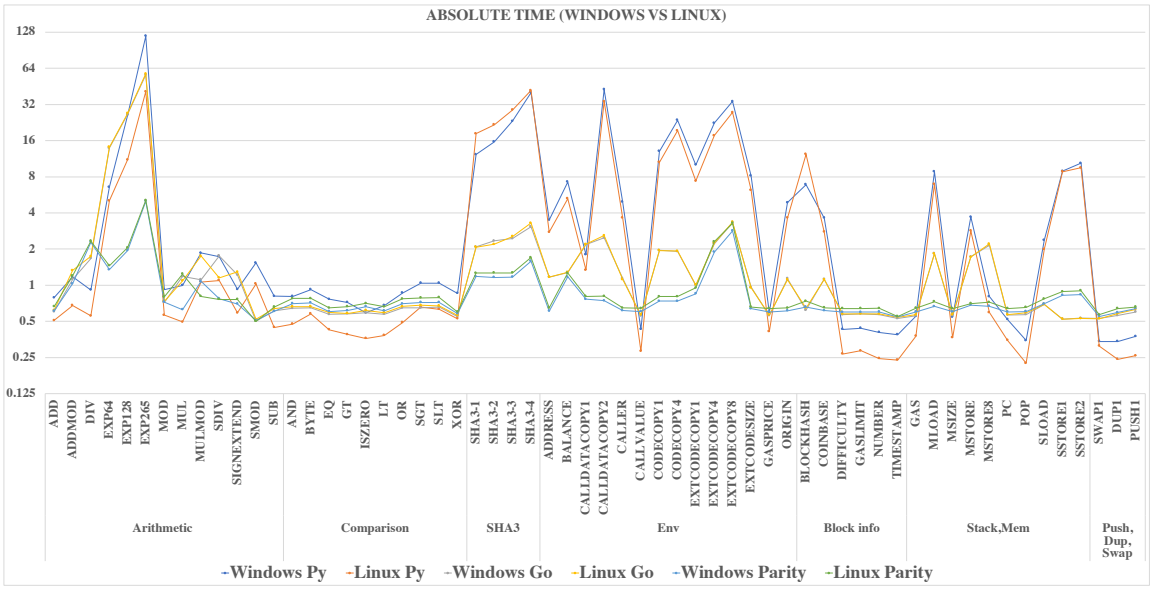

Figure 5.14: Absolute CPU time for Windows and Linux machines for all clients.

5.4 Result Validation

In this section, we validate the results collected by OpBench system and check their correctness. The validation is divided in the following subsections. The first subsection evaluates the results of any opcode that has been generated repeatedly and composed them in a smart contract function that can be triggered utilizing the Ethereum

transaction. The second subsection presents an evaluation that shows that subtracting of timing and the `POP` opcode do not impact the overall execution time.

5.4.1 Evaluation of Individual Opcodes and Composed Complete Contract

OpBench considers each instruction independently, and executes it multiple times (interleaved if necessary, with `POP` opcode to keep the stack at a target size). It remains unclear if this is representative of the individual cost of these operations when *composing a complete contract*. The repeated execution of the same instruction may benefit the use of just-in-time compilation available on Ethereum. In this section, we show the correlation between individual opcodes performance and the composed performance of opcodes in complete contracts.

To accomplish this, we select a set of opcodes to carry out our evaluation. The selected opcodes belong to the Arithmetic operations category, for simplicity of the implementation. We implement a smart contract that has four different functions, each function containing an opcode. Listing 5.1 shows the smart contract code that contains these functions. The smart contact is implemented and compiled using Solidity [48] and Solidity compiler [47], respectively. The experiment was conducted on a MacBook Pro with a 2.8GHz Intel i5 CPU and 8GM RAM, on the PyEthApp client. Firstly, we compile the smart contact and then deploy the compiled bytecode into the Ethereum blockchain. Secondly, we invoke each function by sending an Ethereum transaction (similar to the previous implementation in Section 4.5). Finally, we calculate the average of the execution time of each function.

As depicted in Listing 5.1, each function, once triggered, repeats the expression 100k times (Line 7, 13, 18 and 23). 100k times was selected because it provides a tight confidence interval (%95-Confidence Interval $\approx 0.004\ \mu s$). The EVM keep a record of each opcode it executes and returns the overall execution time, a list of each opcode execution time and Used Gas. Table 5.9, shows the result of the experiment for each opcode in microseconds (μs) and for each function in seconds. The value for each opcode is the average of the values obtained in each run. According to the results, OpBench's results presented earlier is believed to be accurate.

Opcode	Function overall time	Opcode avg CPU time	Used Gas
ADD	3.80	2.07	3
SUB	4.00	1.48	3
MUL	4.43	1.33	5
DIV	4.95	1.47	5

Table 5.9: The CPU time for functions execution in (second) and each selected opcode in (microsecond).

```
1
2  pragma solidity ^0.5.0;
3
4  contract Evaluation {
5
6      function sum () public{
7          for (uint i=0;  i<100000; i++ ){
8              uint s = 1000+1000;
9          }
10     }
11     function sub () public{
12         for (uint i=0;  i<100000; i++ ){
13             uint s = 1000-1000;
14         }
15     }
16     function mul () public{
17         for (uint i=0;  i<100000; i++ ){
18            uint s = 1000*1000;
19         }
20     }
21     function div () public{
22         for (uint i=0;  i<100000; i++ ){
23            uint s = 1000/100;
24         }
25     }
26 }
```

Listing 5.1: The smart contact implementation of the selected opcodes.

5.4.2 Evaluating the Overhead Effects of the POP Opcode and the Timing

As discussed in Subsection 4.4.3, in OpBench system we identified a new approach to manage the stack limit size using POP opcode to be able to repeat an individual opcode unlimited times. However, the introduction of the POP opcode as well as setting the timer could have an impact on the overall CPU performance (execution time) of the actual opcode.

In this section, we show our evaluation of the impact on setting and including both timing and the POP opcode in the generated bytecode and, therefore the final execution results.

For setting timing, as mentioned in the implementation in Section 4.5, we utilized the build-in functions *Profiling* and *Benchmark* modules in both Python and Golang programming languages, respectively. These modules can be used as either as a callable (inside the code) or as a command line interface [59][21]. In OpBench, we used these modules inside the code and we estimated their overheads by running them

without the bytecodes (i.e., we test nothing but the startup and shutdown overhead). The execution overheads of these modules are recorded and subtracted from the final execution results of the actual bytecode.

For the validation purpose, we utilized the command line interface. In the command line interface, the starting and ending times are taken before and after the execution of the bytecode outside the real implementation. Therefore, the overheads of start-up and shutdown are managed by the operating system [21][59]. Next, these modules output a set of execution statistics that shows the CPU time for each line in the programming code. According to [59][21], the side effects of these modules is very minimal. Fortunately, the results of both interfaces are identical (i.e., the callable and command line). Hence, OpBench's results are not affected by introducing the timing.

In order to estimate the side effects of injecting the `POP` opcode on the overall execution time, we assess a set of selected opcodes without the `POP` opcode approach. In this way, we only managed to repeat each opcode 1024 times. Fortuitously, the results of these opcodes are comparable to OpBench system's results. The side-effects evaluation of the timing and `POP` opcode are conducted on a MacBook Pro 2.8GHz Intel i5 CPU with 8GB RAM.

5.5 Conclusion

In this chapter, we presented two sets of experimental results one for six system configurations, running the two different implementations on three different computers (and operating systems) and other for six system configurations, running the three different implementations on two operating systems with *same* hardware platform. These experiments demonstrate the validity of OpBench approach across platforms. It also allows us to obtain results comparing PyEthApp with Go-Ethereum with Parity, all with respect to CPU usage for various opcodes, and for the fee rewarded per unit of CPU time. Our results show that the CPU time required for opcodes is not always proportional to the gas used and fee received. The difference can be an order of magnitude between the opcodes.

Our results also show that there can be an order of magnitude difference in terms of the reward per unit of CPU time for different opcodes. Our experiments also indicate that there is a considerable performance difference between clients, with the Parity client outperforming the PyEthApp and the Go-Ethereum clients. Moreover,

the results indicate the Windows operating system is faster than the Linux operating system for most of opcodes.

Future work, it will be of interest to expand the scope of the benchmark to include assessment of occupying resources in general, including storage, blocking the machine as well as actual energy consumption. To support such efforts, the code of OpBench system is available to others[1].

[1] `https://github.com/Amjad13`

Chapter 6

Implementation and Evaluation of Counter-Collusion Smart Contracts for Verifiable Cloud Computing

6.1 Summary

Cloud computing has become an irreversible trend. With this comes the pressing need for verifiability, to assure the correctness of computation outsourced to the cloud. Existing verifiable computation techniques all have a high overhead. Thus applications deployed in the cloud, would render cloud computing more expensive than its on-premises counterpart. To achieve verifiability at a reasonable cost, the approach in this chapter leverages game theory and proposes a smart contract based solution. In a nutshell, a client lets two clouds compute the same task, and uses smart contracts to stimulate tension, betrayal and distrust between the clouds so that rational clouds will not collude and cheat. In the absence of collusion, verification of correctness can be done easily by cross-checking the results from the two clouds. By resorting to game theory and smart contracts, we can avoid heavy cryptographic protocols. The client only needs to pay two clouds to compute in the clear, and a small transaction fee to use the smart contracts. The focus in this chapter is on the design of smart contracts and the implementation using Ethereum, as well as performance evaluation of the approach.

This chapter is structured as follow. Section 6.2 provides an introduction that introduces the problem statement and our contributions. In Section 6.3, we provide the adversary model and the assumptions. The monetary variables that used in the smart contracts are presented in Section 6.4. Contracts their explanations and their pseudo-codes are provided in Sections 6.5, 6.6, 6.7 and 6.8. Section 6.9 presents the

implementation of our solution. In Section 6.10, we present the cost and the overhead of our solution, and we conclude the chapter in Section 5.5.

6.2 Introduction

Cloud computing has gained considerable interest and becomes vital for businesses. According to [80] in 2016 around 95% of organizations adopting and/or running their applications with the cloud. In addition, the Synergy Research Group [84] reported that about $148 billion is the market cap for the worldwide cloud computing in 2016, having grown up by 25% annually. In [53], the authors predicted that the spending on IT and cloud computing specifically would reach $1 trillion by 2020.

Verifiability for cloud computing is a very crucial requirement for the organizations who have moved to earn and gain services with the cloud. It is difficult to fully trust the cloud services provider, which is a third party that provides the clouding services, to perform a crucial task correctly. As a result, clients should be able to verify the correctness of the result of their requested services, to gain greater confidence in the outsources computation.

Roughly, solutions techniques based on either cryptography or replication are the most existing techniques for verifying the outsourced computation. Usually, the **cryptography-based approaches** are where a single cloud is used to outsource a computation by the client, then the cloud returns and proves the results were correctly computed. Relying on the cryptography to ensure that with a high probability that the client will reject if the result was incorrect. The **replication-based approaches** are where the client outsources a task to multi-cloud who independently compute the same task. The returned results are collected and cross-checked by the client. Using a consensus protocol, the correctness of the result can be verified if the number fault servers below a threshold.

A Cost Analysis Since the biggest motivation for adopting cloud services is perhaps the cost saving, the verifying existing approaches are not quite economically. For example, we used the Amazon AWS Total Cost of Ownership Calculator [11] on a few typical settings, and found that by moving their on-premises IT infrastructures to AWS, companies could save 50% to 69% of the cost. The cost of moving to the cloud is less, so the saving is significant, however, is not large enough to support existing verifying approaches. Cloud computing uses the pay-per-use paradigm, and so, the users are charged for resources usage. Adopting the cryptography-based approach is much more expensive since the overheads imposed by executing the cryptographic

algorithms/protocols. The typical overhead is $10^3 - 10^9$ times higher than computing the task itself [91] and would translate to a prohibitively high financial cost to the client. The replication-based approach does not require an overhead computation, because it usually computes the task in the clear. The overhead mostly derives from the replication on multi-clouds. Usually three clouds a required, so the total costs to the client are tripled. From the cost-saving figures showed earlier, it is clear that using 3 or more clouds for verification is very likely to cost more than simply using on-premises IT infrastructures.

Problem Statement In general, a verifiable cloud with a competitively low cost is what we want. The guarantee of the computation and the cost of adoption the cloud must be less or equal than the client pay and use when using on-premises IT infrastructures. To achieve this, we pick the second approach replication-based approach since it is the most practical approach. According to the above analysis, to use a low or equal cost solution compared to on-premises IT, the client should not assign the task to more than two clouds as well as minimizing the overheads. Collusion is the biggest challenging of using two clouds. The client might accept a wrong result without realizing if the two clouds collude and output the same result. It becomes even more challenging when heavyweight cryptographic protocols have to be avoided to reduce the overhead to an acceptable level. To this end, we resort to the new technology, namely Smart contract for mitigating the problem.

The idea Rather than forbidding or preventing collisions through technical means such as cryptography, we work towards undermining, through economic means, the foundation that collusion is grounded on. This should not be surprising since collusion is a topic studied in economics for many years. Three insights from economists establish the premise of our work:

- Collusion occurs whenever it is more profitable to all of the participants than their feasible alternatives [88]. Since economic incentives often drive collusion, imposing high fines on collusion has become a major instrument for preventing collusions in the real world. The fines make collusion a less profitable choice than not colluding, thus offset the motivation for collusion.

- Colluding parties have their interests, and this is a source of tension between them [71]. Colluding parties are not a single corporate entity. More interestingly, they are often competitors who collude in order to gain extra profit. Nevertheless, each party is responsible for its own and acts in its interest. Under suitable conditions, collusion can dissolve, and competition can resume.

- The most pressing problem for the colluding parties is how to prevent cheating. This is a natural consequence of pursuing self-interest, i.e. parties act in their interest and try to maximize their profit. In fact, the central difficulty of collusion is that it is often profitable for firms to secretly deviate from the collusive agreement [71].

Our key idea is to sabotage collusion by using smart contracts. Here smart contracts materialize self-enforcing agreements and payments that serve multiple purposes:

1. To weaken the incentive for collusion by taking a deposit from the clouds as security for the delivery of the correct result. The clouds will be penalized by losing their deposit should they deliver a wrong result.

2. To create an incentive for correct computation by redistributing the fine to the honest cloud as a reward.

3. To create distrust between he colluders by incentivizing them to betray their partner in the collusion coalition.

On the whole, we intend to make collusion a less favorable choice and make it much harder for potential colluding parties to trust each other, so that rational parties will stay away from collusion because it is unprofitable and too risky.

Contributions Based on the idea above, we designed two smart contracts (the Prisoners contract and the Traitors contract) to be used in scenarios where a client outsources a computation task to two clouds and cross-checks the results from the two clouds. With moderate and reasonable assumptions, the contracts guarantee that the two clouds, if they are rational, will behave honestly even though they have the opportunity to collude together and cheat. We conducted a detailed game theoretical analysis of the contracts. We proved that for the two clouds, both being honest and not colluding is the unique sequential equilibrium (a stronger form of Nash equilibrium) of the game. We also show the feasibility of the contracts by building them for the Ethereum network. We created the contracts using Solidity and executed them on the official Ethereum network. We provide a breakdown of financial and computational overheads for our contracts. Our figures show that the total transaction cost for executing each contract is below $1.

The Prisoners contract is to be signed by a client and two clouds. The name comes from the fact that the contract induces a game similar to the famous Prisoners Dilemma game between the two clouds. At a high level, the contract says that the

client will pay the two clouds to compute a task, but to get the job, each cloud has to pay a deposit. The honest cloud will get its deposit back later, and the cheating cloud will lose its deposit (if cheating is detected). Moreover, if one cloud cheats and one cloud is honest, the cheating clouds deposit goes to the honest cloud as a bonus (after deducting certain necessary costs). Similar to in the Prisoners Dilemma game, although it seems both clouds gain most by colluding with each other, both clouds eventually end up being honest. This is because they know the other will act in its interest, which means they will deviate from collusion for a higher payoff.

The problem with the Prisoners contract is that it only works if the two clouds cannot make credible and enforceable promises. This is not true, especially with the help of smart contracts. We demonstrate this by the **Colluders Contract**, which is a secret smart contract between the two clouds. In the contract, the cloud who initiates the collusion coalition agrees to pay a bribe to incentivize the other cloud to collude. More importantly, both clouds make a commitment by paying a deposit which will be taken if they do not follow the collusion strategy. The contract changes the game: when the deposit is high enough to offset the benefit a cloud can gain by betraying the other, betrayal is no longer more profitable, and collusion becomes the best strategy for both clouds.

To bust this form of more robust collusion coalition policed by collusion agreements such as the Colluders contract, we designed the Traitors contract. Intriguingly, the **Traitors contract** works not by countering the collusion agreement directly, but by forgiving one (and only one) cloud which follows the collusion strategy. The aim of the Traitors contract is not to incentivize the clouds to deviate from the collusion but to encourage them to report the collusion to the client. By getting information about collusion, the client can further investigate the case and punish the cheating cloud. By following the collusion strategy, the reporting cloud avoids the punishment imposed by the collusion agreement, thus making the agreement useless. If the other cloud does cheat, the reporting cloud will get a reward, which makes reporting the most profitable strategy. Overall, reporting is risk-free (the reporting cloud will not be punished by the Prisoners contract and the Colluders contract)and more profitable. The consequence is that both clouds know that if they try to initiate a collusion coalition, the other will collude but also report it to the client. This creates distrust between the clouds so that neither will want to initiate the collusion coalition, and they will stay honest to avoid being betrayed and punished.

The main cost of our smart contract based solution is the cost of employing two clouds to compute (in the clear) the same task. We assume that an offline

Trusted Third Party (TTP) is available to resolve the dispute when an inconsistency or anomaly is detected. However, if the two clouds are rational, the TTP will never be involved. Even if in the unlikely cases the TTP is called upon, the cost for dispute resolution is borne by the faulty cloud, not the client. The implementation of the contract requires only a few (constant number) additional cryptographic operations that are very light. Our experiments on the official Ethereum network show that the transaction cost for using smart contract facilities is small.

Potential applications alternative workload definition could be a smart contract for particular applications, e.g., health, e-voting, but we did not define such type of application-dependent workload.

6.3 Adversary Model and Assumptions

A Contractual Verifiable Cloud Computing scheme allows a client to outsource the computation of a function f on input x to two clouds. An honest client can then verify the correctness of results by simply testing whether the results from the two clouds are equal. We treat the clouds as rational adversaries. That is, the clouds are autonomous parties whose behaviors are driven by the motivation of maximizing their payoffs. If trusted auditing services are available to provide proper evidence, then these two types of faults can be treated differently. We assume the clouds are physically isolated and model each cloud as an individual *rational adversary*. Rational means that a party always acts in a way that maximizes its payoff, and is capable of thinking through all possible outcomes and choosing strategies which will result in the best possible outcome. Compared to assuming a malicious adversary who will act arbitrarily, rational is more realistic when modeling corporate behavior of the clouds. Indeed, a cloud provider is more likely to cut corners to maximize its profit than maliciously attack the client with no reason. On the other hand, rational adversaries are weaker than malicious adversaries because rationality precludes specific strategies. There is a trade-off between the level of security guarantee and costs. In the case that adversaries may behave irrational, cryptography-based approaches could be used to ensure verifiability.

We assume incorrect computation costs less (e.g. by skipping part or all of the computation), so the clouds are motivated to cheat. For simplicity, we assume a cloud can come up with an incorrect but plausible answer (cannot be easily proved to be wrong) at no cost. In reality, this is not free. However, assuming such an answer can be picked with no cost guarantees that the lower bound of deposits we

derive later is always valid because the cheating cloud loses strictly more if the cost of picking such an answer is more than 0. We view collusion as coordinated actions that follows from a mutual agreement between the adversaries. In reality, even if parties collude, they still retain their separate judgement and act in their interests. Therefore modeling each cloud as an individual adversary is more realistic than as a monolithic adversary who corrupts and controls multiple clouds. We assume the adversaries are computationally bounded so all cryptographic primitives we need to use remain secure.

We assume there exist one or more cryptocurrencies that support smart contracts. Most smart contracts platforms are experimental now, but there has been much effort to bring them into the real world. We assume the currency in these systems carries a certain amount of monetary value and is accepted by all parties under consideration as a medium of exchange. We assume the value of the currency is stable during the whole lifetime of the contract (and contracts derived from it). We assume the cryptocurrencies are secure, and the smart contracts are executed faithfully.

We assume the existence of a trusted third party (TTP), who is offline most of the time but can be called upon to recompute the task and resolve any disputes. We stress that if the clouds are rational, then the TTP would never be involved. The very existence of such a TTP provides a deterrence power which the adversaries have to take into account when making decisions. Even without taking actions, the TTP is a tangible threat to the adversaries and will have a controlling influence over them. The idea is similar to some strategic concepts in modern warfare and politics, e.g. "fleet in being" and "nuclear deference".

We also assume the following:

- The task to be computed is deterministic or can be reduced to being deterministic, e.g. by providing seed and using a pseudorandom generator for the random choices if the task is probabilistic. This is a common requirement in replication-based verifiable computation. We also assume the probability of guessing the correct result is small (e.g. by using inner state hash [18]).
- The task to be computed is not time-critical. We rely on the smart contract network to enforce the contracts, which may have large latency. The latency greatly depends on the status and parameters of the smart contract network, and we will be unlikely to get any guarantee for time-critical tasks.
- The two cloud able to communicate securely and through reliable authenticated public or private channels. The channel can provide non-repudiable evidence of sending/receiving messages if requested.

- Assuming equal cost greatly simplifies the analysis because of this rules out collusions for monopoly/oligopoly purposes in which the strategies are very different, and the correctness of the computation result is not the focus.
- The client has a low computational capability. This means the client needs the TTP to verify the correctness of the task's result. Also, unless there is clear evidence of incorrect result, the client is lazy to ask TTP to verify results.
- Funds only flow among the parties under consideration, not to/from external parties. For example, we do not consider fines imposed by legal systems or bribes offered by the client's rival in exchange for the clouds to output a wrong result. In general, if the cloud can gain additional benefits, one solution could be to increase the deposit. When the increment of the deposit is large enough and surpasses the benefit, the cloud will behave honestly because otherwise, the payoff will be worse than behaving honestly.
- Parties are risk neutral. For other risk profiles (risk seeking or risk aversion), the utility function can be adjusted to the risk profile, and the equilibria still hold by choosing the deposits according to the risk profile.

6.4 Monetary Variables

Below are the monetary variables we will use in the contracts (listed in alphabetic order). They are all non-negative.

- b: the bribe paid by the ringleader of the collusion to the other cloud in the collusion agreement (the Colluder's contract).
- c: the cloud's cost for computing the task.
- ch: the fee to invoke the TTP for recomputing a task and resolving disputes.
- d: the deposit a cloud needs to pay to the client to get the job.
- t: the deposit the colluding parties need to pay in the collusion agreement (the Colluder's contract).
- w: the amount that the client agrees to pay to a cloud for computing the task.
- z: shorthand for $w - c + d - ch$

The following relations hold for obvious reasons:

- $w \geq c$: the clouds do not accept underpaid jobs.
- $ch > 2w$: otherwise there is no need to use the clouds, the client uses the TTP for the computation. Note that the cheating cloud will pay ch. An honest client pays strictly no more than hiring two clouds (plus the mere transaction cost).

The following relations need to hold when setting the contracts for the desirable equilibria to hold. The client can set the parameter d in the Prisoner's contract, b and t can be set by the clouds in the Colluder's contract (see explanations in later sections):

- $d > c + ch$
- $b < c$
- $t > z + d - b$

6.5 The Prisoner's Contract

In this Section, we present the Prisoners' contract. The name comes from the fact that the contract induces a game similar to the classical Prisoners' Dilemma game.

As a starting point, we put a constraint that communication between the clouds is limited to "cheap talk", i.e. unlimited cost-free exchange of unverifiable and non-binding messages. In other words, the clouds can exchange information, but the information they get from the other cannot be regarded as truth or credible commitments.

6.5.1 The contract

The Prisoner's contract is an outsourcing contract signed between a client and two clouds. At a high level, it tries to incentivize correct computation by asking the clouds to pay a deposit upfront. If a cloud behaves honestly, the deposit will be refunded; if cloud cheats (and is detected), the deposit will be taken by the client. Moreover, in the case where one cloud is honest and one cheat, the honest cloud gets an additional reward that comes from the deposit of the cheating cloud. The intuition is to create a Prisoner's dilemma between the clouds: although collusion leads to a higher payoff than both behaving honestly, there is an even higher payoff if one can lure the other into cheating while being honest itself. Once both clouds understand this, they know collusion is not stable because the other cloud will always try to deviate from it. Any attempts (without a credible and enforceable promise) to persuade the other to collude will be deemed to be a trap and thus will not be successful. The contract is presented below, and more comments will follow afterwards.

1. The contract should be signed between a client (CLT) and two clouds ($\mathsf{C}_1, \mathsf{C}_2$). Should there be any dispute, the dispute will be resolved by a trusted third party TTP.
2. $\mathsf{C}_1, \mathsf{C}_2$ agree to compute a function $f()$ on an input x. Both $f()$ and x are chosen by CLT.

3. The parties agree on deadlines $T_1 < T_2 < T_3$.
4. CLT agrees to pay w to each cloud for the correct and timely computation of $f(x)$.
5. As a condition, each of $\mathsf{C}_1, \mathsf{C}_2$ must pay a deposit of amount d when signing the contract. The deposit will be held by the smart contract.
6. $\mathsf{C}_1, \mathsf{C}_2$ must pay the deposit before T_1. If any C_i fails to do so, the contract will terminate and any deposit paid will be refunded.
7. $\mathsf{C}_1, \mathsf{C}_2$ must deliver the computation result $f(x)$ before T_2.
8. Upon receiving the computation result from both $\mathsf{C}_1, \mathsf{C}_2$, or when the deadline T_2 has passed, CLT should do the following:
   (a) If both $\mathsf{C}_1, \mathsf{C}_2$ failed to deliver the result, their deposits will be taken in full by CLT;
   (b) If both $\mathsf{C}_1, \mathsf{C}_2$ delivered the result, and the results are equal, then after verifying the results, CLT must pay the agreed amount w and refund the deposit d to each C_i;
   (c) Otherwise CLT will raise a dispute to TTP.
9. Upon receiving a dispute raised by CLT, TTP computes $f(x)$. Let y_t, y_1, y_2 be the results computed by $\mathsf{TTP}, \mathsf{C}_1, \mathsf{C}_2$ respectively. Then the cheating party can be decided by the following rule:
   (a) For each C_i, if C_i failed to deliver the result, C_i cheated;
   (b) For each y_i ($i \in \{1, 2\}$) delivered before the deadline, if $y_i \neq y_t$, C_i cheated;

   TTP communicates the decision to CLT as well as to $\mathsf{C}_1, \mathsf{C}_2$.
10. Upon receiving TTP's decision, the dispute is resolved as follows:
    (a) If none of $\mathsf{C}_1, \mathsf{C}_2$ cheated, CLT must pay the agreed amount w and refund the deposit d to each C_i, and pay the fee for resolving the dispute ch to TTP.
    (b) If both $\mathsf{C}_1, \mathsf{C}_2$ cheated, their deposits will be taken in full by CLT, and CLT pays the fee ch to TTP.
    (c) If only one of $\mathsf{C}_1, \mathsf{C}_2$ cheated, then (1) the deposit of the cheating cloud will be taken in full by CLT, and (2) CLT pays the honest cloud w plus a bonus $d - ch$ and refunds its deposit d. CLT pays the fee ch to TTP.
11. If after $T_3 > T_2$, the client has neither paid nor raised dispute, then for any cloud C_i who delivered a result before T_2, CLT must pay C_i the agreed amount w and refund its deposit. Any deposit left after that will be transferred to CLT.

In the contract there are various deadlines ($T_1 < T_2 < T_3$). The deadlines are used to enforce timeliness and also to avoid locking away funds if some parties refuse to move forward. The latter is particularly important in smart contracts as the balance in a contract is controlled by a program. Without specific deadlines and code specifying what to do after the deadlines, the fund can be locked forever by the contract. Note that we assume the client is honest; therefore, Clause 11 will never be invoked in this case. The clause is included in the contract to assure the clouds that their funds will not be locked.

Clause 8 says that the client is empowered to settle the contract only when there is an obvious fault, i.e. none of the clouds delivers the result, or when he is satisfied with results. In all other situations, e.g. when only one result is received, or the results do not match, the TTP must settle the contract. Clauses 9 and 10 deal with the cases in which the TTP is involved. The TTP declares who cheated and then the TTP's judgement dictates the penalty/reward. If the client is honest, the dispute is only raised when something went wrong, and the cost for dispute resolution is covered by the deposit(s) of the cheating cloud(s).

6.6 The Colluder's Contract

In the previous Section, we consider the case where the clouds can only communicate via cheap talk. In this Section, we remove this constraint. We will show how clouds change the game by using smart contracts and make collusion a favorable choice for both.

6.6.1 The Contract

In the Prisoners' contract, two clouds will behave honestly because they know once they committed to send a wrong result, the other party will take advantage of this and send the correct result to get a higher payoff. Even if the other party promises to collude, the promise cannot be trusted because the other party might just be lying about its true intention.

Even if they verbally agree on colluding, there is nothing to prevent a party from deviating collusion to get a higher payoff. However, this only works if the colluding parties cannot make binding commitments. In the real world, the collusion coalition can often form after having an enforceable agreement among the colluders to redistribute profit and to punish those who deviate from collusion. Essentially, since the main problem is that collusion does lead to high enough payoffs, the agreement

imposes additional rules that will affect the parties' payoffs to make collusion the most profitable strategy for all colluding parties.

Essentially, since the main problem is that collusion does lead to high enough payoffs, the agreement imposes additional rules that will affect the parties' payoffs to make collusion the most profitable strategy for all colluding parties.

The collusion agreement is captured by the Colluders' contract that can be used to counter the Prisoners' contract. . The contract is presented below:

1. The contract should be signed by two clouds C_1 and C_2. We call the cloud who initiates the collusion the ringleader (LDR). The ringleader can be either C_1 or C_2. We call the other cloud the follower (FLR).
2. LDR and FLR agree to deliver a value $r \neq f(x)$ as the computation result in **CTP**, which is a Prisoner's Contract signed by LDR and FLR and a client CLT to compute$f()$ on input x.
3. As a condition, LDR must pay $t + b$ and FLR must pay t when they sign the Colluder's contract. The amount will be paid into and held by the smart contract.
4. LDR and FLR must pay the amounts stated above before $T_4 < \mathbf{CTP}.T_2$, where $\mathbf{CTP}.T_2$ is the result delivery deadline specified in **CTP**. If anyone fails to do so, the contract will terminate and any deposits paid will be refunded.
5. Once **CTP** has concluded, the following will be done to the balance held by the contract:
   (a) (Both follow) If both LDR and FLR output r in **CTP**, then t is paid to LDR and $t + b$ is paid to FLR;
   (b) (FLR deviates) Else if LDR outputs r in **CTP** and FLR's output in **CTP** is not r, then $2 \cdot t + b$ is paid to LDR and FLR gets nothing;
   (c) (LDR deviates) Else if LDR's output is not r in **CTP** and FLR outputs r in **CTP**, then $2 \cdot t + b$ is paid to FLR and LDR gets nothing;
   (d) (Both deviate) Else $t + b$ is paid to LDR and t is paid to FLR.

The contract must be signed before $\mathbf{CTP}.T_2$ because otherwise, it would be too late. The clouds needs to deliver the results in **CTP** (Prisoner's contract) before $\mathbf{CTP}.T_2$. The collusion agreement must be signed before this time so that the clouds know for sure that the collusion is secured and can deliver r without any risk. In clause 5d, when both clouds deviate from collusion, none of them is punished. Of course, another choice is to punish both in this case. The analysis of this variant is similar, and the equilibrium remains the same.

6.7 The Traitor's Contract

In the previous Section, we showed the Colluders contract that captures and enforces a collusion agreement. The contract enables two clouds to collude and ensures that no one will deviate from collusion. In this Section, we show the Traitors contract, which is designed to address the collusion problem and force the clouds to behave honestly.

6.7.1 The Contract

Knowing there could be a Colluder's contract between the two clouds, one way to solve the problem is to design another contract to incentivize the clouds to deviate from the collusion. The contract could deter collusion and make both clouds stay honest, or at least make cheating detectable if it can keep one cloud honest. The first difficulty when going this way is how to avoid creating a counter/counter-back loop. The client can provide an additional reward to the honest cloud and change the equilibrium so that collusion is less preferable. However, once the clouds know what is offered in the contract, they may be able to create a counter contract so that collusion becomes the equilibrium again. This loop can go endless.

The second difficulty is how to persuade a cloud to betray the other. In the Traitor's contract, rather than incentivizing one of the clouds to deviate from the collusion, it tries to incentivize the clouds to report the collusion. The reporting cloud is permitted to follow the collusion strategy, thus can get away from the punishment prescribed by the Colluders' contract. If a collusion is reported, the TTP will step in and decide who cheated. The reporting cloud would get a reward if the other cloud did cheat.

In short, the Traitor's contract is a *leniency policy*. It offers the first cloud who reports a collusion to the client the total immunity of the penalty that is imposed by the Prisoners' contract. It also offers the reporting cloud a reward if the collusion is real. The aim is to encourage a cloud to betray the partner in the collusion coalition. The subtlety of the Traitor's contract is that the immunity granted will allow the reporting cloud to betray the partner while pretending to follow collusion strategy secretly. By doing so, the reporting cloud can get away from the punishment imposed by the Colluders' contract. In consequence, betrayal is preferable to staying in the collusion coalition because it is risk-free and leads to a higher payoff. The Traitor's contract destabilizes collusion by encouraging betrayal. Moreover, the fear of betrayal creates distrust between the clouds. The distrust will eventually deter the formation of

the collusion coalition. In addition to all above, the Traitor's contract also includes a clause to punish misreporting, i.e. a cloud reporting a fabricated case to gain benefits. Experience from real-world shows that misreporting is a serious issue, especially for leniency policies that offer a reward. Therefore being able to deal with it is important. The contract is presented below:

1. The contract should be signed between a client (CLT) and a cloud who reports collusion. We call this cloud the Traitor (TRA). CLT and TRA must have signed **CTP**, a Prisoner's contract.
2. CLT only signs the Traitor's Contract with the first cloud who reports the collusion. CLT agrees to compensate TRA's loss in **CTP** in suitable cases.
3. TRA must deliver the computation result of $f(x)$ in this contract, which can be different from the one delivered in **CTP**.
4. As a condition, CLT must pay a deposit of amount $w + 2 \cdot d - ch$ that equals the maximum amount TRA could lose in **CTP** plus the reward. TRA must pay a deposit of amount ch that equals the fee for dispute resolution. The deposits will be held by the smart contract.
5. The contract should be fully signed before $\mathbf{CTP}.T_2$, the deadline for delivering the result in **CTP**. Otherwise the contract terminates and any deposit paid will be refunded.
6. TRA must deliver a result in this contract before $\mathbf{CTP}.T_2$.
7. CLT always raises a dispute instead of invoking Clause 8 in **CTP**.
8. Once **CTP** is settled by TTP, the following will be done to the deposits held by this contract:
   (a) If in **CTP** none of the clouds cheated (as asserted by TTP), then CLT's deposit $w + 2 \cdot d - ch$ is refunded, and TRA's deposit ch is paid to CLT. Nothing is paid to TRA;
   (b) Else if in **CTP** the other cloud did not cheat and TRA cheated and TRA delivered a correct result in this contract, then $2 \cdot d - ch$ is paid to CLT and $w + ch$ is paid to TRA;
   (c) Else if in **CTP** both clouds cheated and TRA delivered a correct result in this contract, then TRA gets back its deposit ch. TRA is also paid $w + 2 \cdot d - ch$. Nothing is paid to CLT;
   (d) Else $w + 2 \cdot d - ch$ is paid to CLT and ch is paid to TRA.

9. If TRA delivered a result in this contract, and **CTP**.T_3 has passed, then all deposits, if any left, go to TRA.

To report collusion, TRA must follow the following procedure:

(i) Wait until the Colluder's contract has been created and signed by the other cloud.
(ii) Before signing the Colluder's contract, reports the collusion to the client. Optionally, TRA can submit evidence of collusion, e.g. the address of the Colluder's contract and the value r that to be output in the event of collusion.
(iii) Sign the Colluder's contract only after it has signed the Traitor's contract with the client.

CLT only signs the Traitor's contract with the first cloud who reports the collusion. This is because in our case, the collusion coalition has only two members. It is too generous to forgive both of them. Once the Traitor's contract is fully signed, CLT always raises a dispute in **CTP**. There are two potential punishments imposed on TRA by the Prisoner's contract and the Colluder's contract. To ensure that TRA's payoff is not worse off in the event of a true collusion, TRA needs to deliver r in **CTP** to get away from the punishment imposed by the Colluder's contract, and then deliver $f(x)$ in the Traitor's contract to get the compensation of the penalty imposed by **CTP** (the Prisoner's contract). It is important that TRA follows the procedure to ensure it signs all three contracts or only **CTP**; otherwise it might have to bear a loss (see Game 3 and Game 4 in the following sections). To dispel TRA's concern of being cheated to "turn in", CLT pays into the contract $w + 2 \cdot d - ch$ to assure TRA that its loss will be compensated and its reward will be given.

Before reporting, TRA needs to wait until the other cloud has signed the contract, i.e. fully committed to collusion. Otherwise, if TRA reports and the other cloud decides not to sign the Colluder's contract, TRA will be in the situation of (unintentional) misreporting because the other cloud can deliver the correct result in **CTP**. When reporting, TRA can submit evidence of collusion. Note that the evidence submitted by TRA is a "best-effort proof". The purpose of the evidence is not to convince the client about the collusion, but to give the client more information about the collusion. The conclusive evidence of collusion/cheating is TTP's decision and the settlement of Traitor's contract (clause 8) relies only on values in Prisoner's contract and TTP's decision. CLT will sign the Traitor's contract even if the evidence is not strong or verifiable. TRA can falsely report with some fabricated evidence, but as we will show in the next Section, a rational cloud will not misreport. This is

because when signing the contract, TRA needs to pay ch into the contract and will lose this amount in the event of misreporting.

6.8 Address and Pseudocode

In this section, we show the pseudocode for the three contracts explained earlier. All of these contracts were implemented in Solidity. Also we deployed and ran these contracts on the Ethereum official network as well as the test-net. Below are the account addresses of each contract which can be viewed through [1] or any Ethereum client and their pseudocodes.

6.8.1 Contract Account Address

- Prisoner's Contract:
  `0x09b61d58448d580c42b387334ac3fe28f2868887`
- Colluder's Contract:
  `0x255309e0612de2ab1812e21190b9a9b8f9a216d8`
- Traitor's Contract:
  `0x57b032d5a6adcc67739e8fd87a00c69bedbf7c65`

6.8.2 Prisoner's Contract

Prisoner's Smart Contract

Init: Set state := INIT, deposit := {}, worker := {}, result := {}

Create: Upon receiving from a client CLT
("create", $com_f, com_x, w, d, ch, T_1, T_2, T_3$, TTP):
Assert state = INIT *and* $T < T_1 < T_2 < T_3$ *and* ledger[CLT] $\geq \$(2 \cdot w + ch)$
ledger[CLT] := ledger[CLT]$-\$(2 \cdot w + ch)$
deposit := deposit $\cup$ (CLT, $\$(2 \cdot w + ch)$) state := CREATED

Bid: Upon receiving ("bid") from a Cloud C_i:
Assert state = CREATED *and* $T < T_1$ *and* $(C_i, \$d) \notin$ deposit *and* ledger[C_i] $\geq \$d$
ledger[C_i] := ledger[C_i] $-\$d$
deposit := deposit $\cup$ ($C_i, \$d$)
worker := worker $\cup$ C_i
if |worker| = 2 then state := COMPUTE

[1] `https://etherscan.io`

Deliver: Upon receiving ("output", com_{y_i}) from a cloud C_i:
Assert state = COMPUTE *and* $T < T_2$ *and* $\mathsf{C}_i \in$ worker *and* $(\mathsf{C}_i, *) \notin$ result
result := result $\cup$ $(\mathsf{C}_i, com_{y_i})$
if |result| = 2 then state := PAY

Pay: Upon receiving ("pay", $NIZK$) from CLT:
Assert state = PAY *and* $T < T_3$
if |result| = 0 then ledger[CLT] := ledger[CLT] $+\$(2 \cdot w + 2 \cdot d + ch)$
state := DONE
else if |result| = 2 *and* $verify(NIZK, com_{y_1}, com_{y_2}) \rightarrow y_1 = y_2$ then
ledger[C_1] := ledger[C_1] $+\$w + \$d$
ledger[C_2] := ledger[C_2] $+\$w + \$d$
ledger[CLT] := ledger[CLT] $+\$ch$
state := DONE else state := ERROR

Dispute: Upon receiving ("resolve", com_{y_t}, $NIZK_1$,$NIZK_2$) from TTP:
Let result = $(\mathsf{C}_1, com_{y_1}), (\mathsf{C}_2, com_{y_2})$
Cheated := [false,false]
for $i = 1$ *to* 2
if $NIZK_i = NULL$ then Cheated[i] := true
Else if $verify(NIZK_i, com_{y_i}, com_{y_t}) \rightarrow y_i \neq y_t$ then Cheated[i] := true
if Cheated[1] *and* Cheated[2] then ledger[CLT] := ledger[CLT] $+\$2 \cdot (w + d)$
else if ¬Cheated[1] *and* ¬Cheated[2] then ledger[C_1] := ledger[C_1] $+\$w + \$d$
ledger[C_2] := ledger[C_2] $+\$w + \$d$
else if ¬Cheated[1] *and* Cheated[2] then ledger[C_1] := ledger[C_1] $+\$(w+2\cdot d-ch)$
ledger[CLT]:= ledger[CLT] $+ w + ch$
else if Cheated[1] *and* ¬Cheated[2] then ledger[C_2] := ledger[C_2] $+\$(w+2\cdot d-ch)$
ledger[CLT]:= ledger[CLT] $+ w + ch$
ledger[TTP] := ledger[TTP] $+\$ch$
state := DONE

Timer: if $T \geq T_1$ *and* state = CREATED then refund(deposit) state := ABORTED
else if $T \geq T_2$ *and* state = COMPUTE then state := PAY
else if $T \geq T_3$ *and* state = PAY then for each (a,b) in result
deposit := deposit $-(\mathsf{CLT}, \$w) - (a, \$d)$
ledger[a] := ledger[a] $+\$w + \$d$
Let res be any amount left in deposit ledger[CLT] := ledger[CLT] $+\$res$
state := DONE

6.8.3 Colluder's Contract

Colluder's Smart Contract

Init: Set $\mathsf{state} := \mathsf{INIT}$, $\mathsf{deposit} := \{\}$

Create: Upon receiving the message ("create", CTP, C_2, $com(r)_1$, $com(r)_2, t, b, T_4, T_5$) from C_1:
Assert $\mathsf{state} = \mathsf{INIT}$ *and* $\mathsf{CTP} = G(Prisoner'sContract)$ *and* $T < T_4 < \mathsf{CTP}.T_2 < \mathsf{CTP}.T_3 < T_5$ *and* $\mathsf{CTP.state} = \mathsf{COMPUTE}$ *and* $\mathsf{ledger}[\mathsf{C}_1] \geq \$(t+b)$
$\mathsf{ledger}[\mathsf{C}_1] := \mathsf{ledger}[\mathsf{C}_1] - \$(t+b)$
$\mathsf{deposit} := \mathsf{deposit} \cup (\mathsf{C}_1, \$(t+b)$
$\mathsf{state} := \mathsf{CREATED}$

Join: Upon receiving the message ("join") from C_2:
Assert $\mathsf{state} = \mathsf{CREATED}$ *and* $T < T_4$ *and* $\mathsf{CTP.state} = \mathsf{COMPUTE}$ and $\mathsf{ledger}[\mathsf{C}_2] \geq \$t$
$\mathsf{ledger}[\mathsf{C}_2] := \mathsf{ledger}[\mathsf{C}_2] - \$t$
$\mathsf{deposit} := \mathsf{deposit} \cup (\mathsf{C}_2, \$t)$
$\mathsf{state} := \mathsf{COLLUDED}$

Enforce: If $T \geq T_5$ *and* $\mathsf{state} = \mathsf{COLLUDED}$ *and* $\mathsf{CTP.state} = \mathsf{DONE}$ then
Let $(\mathsf{C}_1, com_{y_1}), (\mathsf{C}_2, com_{y_2}) = \mathsf{CTP.result}$
if $com_{y_1} = com_{r,1}$ *and* $com_{y_2} = com_{r,2}$, then
$\mathsf{ledger}[\mathsf{C}_1] := \mathsf{ledger}[\mathsf{C}_1] + \$t$
$\mathsf{ledger}[\mathsf{C}_2] := \mathsf{ledger}[\mathsf{C}_2] + \$(t+b)$
if $com_{y_1} = com_{r,1}$ *and* $com_{y_2} \neq com_{r,2}$, then
$\mathsf{ledger}[\mathsf{C}_1] := \mathsf{ledger}[\mathsf{C}_1] + \$(2 \cdot t + b)$
else if $com_{y_1} \neq com_{r,1}$ *and* $com_{y_2} = com_{r,2}$, then
$\mathsf{ledger}[\mathsf{C}_2] := \mathsf{ledger}[\mathsf{C}_2] + \$(2 \cdot t + b)$
else refund($\mathsf{deposit}$) $\mathsf{state} := \mathsf{DONE}$

Timer: If $T \geq T_4$ *and* $\mathsf{state} = \mathsf{CREATED}$, then refund($\mathsf{deposit}$) $\mathsf{state} = \mathsf{ABORTED}$

6.8.4 Traitor's Contract

Traitor's Smart Contract

(assuming C_2 is the traitor)

Init: Set $\mathsf{state} := \mathsf{INIT}$, $\mathsf{deposit} := \{\}$

Create: Upon receiving the message ("create", CTP, CTC, C_2) from CLT:
Assert state = INIT *and* CTP = $G(Prisoner'sContract)$
CTC = $G(Colluder'sContract)$ *and* $T <$ CTP.T_2 *and* CTC.state = CREATED or COLLUDED
Let d, w, ch be the same as in CTP
Assert ledger[CLT] $\geq \$(w + 2 \cdot d - ch)$
ledger[CLT] := ledger[CLT] $- \$(w + 2 \cdot d - ch)$
deposit := deposit $\cup$(CLT, $\$(w + 2 \cdot d - ch)$) state := CREATED

Join: Upon receiving the message ("join") from C_2:
Assert state = CREATED *and* ledger[C_2] $\geq \$ch$ *and* CTP.state = COMPUTE *and* $T <$ CTP.T_2
ledger[C_2] := ledger[C_2] $- \$ch$
deposit := deposit $\cup(\mathsf{C}_2, \$ch)$
state := JOINED

Deliver: Upon receiving the message ("output", $com_{y'}$) from C_2:
Assert state = JOINED *and* $T <$ CTP.T_2 *and* CTP.state = COMPUTE
state := COMPUTED

Check: Upon receiving the message ("check", $NIZK$) from CLT:
Assert state = COMPUTED *and* CTP.state = DONE
Cheated := CTP.dispute.Cheated
com_{y_t} := CTP.dispute.com_{y_t} Correct := false
if $verify(NIZK, com_{y'}, com_{y_t}) \rightarrow y' = y_t$ then Correct := true
if $\neg$Cheated[1] *and* $\neg$Cheated[2] then ledger[CLT] := ledger[CLT] $+ \$(w + 2 \cdot d)$
else if $\neg$Cheated[1] *and* Cheated[2] *and* Correct then ledger[C_2]:= ledger[C_2] $+ \$w$ ledger[CLT] := ledger[CLT] $+\$2 \cdot d)$
else if Cheated[1] *and* Cheated[2] *and* Correct then ledger[C_2]:= ledger[C_2] $+ \$w +$ $2 \cdot \$d$
else refund(deposit) state := DONE

Timer: If $T \geq$ CTP.T_2 and state = CREATED then refund(deposit) state := ABORTED
Else if $T \geq$ CTP.T_2 and state = COMPUTE then ledger[CLT] := ledger[CLT] + $\$(w + 2 \cdot d)$ state := DONE
If $T \geq$ CTP.T_3 and state = COMPUTED then ledger[C_2]:= ledger[C_2] $+ \$w + 2 \cdot \$d$
state := DONE

6.9 Implementation

We implemented the contracts in Solidity 0.4.4 [48] and tested them on the Ethereum network with Geth [44]. We used the CryptoCon [73], a smart contract that implements elliptic curve cryptography (ECC), for implementing cryptographic operations on the blockchain. The contracts are loosely coupled with the actual computation tasks as an external service. The actual computation tasks can be treated as, and the contracts do not need to know their internal details. The contracts will be called before/during/after executing the tasks, e.g. the input and output of the tasks. The source code of our contracts can be found at [2]. The protocols of the smart contracts can be found in the full version of the [34]. We ran the experiments on a MacBook Pro with a 2.8 GHz intel i5 CPU and 8GB RAM.

6.10 Overhead and Cost

Overhead The additional overhead incurred by cryptography is small. We implement the commitment and NIZK schemes in ECC. In each contract, each party need to generate at most two commitments. Also, in each contract, at most 2 NIZKs need to be generated and verified. The most costly cryptographic operation is the point multiplication (`MUL`) operation. Generating a commitment needs 2 `MUL`. Generating and verifying an equality NIZK each needs 2 `MUL` as well. Generating an inequality NIZK needs 4 `MUL` and verifying needs 3 `MUL`. The commitments and NIZKs are generated locally by the parties. On the blockchain, the peers only need to verify the NIZKs. The commitments and NIZKs are small in size. When using 256-bit ECC, a commitment is only 512 bits, an equality NIZK is 768 bits, and an inequality NIZK is 1536 bits. The size can be further reduced if point compression is used.

Financial Cost In Table 6.1, we show the cost of setting up and executing the contracts on the official Ethereum network. The cost is in the amount of gas consumed by each function, and the converted monetary value in the US dollar. The gas price was 2×10^9 ether (2 Gwei) in all transactions, and the exchange rate was 1 ether =$87.32. As we can see, the financial cost for using smart contracts on the Ethereum network is low. The cost is roughly related to the computational and storage complexity of the function. For example, in Prisoners contract, Init (to store a contract on the blockchain) and Dispute (require verification of NIZKs) cost more than other functions. For the Prisoners contract, the total cost (for the client and the

[2] `https://github.com/mjod89/SmartContracts`

Contract	Functions	Cost in Gas	Cost in $
Prisoner's	Init	2,298,950	0.4015
	Create	206,972	0.0361
	Bid	74,899	0.0131
	Deliver	94,373	0.0164
	Pay	821,244	0.1434
	Dispute	2,126,950	0.3714
Colluder's	Init	1,971,270	0.3443
	Create	281,852	0.0492
	Join	58,587	0.0102
	Enforce	103,156	0.0180
Traitor's	Init	2,018,459	0.3525
	Create	161,155	0.0281
	Join	66,802	0.0117
	Deliver	82,846	0.0145
	Check	719,051	0.1256

Table 6.1: Cost of using the smart contracts on the official Ethereum network. The transactions are viewable on the blockchain.

two clouds) is about 3.8 million gas ($0.65) if there is no dispute, or about 5 million gas ($0.88) with dispute resolution. For the Colluders contract, the total cost is about 2.4 million gas ($0.42).And for the Traitors contract, the total cost is about 3 million gas ($0.53). The cost can be further reduced if the contracts are reused. Note that Ethereum will have native support for ECC [76], which means we can expect a much lower cost for calling functions that involves ECC operations (e.g. Dispute).

6.11 Conclusion

Verifiability is a highly desirable property in cloud computing, cost-efficiency is another one. In this chapter, we propose a smart contract based solution aiming to achieve both. In our solution, the client outsources the same computation for two clouds and uses smart contracts to create games between two rational clouds. The games will restrain the clouds from colluding and cutting corners. Instead, they will stay honest to pursue their highest payoffs. Now without collusion, verifiability can be achieved by cross-checking the results returned by the clouds. The main cost is the cost of employing two clouds, and other costs are small. In this work, we assume the client is honest. One future direction would be to consider the client as a potential adversary. This would make the interplay among parties more complex and requires significant changes to the contracts. Another future direction would be to consider

repeated interactions among the parties. Repeated interactions introduce significant changes to the settings because the incentive can be now influenced by reputation and long-term profitability. Also, the current deposit mechanism is not very efficient from the cloud point of view. If the cloud has many clients and simultaneous contracts, the cloud must have a large cash reserve to pay all deposits at the same time. One direction would be to investigate more efficient deposit mechanisms by, e.g. pooling contracts or insurance. Currently, the contracts are written case-by-case. Ultimately we would like to have standard, verified and composable templates/subroutines, much like standard wording/clauses we use in traditional contracts. We would also like to develop counter-collusion contracts in general for other purposes, e.g. to prevent vote buying in e-voting systems like [73].

Chapter 7

Implementation and Evaluation of Non-Blocking Two Phase Commit Protocol Using Blockchain

7.1 Summary

Since the introduction of Bitcoin in 2008, cryptocurrencies such as Ethereum, which empowers users to implement and execute custom-made distributed applications leveraging smart contracts and blockchain technology, have gained a considerable interest by both academia and industry. To enhance accountability, audibility and trust, these technologies have been utilized in a variety of applications outside the domain of cryptocurrencies, such as in cloud computing [34], banking and energy trade[1].

In this chapter, relying on results we learned from Chapters 3 and 4, which show how to implement an efficient smart contract with less Gas and CPU usage, we present a blockchain-coordinated 2PC protocol that has rigorous arguments for its correctness under the synchrony requirements. Our focus is on the implementation of this protocol on both Ethereum private and testnet networks. We demonstrate, through our experiments, that the monetary cost of executing smart contracts is relatively small, that the protocol performance slows down when using a public blockchain like Ethereum, and that even major violations of synchrony requirements lead only to relatively small increases in unnecessary aborts. We thus identify a trade-off between improving protocol performance and admitting a risk that transactions could occasionally abort unnecessarily.

In summary, this chapter explores and exposes the impossibilities, the possibilities, the cost and the trade-offs involved in using a blockchain to implement non-blocking

atomic commit. Its structure and contributions are as follows. The next Section presents the atomic commit problem that 2PC solves, the notion of blocking and the distinction between synchronous *versus* asynchronous distributed systems. Assuming a synchronous system, Section 7.4 describes the traditional version of 2PC and explains the causes of 2PC blocking. It thus provides the essential background for Section 7.5 which describes in detail our contribution that is in the domain of *protocol design*: a non-blocking 2PC with a synchronous blockchain, together with pseudo-code for smart contracts and correctness arguments. Our *practical* contributions are detailed in Section 7.7, which describes an Ethereum based implementation of the protocol and discusses the results of our experiments. The discussions present the cost of smart contract execution, report both the estimated and observed worst-case 2PC execution latency values, quantify the probability of occurrence of unwarranted aborts caused by synchrony violations and point out the scope for a trade-off between improving performance and minimizing wasteful aborts. Finally, Section 7.8 concludes the chapter.

7.2 Introduction

Since the advent of Bitcoin in 2008 [75], cryptocurrencies have gained considerable interest. This is then followed by an even larger interest being accorded to Bitcoin's underlying technology, the blockchain, and to Ethereum's development of smart contracts that empower users to execute custom-made programs on a blockchain [10]. A variety of applications outside the cryptocurrency domain, such as finance [86], banking and energy trade [1], have been leveraging blockchain and smart contract technologies to enhance accountability, auditability and trust in their core processes.

This chapter investigates the use of these technologies in enhancing the availability of distributed database management systems [56, 69] and the associated cost. Precisely, we revisit a well-known impossibility result [83, 58] related to *blocking* in atomically committing database transactions and demonstrate that these new technologies, under certain conditions, help accomplish what would otherwise be impossible.

When multiple processes execute a database transaction in a distributed system, an atomic commit protocol ensures the essential requirement that all processes either commit the transaction or abort it - a requirement that is commonly known as *atomicity* or *agreement*. The two phase commit protocol (2PC, for short) is widely used as an atomic commit protocol due to its conceptual simplicity, ease of implementation and low message cost. It is, however, vulnerable to periods of non-progress

or *blocking*. This vulnerability is proven [83] to be inevitable even in *synchronous* distributed systems where bounds on delays (e.g., message transfer delays) can be reliably estimated, and the only type of undesirable events that can occur is process crash.

The definition of a 'synchronous' distributed system has long been established in the literature [15]. In our earlier work [37], we extended this definition for a blockchain system and developed a protocol in which the blockchain plays specific roles in the execution of 2PC. This protocol was shown to eliminate blocking when both the distributed system and the blockchain used are synchronous. Its design, however, required that the timestamps of blocks in a blockchain be increasing in value and that they emulate 'ticks' of a global clock to database servers. While the Ethereum blockchain meets this requirement, other blockchain systems do not and newly emerging ones may not. So, in this chapter, we remove this requirement and present a new protocol together with correctness arguments. This new version also eliminates blocking under synchronous constraints and retains the native structure of 2PC for database processes which makes it easily adoptable in legacy systems.

To the best of our knowledge, this work is the first in the literature to demonstrate that the impossibility result of Skeen [83] can be circumvented in synchronous distributed systems by using a synchronous blockchain. This revised and extended version not only improves on the earlier protocol but also addresses two significantly pertinent questions: can blocking be eliminated if the blockchain or the distributed system is *not* synchronous, and, if the answer is no, what are the practical implications if the blockchain and the distributed system can be synchronous most of the times, but not always?

Some blockchain systems, typically the public ones with miners having the freedom of choice in composing their blocks, may cease to be synchronous if it becomes harder to estimate delay bounds accurately. Similarly, a *cluster* hosting distributed database servers becomes asynchronous if the accurate delay bound estimation within the cluster is not guaranteed.

We are thus faced with four possible combinations: (i) the blockchain is synchronous, and the database cluster is asynchronous, (ii) blockchain is asynchronous, and the cluster is synchronous, (iii) both are asynchronous, and (iv) both are synchronous. 2PC blocking is eliminated for case (iv) as our protocol would demonstrate. Still to be addressed, therefore, is the question of whether 2PC blocking can be eliminated for the other three cases.

We argued that elimination of 2PC blocking cannot be guaranteed for (iii). We also prove that the same impossibility holds for more restricted cases of (i) and (ii) as well.

Thus, the impossibility results presented here are point to quite a fundamental result: a non-blocking 2PC using a blockchain is possible *if and only if* both the blockchain and the database cluster are synchronous. That is, many desirable features that a blockchain system has, such as reliability, immutability, etc., are not by themselves sufficient to eliminate 2PC blocking, and synchrony is required additionally.

Finally, when the blockchain and the distributed system are considered to be synchronous, even carefully computed delay-bound estimates are at risk of being violated, e.g., due to bursts in network traffic. We argue that such violations can cause some *commit*-worthy database transactions to *abort* unnecessarily, but cannot undermine the core *atomicity* requirement that all servers either *commit* or *abort*. We investigate the relation between the number of *unwarranted* aborts and the degree of violations in the synchronous assumption, and observe that the former is small even when the latter is large.

7.3 The Atomic Commit Problem

The problem is specified in the context of a set of distributed processes as follows: $\Pi = \{P_1, P_2, \ldots, P_n\}$, where $n > 1$ is known. A process P_i, $1 \leq i \leq n$, can crash at any time and recover after some arbitrary amount of time. Information *logged* in the disk prior to crash survives the crash. At any given instance, there are two complementary subsets of Π, the *crashed* and the *operative*. For discussions, we would assume that the former is small and a strict subset of Π.

Each operative process autonomously evaluates a *vote* that can be either *yes* or *no*. The problem is to have processes *decide* either on *commit* or *abort*, subject to the following four requirements [57]:

- *Agreement*: No two processes decide differently;
- *Termination*: All operative processes decide;
- *Abort-Validity*: Abort is the only possible decision if some process votes *no* or does not vote at all; and,

- *Commit-Validity*: Commit is the only possible decision if every process is operative and votes yes.

Agreement requires any two decided processes, currently crashed or operative, to have decided identically. Say, P_k decides on *commit* and immediately crashes; then no other process can decide on *abort* even if all but P_k are operative and deduce P_k to have crashed. *Termination* ensures that the decision is available to all working processes; in particular, if a process crashes undecided, it should be able to decide when it becomes operative again, post-recovery.

Abort-Validity permits a process with *no* vote, not to exercise its vote at all. *Commit-validity* rules out trivial solutions such as all processes perforce decide on *abort* irrespective of their votes. . This last requirement is impossible to guarantee even in blockchain based solutions when the worst-case delay estimates being used are not guaranteed to hold.

Observe that any non-trivial solution to atomic commit requires operative processes of Π to interact amongst themselves - either directly leading to *decentralized* protocols or via a protocol *coordinator* C leading to *centralized* versions. The former extract a huge message cost. The widely-used 2-Phase Commit (2PC) protocol is a centralized one and is highly message efficient. It would be our focus here. (In practice, the role of C is typically played by a designated process in Π.)

Definition. An atomic commit protocol is said to be *blocking*, if there *can* exist executions in which operative processes cannot decide until some non-empty subset of crashed processes ought to recover [83, 79]. Blocking is thus undesirable as the progress of operative processes, normally larger in number, is dictated by the recovery times of crashed ones. A protocol is *non-blocking* if operative processes are *guaranteed* to decide even if each crashed process is never to recover. Whether one can have a non-blocking atomic commit protocol or not, depends on if the distributed system is *synchronous* or *asynchronous* [58, 57].

7.3.1 Synchronous *vs* Asynchronous Systems

Definition: A distributed system is said to be *synchronous*, if bounds on processing delays and inter-process communication delays can be reliably estimated; otherwise, it is said to be *asynchronous* [58, 57].

Note that the bound estimates in a synchronous system can be large (typically, worst-case estimates) but must be finite and hold reliably. Typically, distributed sys-

tems where delays can fluctuate arbitrarily, and therefore reliable bound estimations are not possible, are classed as asynchronous.

It is known that non-blocking atomic commit is not possible when the distributed database system is asynchronous [58], unless the system obliges *every* execution by behaving in certain desirable ways [57]. It is, however, possible to have a non-blocking atomic commit in a synchronous system by using the message-expensive, decentralized approach [78, 35]. Intuitively, the design rationale in this approach is as follows. Reliable bound estimates in a synchronous system are used to implement *perfect* crash detection using timeouts: a crash is always detected, and an operative process is never mis-detected (no false positive/negative). In addition, protocol performance is speeded up by assuming a bound on the maximum number of processes that can crash [35].

Nevertheless, the centralized 2PC is a blocking protocol *even* in a synchronous system [83], i.e., even when a cluster hosting Π supports delay bounds to be estimated reliably and can thereby facilitate perfect crash detection!

7.3.2 Synchronous *vs* Asynchronous Blockchains

We observe that this synchronous *vs* asynchronous classification holds for blockchain based 21 systems[37] as much as for traditional distributed systems. (Earlier definitions [37] will be re-stated in § 7.5.2 for completeness.) In public blockchain systems, such as Ethereum, the time taken for a *valid* transaction to be *confirmed* or irreversibly placed in the blockchain is determined by a variety of delay-prone factors - both human as well as system related; for instance, a miner being (un)willing to include a transaction in their block [92] falls under the former category and factors such as the required number of follow-up blocks to assure blockchain linearity and incoming transaction rate fall under the latter.

Ethereum blockchain confirmation time for a transaction can be unbounded with a significant probability [92], suggesting large variances in end-to-end processing delays within the blockchain infrastructure. On the other hand, permissioned blockchain systems (e.g., HyperLedger [12]), with their hardened modular implementation of consensus protocols (e.g., [23]) over dedicated machines, appear to promise that the delays for transaction confirmation have small mean (in the order of milliseconds) and also small variance and can, therefore, be reliably bounded, thus making such systems candidates for a synchronous blockchain.

7.4 2PC in Synchronous Systems

The 2-Phase Commit protocol, 2PC for short, is explained below in the context of database transactions [69]. Shards of a database are distributed over processes in Π. We assume that a crash-prone process, called the *coordinator* and denoted as C, launches a multi-shard transaction that requires every process in Π to execute a set of serializable operations on their respective shards. We refer to this launching by C as each process in Π *getting_work* from C.

Let ω and δ denote upper bound estimates on the time any operative $P_i \in \Pi$ takes to complete its work and on message transfer delays between any two operative processes, respectively. Since the system is assumed to be synchronous, ω and δ always hold.

PHASE 1	
Coordinator C:	1. *Broadcast cast_vote* to all $P_1 \dots P_n$ 2. *Set Timeout* $\Delta = 2\delta$; *go to Phase 2*
P_i*:*	1. IF (*cast_vote* not received until T_i or $V_i = 0$) THEN *quit* ELSE {Log $V_i = 1$; *send* V_i to C; *Set timer*; *go to Phase 2*}
PHASE 2	
C *on timeout* Δ:	1. IF any absent V_i THEN *verdict = abort* ELSE *verdict = commit* 2. Log *verdict*; *Broadcast verdict* to all $P_1 \dots P_n$
P_i:	1. *Repeat* on *timer*: IF *verdict* arrived THEN Log *verdict* ELSE {request C; reset *timer*} 2. *Until verdict* logged

Figure 7.1: Two phase commit protocol.

C disseminates the work and awaits on a timeout of $(\omega + \delta)$ duration which is sufficient for any operative P_i to receive and complete the work given to it. At the expiry of the timeout, it initiates an execution of 2PC by broadcasting *cast_vote* to all processes - as shown in line 1, phase 1 for *Coordinator* C in Figure 7.1. This is then followed by setting a timer for $\Delta = 2\delta$ and proceeding to phase 2. (Note: C waiting for $(\omega + \delta)$ time before broadcasting *cast_vote* is not shown in Figure 7.1.)

When P_i receives work from C, it computes T_i as the local time when a duration $(\omega + 2\delta)$ would elapse after the receipt of the work. While doing the work, P_i will either complete it and set its vote $V_i = 1$ or decide that work cannot be completed in a serializable manner and set $V_i = 0$. In the latter case, by the *Abort-Validity* property, P_i can deduce that the decision or *verdict* is abort i.e., the transaction would be aborted system wide; so, P_i quits executing 2PC as shown in line 1 of Phase 1 for P_i in Figure 7.1.

Note that it is possible to have a 2PC implementation that makes P_i send $V_i = 0$ to C; we consider such an implementation only where relevant, but otherwise we will assume the common (and message-optimal) case of P_i with $V_i = 0$ simply halting the execution with *abort* decision.

If P_i has set $V_i = 1$, it waits to receive *cast_vote*. If *cast_vote* message is not received until T_i, P_i assumes that C has crashed, decides *abort* and quits its execution of 2PC. If, on the other hand, *cast_vote* arrives by T_i, P_i continues executing 2PC by logging its vote $V_i = 1$, sending V_i to C and proceeding to Phase 2. That is, the 'ELSE' part in line 1 of Phase 1 for P_i in Figure 7.1 is executed when (*cast_vote* not received until T_i or $V_i = 0$) is false which is equivalent to (*cast_vote* received before T_i and $V_i = 1$) becoming true.

Note that while a given P_i may or may not enter phase 2, C always does. When its Δ-timeout expires, C counts an absent vote from any P_k as $V_k = 0$; it decides on *commit verdict*, if $V_i = 1, \forall i : 1 \leq i \leq n$; on *abort* verdict, otherwise. The *verdict* decided is logged and broadcast to all P_i. (See Phase 2 of Figure 7.1).

Any P_i that executes phase 2, awaits *verdict* from C and requests C periodically (as per some timer value), if *verdict* is not forthcoming. This periodic request will prompt a crashed C to respond after its recovery by referring to the *verdict* it logged prior to the crash. If no *verdict* has been logged, C must have crashed prior to computing the *verdict*; in that case, C's response would be *abort*.

Similarly, if P_i crashes after sending $V_i = 1$ to C, it will observe, after recovery, the log entry of $V_i = 1$ and request C to send the *verdict*. Thus, all operative processes, including those that crash during execution and recover, decide - ensuring *termination*. It is easy to see that the other three requirements of atomic commit are also met in 2PC.

Figure 7.2 depicts the state transition diagram for any P_i where a circle denotes a state and a double circle a terminal state; a state transition is indicated by an unidirectional arrow with a label $\frac{I}{O}$ where I indicates the input received by P_i which causes the transition and O any output produced by P_i after the transition. ('-'

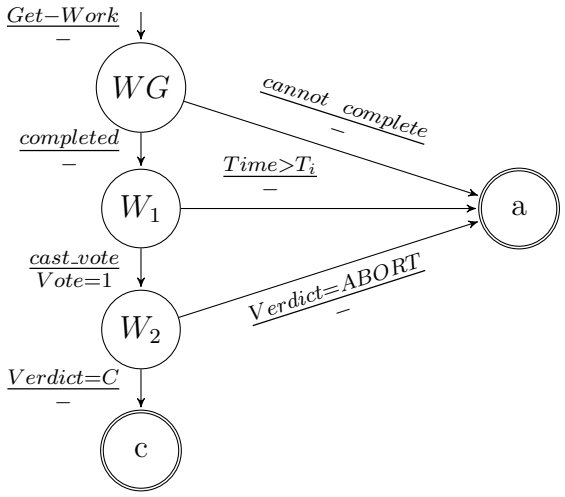

Figure 7.2: 2PC State Transition Diagram for Process P_i.

indicates null output.) WG, W_1 and W_2 represent states where P_i is doing the work given, waiting for *cast_vote* (see line 1, phase 1 in Fig 7.1) and for *verdict* (line 1, phase 2 in Fig 7.1, respectively; a and c denote the *terminal* states where P_i *aborts* and *commits*, respectively.

7.4.1 Inevitability of Blocking in 2PC

While Skeen [83] formally proves this inevitability, we offer here, for completeness, an intuitive understanding of the reasons for it. By the definition of blocking (see Section 7.3), in every execution of a non-blocking 2PC protocol, operative processes decide despite some processes crashing and staying crashed; i.e., operative processes reach a *verdict* that satisfies the atomic commit requirements without having to wait for any crashed process to recover.

We present three distinct execution *scenarios* of 2PC and show that no mechanism can possibly exist that avoids blocking in *all* scenarios and all meets all atomic commit requirements.

Scenario 1: In this execution of 2PC, every $P_i \in \Pi$ votes $V_i = 1$ and C crashes just before it is to broadcast its *verdict*. C remains crashed, i.e., does not recover, for a long time.

Each P_i is blocked until C recovers. Suppose that blocking is avoided by using some *repair* sub-protocol $\mathcal{R}$ that enables operative processes to decide on a *verdict* (here *commit*) without waiting for the crashed C to recover. For example, $\mathcal{R}$ may require operative processes to interact among themselves on how they voted and to

arrive at a *verdict* that C would have broadcast had it not crashed. Next two scenarios prove that $\mathcal{R}$ cannot exist.

Scenario 2: It is identical to *scenario* 1 except that one $P_k \in \Pi$ could not complete its work, decides on *abort* and then crashes. P_k also remains crashed for a long time.

$\mathcal{R}$ must now enable all operative P_i, $i \neq k$, to decide on *abort* without waiting for P_k or C to recover.

Scenario 3: It is also identical to *scenario* 1, except that C crashes after sending *verdict* = *commit* only to P_k which crashes soon after logging the received *verdict*. P_k, as in *scenario* 2, remains crashed for a long time.

$\mathcal{R}$ must now lead all operative P_i, $i \neq k$, to decide on *commit* without waiting for P_k or C to recover.

We observe that the execution environments of scenarios 2 and 3 are identical for all operative P_i, $i \neq k$: both C and P_k remain crashed until all P_i decide on *verdict*; secondly, there is no interaction between P_k and C in **Scenario 2** after C broadcast *cast-vote* and P_i cannot deduce any of the pre-crash interactions between P_k and C in **Scenario 3** until one of the crashed ones recovers. Thus, $\mathcal{R}$ is expected to make all operative P_i decide differently in identical execution environments. Such an $\mathcal{R}$ cannot be designed and hence 2PC blocking is inevitable.

Remarks. As per Skeen [83], the root causes for the inevitability of 2PC blocking are two-fold: both terminal states, c and a, are one-step reachable from W_2 as can be seen in Fig 7.2, and (ii) it is possible to have an operative P_i waiting in W_2 and a crashed P_k either in a (see scenario 2) or in c (see scenario 3). In Skeen's terminology, (ii) is referred to as the terminal states, c and a, being in the *concurrency* set of W_2. Designing $\mathcal{R}$ involves modifying 2PC itself and introducing new pre-terminal '*buffer*' states so that both terminal states are not in the concurrency set of W_2. This 2PC modification leads to 3 phase commit and details are in [83].

7.5 Non-Blocking with Blockchain

7.5.1 Approach

We can observe that if C *were* never to crash *during* 2PC execution, then blocking cannot happen. We build on this observation by having C initiate a transaction by delegating work to all P_i and then entrust the 2PC coordination responsibilities to a blockchain infrastructure (BC, for short) which, being a replicated state machine,

must coordinate 2PC execution in a crash-free manner. To accomplish this, several aspects of BC will be made use of, and they are listed below.

Event ordering. Events directed at a BC are also called *transactions*.BC puts a total order on these events and records them in that order; event recording is immutable and recorded events are permanently visible to all concerned parties. Event ordering in BC can also be used to ensure *exactly once* execution of an action, say, A when multiple sources, e.g., processes in Π, can request A's execution: BC can be programmed (see smart contract below) to accept only the first **Smart Contract**. See Chapter 2 Section 2.4.1 **Ethereum [94]**. See Chapter 2 Section 2.4. **Wall Clock**. Ordered transactions are first arranged in blocks of fixed size which are then arranged in BC in the in-creasing order of block timestamps. Assuming that transactions are being continually submitted to BC, the increasing timestamps of the blocks being added constitute a publicly-visible, real-time *wall-clock* (possibly with irregular ticks); processes of Π can use it as a common time-service.

7.5.2 Synchronous Blockchain

Similar to definitions of ω and δ, let β be the *block construction* bound on the delay that can elapse between the instant when a user process U launches a valid (blockchain) transaction TX_U and the instant when a block containing TX_U is (irreversibly) added in BC; let α be the *awareness* bound on the delay that can elapse between the instant when TX_U enters BC irreversibly and the instant when any interested party gets aware of TX_U in BC. A BC infrastructure (together with miner/consensus nodes) is said to be [37] *synchronous* if it supports reliable estimation of bounds β and α; otherwise, it is said to be *asynchronous*.

The assumption of a synchronous BC implies that several requirements have been met: a valid transaction submitted to BC is never lost but is always considered for entry into the BC in a timely manner, a party interested in a given TX_U is periodically scanning BC, etc. This is just like the validity of δ bound requiring that no message be lost, but every message be queued, transmitted, received and delivered - all in a timely manner.

7.5.3 2PC with Synchronous Blockchain

We explain here (i) how C hands over the coordination responsibilities for 2PC execution to the BC infrastructure and, (ii) how P_i interacts with BC to execute 2PC in two phases. Informally, P_i uses Phase 1 to register its *vote* in BC and Phase 2 to

receive the *verdict*, very similar to the traditional 2PC execution. We also assume that the cluster hosting database processes Π is synchronous as well. We do not, however, require processes of Π to detect each other's crash directly (e.g., by operating a failure detector). This is also the case in the traditional 2PC version.

7.5.3.1 Protocol 1

As in traditional 2PC, C disseminates the work to each $P_i \in \Pi$; it then hands over the responsibilities to the BC infrastructure by launching a (BC) transaction TX_C that sets up the 2PC coordination smart contract in BC with initial *state* = *VOTING*. (Smart contract code is explained in § 7.5.4.) The role of C ends with launching TX_C. Note that C may crash after work dissemination and before launching TX_C; in this case, all operative P_i must detect this and end up deciding *abort* as in traditional 2PC execution.

When P_i receives work from C, it computes T_i as the local time when a duration that is maximum of $\{\omega,\ \delta + \beta + \alpha\}$, would elapse after the receipt of the work. T_i is the earliest local time when P_i can complete its work *and* become aware of TX_C being added to BC, if C had launched TX_C.

Thus, if TX_C does not appear in BC until a block with timestamp ¿ T_i is added, i.e., until BC *wall-clock* exceeds T_i, then, by synchrony assumptions, P_i can deduce that C crashed without launching TX_C; it can subsequently abort as shown by the state transition from W_1 to a in Figure 7.3, where WC denotes the BC *wall-clock*. The transitions from state WG in Fig 7.3 are identical to those shown in Fig 7.2. They have here become off-chain activities [95].

If a P_i that completes its work ($WG \rightarrow W_1$ in Figure 7.3), gets aware of TX_C by local time T_i, it logs locally $V_i = 1$ (as in Phase 1 of Fig 7.1) registers its vote by launching TX_i to BC. When TX_i is accepted in BC, it invokes $VOTER$ function of the smart contract with $V_i = 1$ as input. (State of P_i now transits from W_1 to W_2 in Figure 7.3).

Let $TX_C.BlkTime$ be the timestamp of the block containing TX_C. Any operative P_i gets aware of TX_C no later than $WC = TX_C.BlkTime + \alpha$ and its TX_i, launched in response, would be added to BC by $WC \leq TX_C.BlkTime + \alpha + \beta$. (Note: α and β are upper bounds and actual delays can be smaller than them.)

If all P_i vote $V_i = 1$, then the smart contract would compute *verdict* = *commit* and display *state* = *COMMIT* in BC. (Details in § 4.5.) All P_i observe this *state* by $WC \leq TX_C.BlkTime + 2\alpha + \beta$.

Let $\Delta = 2\alpha + \beta$. When WC exceeds $TX_C.BlkTime + \Delta$, if an operative P_i that sent TX_i cannot see $state = COMMIT$ in BC, then some P_k did not launch TX_k. In that case, P_i can safely decide *verdict = abort*. However, our description here assumes that P_i decides *verdict = commit or abort only* in response to what is being indicated in BC, to be consistent with the traditional 2PC description.

When $WC > TX_C.BlkTime + \Delta$ and $state \neq COMMIT$, P_i launches TX_{V_i} to invoke $VERDICT$ function of the smart contract so that *verdict* is computed and displayed in BC. In Figure 7.3, P_i does $W_2 \rightarrow W_3$ after launching TX_{V_i} and then to $W_3 \rightarrow a$ when BC indicates $state = ABORT$. If several TX_V were launched, only one will be effective in executing $VERDICT$ (like A in § 7.5.1).

7.5.3.2 Protocol 2

Coordinator C: C disseminates the work to each $P_i \in \Pi$ and, immediately after that dissemination, it enters Phase 1 to hand over the coordination to BC infrastructure. On entering Phase 1, C launches a blockchain transaction TX_C that sets up the 2PC coordination smart contract in BC with initial $state = VOTING$.

Phase 1 for C ends with the launch of TX_C, and there is no Phase 2. Another significant difference from the traditional 2PC is that C does not wait on any timeout between disseminating its work to Π and entering Phase 1. Note that C may crash during work dissemination or after dissemination and before launching TX_C. Though Subsection 7.5.4 is devoted to explaining the smart contract in detail, the roles of two of its functions are briefly explained here for ease of understanding: function $VOTER$ enables P_i to enter its vote in BC and also computes the *verdict* once all $P_i \in \Pi$ have voted, and function $VERDICT$ allows a P_i to explicitly request for the *verdict* to be computed. Moreover, once the smart contract computes the *verdict*, it changes the initial *state* to display the computed *verdict*, i.e., to $COMMIT$ or $ABORT$.

Get-Work by P_i: When P_i receives work from C, it records its current local clock time as T_i and enters the 'working' state WG (see Figure 7.3). If C has indeed launched TX_C, then TX_C must enter BC no later than the local time $T_i + \delta + \beta$ and P_i must observe TX_C in BC no later than its local time $T_i + \delta + \beta + \alpha$.

If P_i cannot complete the work due to serializability constraints, it unilaterally decides on *abort* and terminates the execution. This is shown by the state transition from WG to a in Figure 7.3.

If, on the other hand, P_i completes the work from C, it enters Phase 1 by transiting from WG to the first *wait* state W_1 in Fig 7.3.

Phase 1 by P_i: P_i starts Phase 1 by looking for TX_C in BC. If it does not observe TX_C in BC until its clock has exceeded $T_i + \alpha + \beta + \delta$, it deduces that C crashed before launching TX_C and subsequently *aborts* as shown by the transition from W_1 to a in Figure 7.3. P_i awaiting TX_C to appear in BC is similar to its waiting for *cast_vote* in Figure 7.1. Also, the transitions from state WG in Fig 7.3 are identical to the traditional 2PC execution shown in Fig 7.2. (Transitions from WG are also called 'off-chain' activities [95].)

If P_i gets aware of TX_C by local time $T_i + \alpha + \beta + \delta$, it logs T_i first, followed by logging of $V_i = 1$ (the latter as in Phase 1 of Fig 7.1). The logging order of T_i and then V_i is important for post-recovery execution by which P_i can decide if it crashed undecided after this point in 2PC execution. (Description in § 7.5.3.2.)

After logging T_i and V_i, P_i launches transaction TX_i with its vote $V_i = 1$. It then enters Phase 2, with its state transiting from W_1 to a second *wait* state W_2 in Figure 7.3. Note that P_i launching its TX_i must happen by its clock time $T_i + max\{\alpha + \beta + \delta, \omega\}$, where $max\{\alpha + \beta + \delta, \omega\}$ is the larger of $(\alpha + \beta + \delta)$ and ω: P_i must observe TX_C in BC by clock time $T_i + \alpha + \beta + \delta$ and complete its work by $T_i + \omega$.

Phase 2 by P_i: When TX_i is accepted in BC, it invokes $VOTER$ function of the smart contract with $V_i = 1$ as input. Moreover, if all $P_j \in \Pi$ launch TX_j, i.e., vote $V_j = 1$, then the $VOTER$ function would compute *verdict = commit* and display *state = COMMIT* when the last $V = 1$ is counted; otherwise, the *state* of BC will remain at the initial *state = VOTING*. (Details in § 7.5.4.)

Let $\Delta = max\{\alpha + \beta + \delta, \omega\} + \alpha + \beta + \delta$. P_i in Phase 2 waits for BC state to change to *state = COMMIT* until its clock time $T_i + \Delta$. If P_i observes BC *state = COMMIT* by then, it decides *verdict = commit*.

If P_i, on the other hand, still observes *state = VOTING* until its clock exceeds $T_i + \Delta$, this means that some P_k, $k \neq i$, did not launch TX_k. So, *verdict* must be *abort*. Though P_i can now safely decide *abort*, our description here assumes that P_i decides on *verdict = abort* in response to such an indication from BC, just as in the traditional 2PC description where a P_i that voted $V_i = 1$ decides on *abort* by receiving *verdict* from C.

When BC *state = VOTING* and clock exceeds $T_i + \Delta$, P_i launches TX_{V_i} to invoke $VERDICT$ function of the smart contract so that *verdict* is computed in BC and displayed. In Figure 7.3, P_i does $W_2 \rightarrow W_3$ after launching TX_{V_i}, waits in W_3 until BC indicates *state = ABORT* and then decides *verdict = abort*.

Waiting by P_i in W_3 must terminate as BC is reliable. It is likely that several other P_j launch their own TX_{V_j} around about the same time when P_i launches TX_{V_i}. If so, only one will be effective in executing $VERDICT$ (like A in § 7.5.1). Once BC indicates $state = ABORT$, P_i decides on *abort* and terminates the execution (W_3 to a in Figure 7.3).

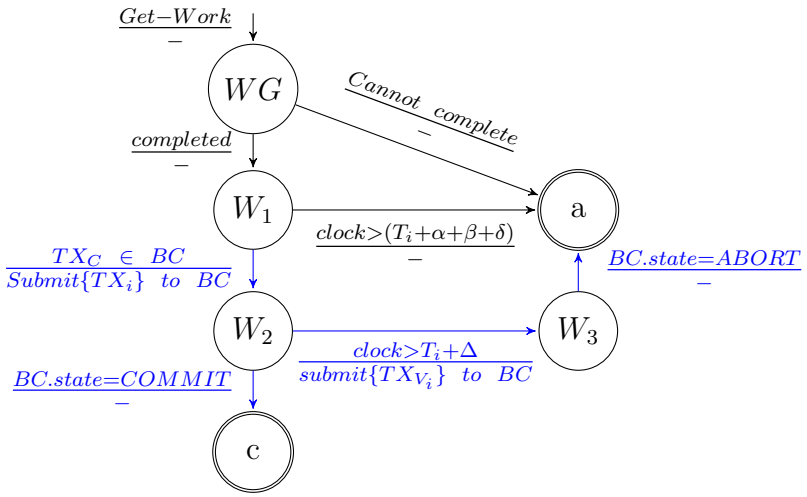

Figure 7.3: State Diagram for 2PC with Blockchain.

Post-Recovery Execution: It is possible that some $P_k \in \Pi$ crashes during the protocol execution. When it recovers, there are two possible cases: log of P_k has or does not have entry $V_k = 1$.

Absence of entry $V_k = 1$ means that TX_k was never launched and any work done by P_k has been erased from its (volatile) memory during the crash. So, the recovered P_k does not know about the database transaction that triggered the 2PC execution. P_k could, and hence would, do nothing regarding that database transaction; in other words, P_k indirectly decides on *abort*. Further, any $P_i, i \neq k$, that logged $V_i = 1$ can also decide only on *abort*.

Suppose that the log of P_k has the entry $V_k = 1$. This means that P_k, prior to its crash, must have observed TX_C in BC during its pre-crash execution of Phase 1 and also logged the local time T_k (see § 7.5.3.2). P_k will resume executing 2PC starting from Phase 2 (with its state in W_2) and get the *verdict* from BC.

Since T_k is logged prior to logging $V_k = 1$, the log that contains $V_k = 1$ must have T_k as well. If P_k had crashed after logging T_k but before V_k (hence before launching TX_k), then V_k would not be found in the post crash execution and the entry T_k without a matching V_k is simply deleted.

Note that the post-recovery execution enables P_k to decide even if P_k is the only process in Π to have logged $V_k = 1$ and crashed before launching TX_k, while all others transited from WG to a: the recovered P_k would then launch TX_{V_k} when its clock $> T_k + \Delta$ and BC would subsequently change its *state* from *VOTING* to *ABORT*. Note also that there is no assumption on how long a crashed P_k can take to recover.

7.5.4 Smart Contract Pseudo Code

Figures 7.4 and 7.5 present the pseudo-codes of 2PC coordination and the description here assumes that the contracts are already deployed on the blockchain with unique addresses.

7.5.4.1 Protocol 1 Pseudo Code

The contract has an initial *state* $INIT$, with three parameters $Timeout$ (initialized to zero), an initially empty set A of named participants and an empty set V of voted participants; its functions have the following interfaces: $REQUEST(A, Timeout)$, $VOTER(boolean)$ and $VERDICT()$.

TX_C submitted by C invokes $REQUEST$ function with $(A = \Pi, Timeout = \Delta)$, where $\Delta = 2\alpha + \beta$. This initialization succeeds if C is *asserted* to have ownership rights to invoke this function and the code is in the initial state *INIT* - as indicated in the *Assert* statement. If this assertion succeeds, TX_C is accepted and the *state* of the contract is changed to $VOTING$ which is publicly visible in BC; otherwise, TX_C is ignored. (This is always the case: a TX is rejected if the pre-invocation assertion fails; throughout this description, assertions are assumed to succeed, except for duplicate calls on *VERDICT* function.)

Each P_i in W_1 checks BC for TX_c; when *state* = *VOTING*, $V_i = YES$ is sent by submitting TX_i that invokes *VOTER* function. Upon receiving TX_i, the contract asserts if P_i is legitimate to vote or not. When P_i is legitimate, P_i is recorded to have voted in the set V. If $V = \Pi$, then the contract *state* is changed to *COMMIT*.

After $WC = TX_C.BlkTime + \Delta$, any P_i in W_2 that still finds the *state* = *VOTING*, invokes *VERDICT* function by submitting TX_{V_i}. The invocation succeeds only if (i) $P_i \in \Pi$, (ii) sufficient time of $\Delta = 2\alpha + \beta$ had elapsed since TX_C was added into BC and (iii) *state* = *VOTING*. If it succeeds, it sets *state* = *ABORT*. An attempt to invoke *VERDICT* when *state* = *COMMIT* or *state* = *ABORT*, will not meet (iii) and not succeed.

7.5.4.2 Protocol 2 Pseudo Code

The deployed contract is in the initial *state* $INIT$ and has two set variables: Σ and Σ_V which are the set of participants eligible to vote and the set of those who actually voted, respectively; both the sets are initially empty (when BC *state* = $INIT$). The smart contract has three functions:

- $REQUEST()$ invoked by TX_C to initialize the contract,
- $VOTER()$ invoked by TX_i to register the vote of P_i and to compute *verdict* once all $P_i \in \Pi$ voted, and
- $VERDICT()$ invoked by TX_{V_i} to request the *verdict* to be computed, if not already done.

TX_C submitted by C contains Π and invokes $REQUEST$ function. This invocation succeeds only if C is *asserted* to have ownership rights to invoke this function and the code is in the initial state $INIT$ - as indicated in the *Assert* statement. If this assertion succeeds, TX_C is accepted and the *state* of the contract is changed to $VOTING$ and Σ to Π; otherwise, TX_C is ignored.

Note that it is the feature of any blockchain that a transaction, such as TX_C, is rejected if any of the pre-invocation assertions fail. Throughout this description here, assertions are assumed to succeed, except for those TX_V that seeks to invoke the $VERDICT$ function, not for the first time.

Having observed TX_C in BC, a $P_i \in \Pi$ with vote $V_i = 1$ launches its TX_i. After asserting that *state* = $VOTING$, $V_i = 1$ and $P_i \in \Sigma = \Pi$, the contract records P_i to have voted by adding it in Σ_V. The BC *state* is changed to $COMMIT$ when $\Sigma_V = \Sigma$.

Any P_i in W_2 that finds *state* = $VOTING$ even after its clock has read $T_i + \Delta$, invokes $VERDICT$ function by submitting TX_{V_i}. The invocation succeeds only if $P_i \in \Sigma = \Pi$ and *state* = $VOTING$. If it succeeds, it sets *state* = $ABORT$. An attempt to redundantly invoke $VERDICT$ when *state* = $ABORT$ will not meet the latter condition and not succeed.

7.6 Asynchrony & Impossibilities

When bounds α and β cannot be reliably estimated, BC becomes asynchronous (see Subsection 7.5.2); similarly, when estimates of bounds δ and ω are not guaranteed to hold, the cluster hosting Π becomes asynchronous (Subsection 7.3.1).

INIT:	Set $state := INIT$; $A := [0x000 \ldots 0x000]$ $Timeout := 0$; $V := [0x000 \ldots 0x000]$
REQUEST:()	Upon C submitting $Tx_C(\Pi, \Delta)$: Assert ($state == INIT$ and $msg.sender == C$) Set $A := \Pi$; Set $Timeout := \Delta$; $state := VOTING$.
VOTER:()	Upon P_i submitting Tx_i ($Vote$): Assert ($state == VOTING$ and $msg.sender == P_i \in \Pi$) Assert ($P_i \notin V$), Assert ($Vote == YES$) Set $V := V \cup \{P_i\}$; if $(V == \Pi)$ then $\{state := COMMIT;\}$
VERDICT:()	Upon P_i submitting TX_{V_i}: Assert ($state == VOTING$ and $msg.sender == P_i \in \Pi$) Assert (block.timestamp > Tx_C.block.timestamp + Δ) Set $state := ABORT$;

Figure 7.4: Smart Contract pseudo-code for 2PC coordination protocol 1.

***INIT*:**	Set $state := INIT$; $\Sigma := [0x000, \ldots, 0x000]$; $\Sigma_V := \Sigma$;
***REQUEST()*:**	Upon C submitting $TX_C(\Pi)$: Assert ($state == INIT$ and *credentials of* C) Set $\Sigma := \Pi$; Set $state := VOTING$;
***VOTER()*:**	Upon P_i submitting TX_i ($Vote$): Assert ($state == VOTING$ and $P_i \in \Sigma$); Assert ($P_i \notin \Sigma_V$); Assert ($Vote == 1$); Set $\Sigma_V := \Sigma_V \cup \{P_i\}$; if $(\Sigma_V == \Sigma)$ then Set $state := COMMIT$;
***VERDICT()*:**	Upon P_i submitting TX_{V_i}: Assert ($state == VOTING$ and $P_i \in \Sigma$); Set $state := ABORT$;

Figure 7.5: Pseudo-code for 2PC coordination smart contract protocol 2.

Note that a public BC can be asynchronous even if the underlying distributed system is synchronous. For example, if miners, at the time of TX_C launch, also encounter several other transactions that are more financially attractive to work on compared to TX_C, then TX_C could take longer to enter BC, if at all, than any β estimated in more favourable environments [92]. Similarly, BC can be synchronous while the underlying distributed system is asynchronous. Thus, from the synchrony requirements perspective, our system is made up of two distinct sub-systems: BC and database cluster. This leads to three pertinent questions: can we have a non-blocking 2PC in which the coordinator C offloads its coordinating responsibilities to a BC, when

1. the BC being used is synchronous, and the cluster hosting Π is asynchronous?
2. the BC is asynchronous, and the cluster is synchronous?
3. both the BC and the cluster are asynchronous?

We formally answer these open questions here and show that non-blocking 2PC is *not* possible in all cases. It turns out that the perfect failure detection capability within Π when the cluster is synchronous, is not enough to construct a non-blocking 2PC if BC is asynchronous (see [38] for more details).

7.6.1 Implications of Synchrony Violations

A closer look at the impossibility proofs reveals that asynchrony in BC or in the cluster prevents only *commit-validity* from being guaranteed i.e., *abort* could be decided when all processes of Π are operative and vote *yes*. This is also confirmed by the correctness arguments in [38] which show that our 2PC protocol operating with BC solve the atomic commit problem when both BC and cluster are synchronous. More precisely, these arguments indicate that if (i) C crashes without launching TX_C, (ii) some P_k crashes, or (iii) some P_i votes *no*, the other three requirements are guaranteed to be met even when the delay bound estimates are violated: arguments for *termination* [38] and *abort-validity* [38] do not refer to synchrony assumptions at all; moreover, in cases (i) - (iii) above, *verdict* = *abort* is the correct outcome and *verdict* = *commit* cannot ever be reached. So, the *agreement* is also met. In summary, synchrony is needed only to guarantee *commit-validity*. Thus, when a bound estimate $b \in \{\alpha, \beta, \delta, \omega\}$ is violated, the only requirement that risks being compromised is *commit-validity*, leading to unwarranted *aborts* of database transactions. Violations of b can occur

due to transient surges in computational loads or network traffic or the traffic and/or loads having increased since the bound estimates were last computed.

At any given time, let b_a be the *actual* prevailing value for an estimate $b \in \{\alpha, \beta, \delta, \omega\}$. Synchrony is violated if $b < b_a$ for *any* b. This does not necessarily mean that the two timeouts used in the protocol would be violated. (Recall that $(\alpha + \beta + \delta)$ is the Phase 1 timeout defined in § 7.5.3.2 for deciding whether TX_C would ever appear in BC, and $\Delta = max\{(\alpha + \beta + \delta), \omega\} + (\alpha + \beta + \delta)$ is the Phase 2 timeout defined in § 7.5.3.2 before launching TX_V.)

For example, if only $\alpha < \alpha_a$ and $b > b_a$ for every other b, we can still have: $\alpha + \beta + \delta \geq \alpha_a + \beta_a + \delta_a$ and $\Delta \geq max\{(\alpha_a + \beta_a + \delta_a), \omega_a\} + (\alpha_a + \beta_a + \delta_a)$.

Denoting $\Delta_a = max\{(\alpha_a + \beta_a + \delta_a), \omega_a\} + (\alpha_a + \beta_a + \delta_a)$,

let us define:

$$m_1 = \frac{\alpha + \beta + \delta}{\alpha_a + \beta_a + \delta_a} \text{ and } m_2 = \frac{\Delta}{\Delta_a}. \tag{7.1}$$

Only when $m_1 < 1$ or $m_2 < 1$, Phase 1 or Phase 2 timeouts are at risk of becoming 'too small' respectively, leading to the possibility of a transaction being unnecessarily aborted and the *commit-validity* not being upheld. As noted, $(m_1 \geq 1 \wedge m_2 \geq 1)$ can still hold when only *some* bound estimates suffer minor violations.

Using our protocol implementation described next, we evaluate the likelihood of unwarranted *abort* occurrences when Phase 1 and Phase 2 timeouts are made small by varying amounts.

7.7 Implementation and Evaluation

We implemented the 2PC-Blockchain contracts from Figures 7.4 and 7.5 in Solidity 0.40.11 [48] and tested their operations on the Ethereum private and test networks [41], using Ethereum Wallet and Ethereum Mist [51]. Four different machines are used: (a) a MacBook Pro with a 2.8 GHz Intel i5 CPU and 8 GB RAM, (b) three desktop PCs with a 3.20 GHz Intel i7 CPU and 8 GB RAM running on Windows 10. The MacBook is the coordinator C and the three desktop PCs constitute the 'cluster' hosting P_1, P_2 and P_3. Each PC is connected to the Ethereum network as a full node, thus having a full copy of the blockchain stored within it. The PCs do not play the role of miners themselves and operate as non-mining database hosts connected to the blockchain. They are also connected to each other and to switches by a standard switched Ethernet local area network, which connects through standard TCP/IP with

the Ethereum network. Smart contracts (see Figures 7.4 and 7.5) are also registered with the Ethereum network.

7.7.1 Delay Bound Estimation

In all our experiments, the database transaction is kept null because our main objective is to assess the cost and performance of coordination activities within and around the blockchain. Consequently, a 'get-work' message from C contains no work for P_i but simply initiates the latter to execute 2PC which votes *yes* or *no* as per the purpose of a given experiment; so, the bound estimate $\omega = 0$. Other bounds α, β and δ are established as follows.

The awareness delay (bounded by α) is calculated by taking the difference between the confirmation time of a given transaction of interest (such as TX_C or TX_i) entering a block in BC and the time of receiving this block by each P_i. The confirmation time is obtained from the Ethereum wallet, which shows the time that the block was added. The time stamps at the three P_i nodes give us three data points, and the maximum of these three results is taken as one data point for estimating α. At the end of 30 experiments in which only C launched TX_C, the maximum of the 30 data points obtained is taken as α.

The block entry delay (bounded by β) is calculated as the difference between the time stamp given to TX_C at the coordinator node when TX_C is sent, and the confirmation time of the block that contains TX_C within the blockchain. Similar to α, we take the maximum of all data points obtained as β.

To obtain α and β, each individual experiment consists of C submitting one single transaction TX_C and ends once we have collected all the data points. Each experiment takes several minutes, as we will see, and is repeated 30 times.

To measure data points for transmission delays (bounded by δ), no P_i needs to interact with the blockchain. We measure these data points by letting C send a $1KB$ Ethernet packet to each processor P_i, which then sends it back to C. We take the round trip time and halve it to get one-way delays. The maximum of all data points collected is taken as δ: we collected 30 round trip times for each P_i, so δ is the maximum over 90 one-way delay estimates.

The results for α, β and δ are shown in Figures 7.6, 7.7 and 7.8. In all three Figures, the x-axis gives the experiment number (from 1 to 30), and the y-axis gives the point estimate of α, β and δ (the max of the results in the three nodes, as explained above).

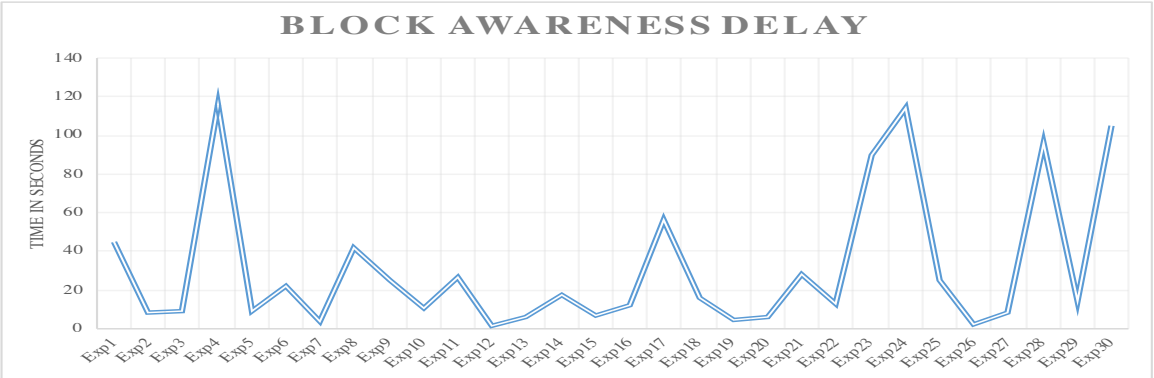

Figure 7.6: Block awareness delay.

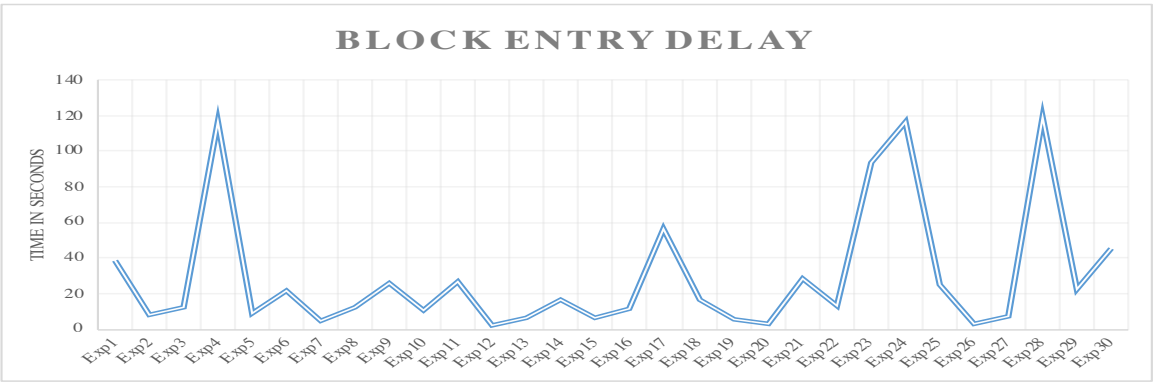

Figure 7.7: Block entry delay.

In estimating α, all experiments return values within the two-minute range. The highest observed value is for experiment 4, at 115.734 seconds. Figure 7.6 shows only the maximum of the values for the three P_i, and we note that the difference between the three obtained values in each of the 30 experiments is minimal, less than one second. For information, the average and the median of the block awareness delays depicted in Fig 7.6 are 30.461 and 13.455 seconds, respectively.

In the experiments for β, the maximum is found in experiment 28, at a value of 118.800 seconds. Note that for some experiments the transaction finds its way into a block in a matter of seconds, the minimum observed delay was 2.355 seconds. The block entry delay is influenced by factors such as the transaction's gas price which in turn influences miners' decisions of which transactions to include into the blocks they work on.

Figure 7.8 shows the results of our experiments for estimating δ. They range from 1.590 seconds to 5.790 seconds.

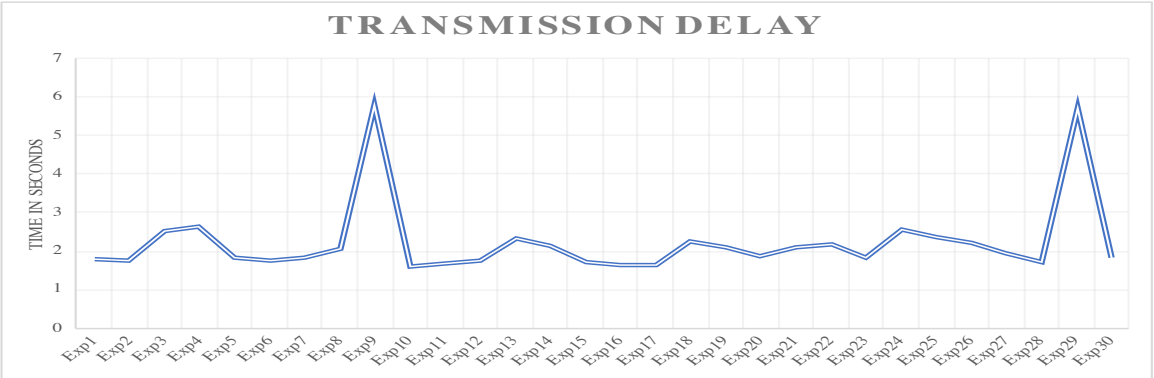

Figure 7.8: Transmission delay.

7.7.2 Cost of 2PC Coordination

As noted in Subsection 7.5.1, the initiator of a blockchain transaction that involves executing one or more functions of a smart contract ought to pay the miner in the cryptocurrency *ether* that is commonly abbreviated as eth. The payment is in proportion to the amount of 'gas' (often written as GAS) consumed by the executions of functions a transaction invokes.

Furthermore, a transaction initiator can quote in the transaction the gas price they are willing to pay for executing the smart contract functions. A higher gas price quoted can act as an incentive to miners in giving preferential treatment over those that quote a lower gas price. In our experiments, the gas price quoted was the lowest possible; e.g., the Coordinator quotes the gas price of 0.001 eth/million for executing the *REQUEST* function. By quoting only the lowest gas price, the cost in eth we report here would indicate the lower bound.

When a smart contract function involves repetitive executions conditional on Boolean statements (e.g., a *while* loop), the gas cost can vary with the inputs supplied at invocations. As we can see from Figure 7.5, the 2PC coordination code does not involve aspects that lead to input-dependent execution cost variations, except when the last $P_i \in \Pi$ casts its vote, the boolean $\Sigma_V == \Sigma$ (which is checked on every invocation of *VOTER()*) comes true and 'Set *state* = *COMMIT*' is additionally executed. This additional execution of a simple 'Set' statement does not incur any extra gas and it is confirmed in all our experiments.

The amount of gas that a miner uses when executing a given contract function is calculated by the Ethereum virtual machine itself and is displayed in the Ethereum wallet at the initiator end. So, it is safer to assume that the reports on the amount of gas expended for executing a given contract function are quite reliable. Table 7.1

provides the cost of executing each of three smart contract functions: *REQUEST()*, *VOTER()* and *VERDICT()*. As per the prevailing exchange rates for eth, the cost is in the order of few US cents or British pence.

Transaction	Reason	GAS Used	Cost in eth
TX_c	By C to request voting	232736	0.000232736
TX_i	By P_i to vote	84625	0.000084625
TX_v	By P_i to seek verdict	55102	0.000055102

Table 7.1: Cost of executing 2PC-Blockchain contracts.

Table 7.2 presents the total cost for 2PC coordination in four possible voting scenarios when the number of P_i in Π is three.

Scenarios	**Gas Used**	**Cost in ETH**
Three vote no	**232736**	**0.000232736**
Two vote no	**372463**	**0.000372463**
One votes no	**457088**	**0.000457088**
All vote yes	**486611**	**0.000486611**

Table 7.2: Total Cost in Various Voting Scenarios.

When a P_i votes *no*, it knows that the *verdict* = *abort* and terminates. Thus, when all three P_i vote *no*, none will launch TX_i or TX_{V_i}. So, only *REQUEST()* function is executed and its gas price the total cost as shown in the row 1 of Table 7.2.

In considering the remaining rows of Table 7.2, let us assume that neither a process crash nor any violation of the bound estimates occurs during 2PC execution. If n' processes, $n' = 1$ or 2, vote *no*, $(3 - n')$ processes launch TX_i and, at the expiry of Δ timeout, also TX_{V_i} of which only one will end up invoking *VERDICT()* function. Thus the total cost incurred is: the cost of row 1 + $(3 - n')$ × the cost of executing *VOTER()* function once + the gas cost of executing *VERDICT()* function once.

When all three processes vote *yes*, none will launch TX_{V_i} and the total cost is: the cost of row 1 + 3× the cost of executing *VOTER()* function once. Generalizing, when y processes, $0 \leq y \leq |\Pi|$, vote *yes*, the total gas cost for 2PC coordination is: gas cost of executing *REQUEST()* function once + y × the gas cost of executing *VOTER()* function once + c × the gas cost of executing *VERDICT()* function once, where $c = 0$ if $y = 0 \vee y = |\Pi|$, and $c = 1$ otherwise (i.e., $0 < y < |\Pi|$).

7.7.3 2PC Execution Latencies

2PC execution latency for an operative P_i can be defined as the duration that can elapse from the moment when P_i receives 'work' from coordinator C until the moment when P_i decides either to *commit* or *abort* the transaction. Let the moments of P_i receiving work and deciding to be denoted as T_i and $T_i + E_i$ respectively and be observed as per P_i's local clock. Thus, E_i is the 2PC execution latency for P_i. We will discuss E_i by first estimating the maximum value it can (theoretically) take and then reporting the actual maximum it took in our experiments, along with an explanation for any wide discrepancy between the two. Our estimation of latency bound will assume that the delay bound estimates used were *conservatively* arrived at by assigning them to the largest data points observed (as described in § 7.7.1) and hence are *safe*, i.e., never violated.

7.7.3.1 Estimated Latency Bound

All possible execution scenarios need to be considered before arriving at the upper bound for E_i. To start with, let us consider the simplest case where P_i takes the transition $WG \rightarrow a$ (see Figure 7.3); here, E_i cannot exceed ω.

Alternatively, P_i can vote *yes* instead of doing $WG \rightarrow a$. In this execution scenario, two cases need to be considered: TX_C does not or does enter BC. When TX_C does not enter BC due to C crashing subsequent to disseminating the 'work', P_i will affirm the absence of TX_C at the expiry of Phase 1 timeout and decide *abort*; so, E_i = Phase 1 timeout = $\alpha + \beta + \delta$. In the second case where C does not crash and TX_C does enter BC, E_i will depend on the number, y, of processes in Π that vote *yes*.

Let $y = |\Pi|$. Measuring time as per P_i's clock, we note that P_i would commence two parallel activities at T_i: doing the work given to it and looking for TX_C to appear in BC. The former must complete by $T_i + \omega$ and TX_C in BC would be known to P_i by T_i + Phase 1 timeout = $T_i + \delta + \beta + \alpha$, at the latest. Thus, at or before $T_i + max\{\omega, (\alpha + \beta + \delta)\}$, P_i must launch its TX_i and all other P_j must do so by P_i's clock time $T_i + max\{\omega, (\alpha + \beta + \delta)\} + \delta$. Thus, the *verdict* computed at BC would be known to P_i no later than its clock time $T_i + max\{\omega, (\alpha + \beta + \delta)\} + (\alpha + \beta + \delta)$. So, $E_i \leq max\{\omega, (\alpha + \beta + \delta)\} + (\alpha + \beta + \delta)$. Typically, ω is very small compared to $(\alpha + \beta + \delta)$ and thus $E_i \leq 2(\alpha + \beta + \delta)$ when $y = |\Pi|$

Let $y < |\Pi|$. (Since P_i votes *yes*, $y > 0$). P_i would launch TX_{V_i} at its clock time $T_i + \Delta$ and would observe BC *state=ABORT* no later than its clock time $T_i + \Delta + \beta + \alpha$.

Thus, $E_i \leq \Delta + \alpha + \beta$. Given that $\Delta = max\{(\alpha + \beta + \delta), \omega\} + (\alpha + \beta + \delta)$ (defined in § 7.5.3.2), $E_i \leq 2(\alpha + \beta + \delta) + (\alpha + \beta)$ when ω is considered small compared to $(\alpha + \beta + \delta)$.

Summarizing, E_i cannot exceed $\Delta + (\alpha + \beta) = 2(\alpha + \beta + \delta) + (\alpha + \beta)$ for an operative P_i in any possible combination of crashing and voting scenarios. Substituting the delay bound estimates, the (upper) bound for E_i is $2(115.734 + 118.800 + 5.790) + (115.734 + 118.800) = 715.182$ seconds, i.e., 11 minutes and 55.182 seconds.

Finally, let us also estimate, for the sake of comparison, the bound for E_i when 2PC is executed without BC (as described in § 7.4). If P_i suffers blocking due to crash of C, E_i can be arbitrarily long as P_i cannot decide until C recovers. When C does not crash, it turns out that $E_i \leq \omega + 4\delta$: having received 'work' from C at its clock time T_i, P_i can receive the broadcast *cast_vote* at or before $T_i + \omega + \delta$; C broadcasts the *verdict* after a 2δ timeout expires following its broadcasting of *cast_vote*; P_i must decide by $T_i + \omega + \delta + 2\delta + \delta$ if it voted *yes*. Thus, using BC to eliminate 2PC blocking results in a performance slow down when C does not crash and the slowdown is bounded by $3(\alpha + \beta) - (\omega + 2\delta) \approx 3(\alpha + \beta) = 703.611$ seconds. Such a large slowdown should be expected, given the features of public blockchains as discussed in Subsection 7.3.2 and also in [92], and the need to use safe delay bound estimates so that both BC and the cluster remain synchronous, i.e., synchrony violations do not occur.

7.7.3.2 Observed Latencies

We carried out 200 2PC executions using our implementation involving the Ethereum blockchain. We disallowed crashes and ensured that the 'work' given by C is trivial to execute and all $P_i, 1 \leq i \leq 3$, always vote *yes*, i.e. $y = |\Pi|$. Note that each execution must result in all three processes deciding *commit*; otherwise, it would mean that Phase 1 or Phase 2 timeout became 'too small' in the prevailing execution environment and expired prematurely. In all 200 experiments, *commit* was indeed the decision.

In each experiment, P_i recorded the local clock times when it received the work, observed TX_C in BC and decided as T_i, $T_i + D_i$ and $T_i + E_i$ respectively. D_i and $(E_i - D_i)$ represent the latency for P_i to execute only Phase 1 and Phase 2, respectively.

Table 7.3 summarizes the minimum, maximum and average of the 200 latency values experienced by individual processes. We observe that the largest E_i is experienced by P_2 and stands at 4 minutes and 36.880 seconds. The corresponding upper bound estimate (when $y = |\Pi|$) is $2\ (\alpha + \beta + \delta) = 2 \times 240.324 = 480.648$ seconds or 8 minutes 0.648 seconds, which is about twice the maximum observed. In addition

	D1	D2	D3	E1	E2	E3	E1-D1	E2-D2	E3-D3
Min	00:09.421	00:42.872	00:20.412	00:08.332	00:42.335	00:20.203	00:07.964	00:41.904	00:20.328
Max	02:56.276	04:37.990	02:40.783	02:55.178	04:36.880	02:40.806	02:55.112	04:36.842	02:40.742
Avg	00:30.336	01:19.295	00:48.959	00:36.843	01:26.309	00:49.466	00:36.728	01:26.217	00:49.489

Table 7.3: Minimum (Min), Maximum (Max) and Average (Avg) Latency in Minutes (Mn) and Seconds (Ss) expressed as Mn:Ss.

to this large discrepancy between the estimated and observed bounds for E_i (when $y = |\Pi|$), we also observe large differences between the maximum and the average (or minimum) latency in each column. The explanation for this lies in the shape of graphs in Figures 7.6, 7.7 and 7.8: the largest data point ends up deciding the estimate $b \in \{\alpha, \beta, \delta\}$ and is substantially larger than most frequently occurring data points. For example, as noted earlier, the largest awareness delay observed in Fig 7.6 is 115.734 seconds which determines α; $0.2\alpha = 23.147$ is still larger than the average awareness delay observed (13.455 seconds) and $0.4\alpha = 46.294 > 30.461$, the median. Similarly, in the experiments for β in Fig 7.7, the peak value of 118.800 seconds was observed in experiment 28 and was adopted as β. Only in two other experiments, the block entry delay came close to β, and in the rest, it was below 50% of β, with the minimum observed delay being 2.355 seconds.

7.7.4 Impact of Synchrony Violations on Commit-Validity

We observed in § 7.7.3.1 that E_i is the largest when C does not crash and $y < |\Pi|$: $E_i = \Delta + \alpha + \beta$. This is because all P_i that vote *yes* are forced to wait until $T_i + \Delta$ before they could launch TX_{V_i} which then causes BC to compute and display the *verdict*. Any attempt to reduce E_i in this worst case and also in other cases, and thus to speed up 2PC execution in general, requires using smaller values for Δ, α and β; this calls for less conservative estimation of α, β and δ as Δ is a function of these delay bound estimates. Deliberately under-estimating delay bounds, however, tends to increase the scope for synchrony violations. We also noted in § 7.6.1 that synchrony violations risk only the *commit-validity* requirement not being met, leading to unwarranted *aborts*. We will here evaluate the probability of *commit-validity* being met as synchrony violations are permitted to occur due to delay bounds being deliberately under-estimated.

Recall that when ω is considered small compared to $(\alpha + \beta + \delta)$, the Phase 2 timeout $\Delta = max\{(\alpha+\beta+\delta), \omega\} + (\alpha+\beta+\delta)$ (defined in § 7.5.3.2) simply becomes $2(\alpha + \beta + \delta)$; Phase 1 timeout (see § 7.5.3.2), $(\alpha + \beta + \delta)$, becomes $\Delta/2$.

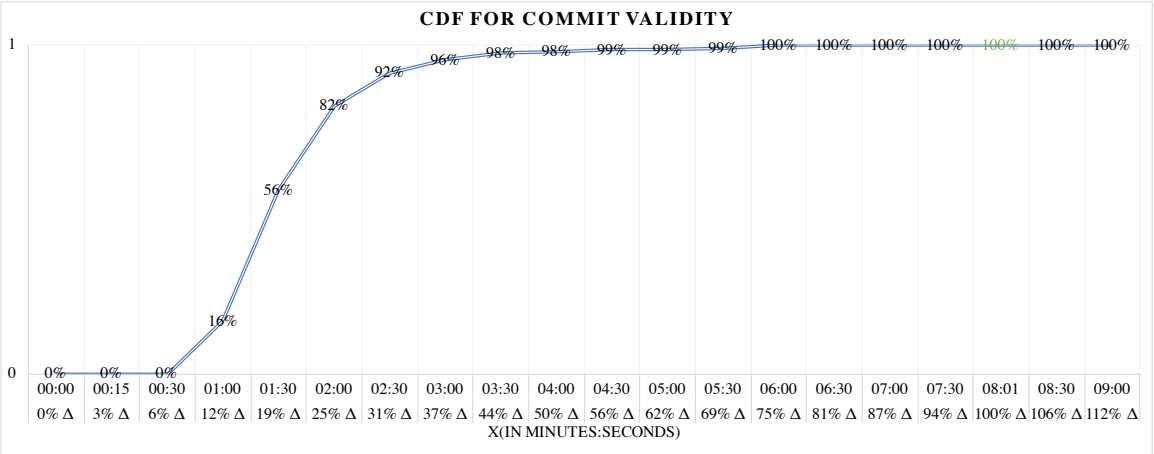

Figure 7.9: Probability for commit-validity.

Suppose that each bound estimate $b \in \{\alpha, \beta, \delta\}$ is chosen not as the largest data point observed (as in conservative estimations) but as m times the largest data point, where m is a small positive real number. When $0 < m < 1$, Phase 1 and Phase 2 timeouts drop to $m\Delta/2$ and $m\Delta$ respectively and execution latency is reduced; in our experiments, *commit-validity* is upheld in an execution only if $D_i < m\Delta/2$ and $E_i < m\Delta$ for all $P_i \in \Pi$. For any given $X = m\Delta$, the probability of *commit-validity* being upheld is the fraction of 200 experiments in which $D_i < X/2$ and $E_i < X$ for all $P_i \in \Pi$.

Figure 7.9 depicts the cumulative distributive function for *commit-validity* for $X = m\Delta$ with m ranging from 0.03 to 1.12. (Absolute values of X are in the first row of x-axis as Minutes:Seconds.) We observe that when X is as small as 0.25Δ, *commit-validity* is upheld with a probability as high as 82%. What this means here is that choosing $b \in \{\alpha, \beta, \delta\}$ to be 25% of the largest data point observed leads only to 18% of runs suffering unwarranted *aborts* while it can reduce 2PC execution latency by 75%. Further, the *commit-validity* probability rises quickly to 98% for m as small as 0.44 and it becomes 100% for $m \geq 0.75$. The latter indicates that 2PC execution latency can be reduced by 25% without suffering any unwarranted *aborts*. All these observations suggest that (i) small under-estimations of delay bounds may not lead to unwarranted *aborts* at all, and (ii) there is much room for reducing execution latency considerably at the expense of a modest increase in unwarranted *aborts*.

7.8 Conclusion

Common choices to avoid 2PC blocking are to use a decentralized protocol [78, 35] or the (centralized) 3 phase commit. These alternatives extract a larger message cost even in the absence of crashes and do not have the structural simplicity of 2PC. We have shown here that the message cost and implementation difficulties of existing 2PC alternatives can be avoided if the 2PC coordinator C simply offloads coordination responsibilities to a blockchain after disseminating database work to servers. Our proposed protocol maintains the low message overhead and the elegant structure of 2PC: those servers that want to commit look up to the crash-free blockchain for progress (instead of crash-prone C) and launch at most two blockchain transactions (instead of periodically pinging the crashed C until it recovers). The extra cost arises in two forms: miners' fees and latency sacrifice when a public blockchain is used; the former is very small in fiat currencies, but the latter can be substantial, in the order of hundreds of seconds as shown by our experiments involving the Ethereum blockchain. We believe that the performance slowdown will not be so serious if permissioned blockchains had been used and our future work would focus on such an investigation.

Though the blockchain infrastructure maintains the abstraction of a reliable state machine with an immutable audit trail display, such features are not sufficient to *guarantee* non-blocking atomic commit, unless it meets synchrony requirements. This is another important contribution of this chapter, which should be borne in mind when building applications similar to atomic commit using blockchain. For example, eVoting, like atomic commit, can be *guaranteed* to be correct only if the blockchain is synchronous; this aspect is not emphasized but is simply assumed in some blockchain based eVoting systems [73]. Informally, the total number of '*yes*' votes cast are counted in both applications, and the count is displayed in eVoting whereas it is used to decide between *commit* and *abort* in the atomic commit. Since a dishonest participant can seek to undermine the result of eVoting, it is important for an eVoting system to specify timing requirements to distinguish between a 'timely' vote that gets counted and the one that arrives 'too late' and gets ignored. This naturally leads to synchrony requirements for correctness.

We have applied the traditional 'best effort, worst-case' method to estimate delay bounds reliably. We then emulated synchrony violations by deliberately choosing to use smaller values as bound estimates and thereby examined the extent of *commit-validity* violations resulting in unwarranted aborts. We observe the number

of unwarranted aborts occurred to be small even when bound under-estimations are considerable. For example, a uniform reduction of 81% across all bound estimates still upholds *commit-validity* (i.e., zero aborts) in more than 50% of runs ($X = 0.19\Delta$ in Fig 7.9). This is because the peak delays observed during bound estimation are much larger than the average or median delays. So, the 'worst-case' bound estimation offers built-in tolerance for synchrony violations. Its downside, however, is that the protocol takes much longer to terminate. Thus, there is a trade-off between reducing protocol latency and using smaller than 'worst-case' bound estimates, which risks violating *commit-validity*.

Chapter 8

Conclusion and Future Work

This chapter summarizes the research presented in this PhD thesis, and discusses future research in the field, motivating several potential research efforts.

8.1 Summary

In this PhD thesis, we introduced and explored two performance benchmarking techniques at both the smart contract level and the operational code (opcode) level of smart contracts. We also explored performance experiments for systems that bootstrap trust from the blockchain in order to build and apply applications in different domains such as cloud computing and distributed database management systems.

Chapter 3 proposed a benchmarking technique to assess whether the fees miners gained from executing smart contract transactions are proportional to the cost of the CPU invested. The experiments of this chapter are conducted on real smart contracts' transactions collected from the Ethereum blockchain using EtherScan [1]. To the best of our knowledge, this benchmarking approach is the first of its kind, and the results showed that the fees collected by miners are not always proportional to the CPU usage for both contract creation and call contract.

Chapter 4 presented OpBench, an Ethereum performance benchmark approach for smart contract operational code. It showed a detailed design framework as well as implementations for three Ethereum clients. OpBench assesses, for each opcode the CPU usage required by the EVM for its execution.

Chapter 5 presented two set of experimental results of the OpBench system in different clients, machines and operating systems. It concluded that there can be an order of magnitude difference in terms of the reward per unit of CPU time for different

[1] https://www.etherscan.io/

opcodes. Our experiments also indicate that there is a considerable performance difference between clients and operation systems, with the Parity client on Windows typically outperforming the PyEthApp and the Go-Ethereum clients.

Chapter 6 relied on the smart contract technology and its underlying, blockchain, to propose a solution aiming to achieve both verifiability and cost-efficiency proprieties for the cloud computing domain. In the solution, the user outsources the same computation for two clouds and uses smart contracts to create games between two rational clouds. The research focused particularly on the implementation and performance measurement. This research concluded that the cost of cheating and/or colluding is higher than the cost of the being honest. In this research, we leverage the previous two Chapters 3 and 4 to create efficient and low-cost smart contacts by understanding which opcodes to include and to avoid in our implementations. Thus, the financial cost of this research was small.

Similar to Chapter 6, Chapter 7 leveraged the results of Chapters 3 and 4 and presented the impossibilities, the possibilities, the cost, and the trade-offs in this blockchain-based approach to blocking-free management of distributed transactions. We presented a blockchain-coordinated 2PC protocol with rigorous arguments for its correctness under the synchrony requirements. We implemented this protocol on the Ethereum private and test networks and demonstrated, through our experiments, that the monetary cost of executing smart contracts is quite small, that the protocol performances is low when using a public blockchain like Ethereum, and that even significant synchrony requirements lead only to relatively small increases in unnecessary aborts. We thus identified a trade-off between improving protocol performance and admitting a risk that transactions could occasionally abort unnecessarily.

8.2 Future Work

Future work is suggested as follows.

- In Chapter 3 (on Performance Benchmarking of Blockchain Smart Contracts), With respect to our proposed benchmarking approach, it would be useful to improve the proposed methods, for instance considering computing effort beyond CPU usage (e.g. storage), and relating computational effort more directly to actual energy costs as appeared to CPU usage. Also, although we conducted experiments on several platforms, additional experiments can cover more contracts, functions, types of client codes, operating systems and hardware.

- In Chapters 4 and 5 (on Performance Benchmark of Blockchain Smart Contract Operation Code (Opcode) and Experimental Results and Discussion), in the future, it appears possible to expand on the reported experiments, to further compare across clients, operating systems and CPU specification. Besides, it will be of interest to expand the scope of the benchmark to include assessment of occupying resources in general, including storage, blocking the machine as well as actual energy consumption.

- In Chapter 5 (on Counter-Collusion Smart Contracts for Verifiable Cloud Computing), one future direction would be to consider the client as a potential adversary. This would make the interplay among parties more complex and requires significant changes to the contracts. Another future direction would be to consider repeated interactions among the parties. Repeated interactions introduce significant changes to the settings because the incentive can be now influenced by reputation and long-term profitability. Also, the current deposit mechanism is not very efficient from the cloud point of view. If the cloud has many clients and simultaneous contracts, the cloud must have a large cash reserve to pay all deposits at the same time. One direction would be to investigate more efficient deposit mechanisms by, e.g. pooling contracts or insurance.

- Finally, in Chapter 6 (on Non-Blocking Two Phase Commit Protocol Using Blockchain), it is worth attempting to build the 2PC protocol in the Ethereum main network to estimate the latency and the fee costs in a real world case. In the Ethereum main network the transaction's fees provided for miners affect on the latency of the transactions, the more you pay, the faster the transactions are mined and then included in the next block. Hence, a trade-off between the fees and the latency should be considered. In addition to the Ethereum blockchain, it would be useful to rebuild the 2PC protocol on the Hyperledger blockchain [13], which considered as a permissioned blockchain. Then, comparing the latency on the Ethereum private network against the Hyperledger network.

Bibliography

[1] Nurzhan Zhumabekuly Aitzhan and Davor Svetinovic. Security and privacy in decentralized energy trading through multi-signatures, blockchain and anonymous messaging streams. *IEEE Transactions on Dependable and Secure Computing*, 15(5):840–852, 2016.

[2] Alcio. Monitor pay to script hash adoption. `http://p2sh.info/`, 2015. Accessed on: 2016-09-09.

[3] A. Aldweesh, M. Alharby, and A. Van Moorsel. Performance benchmarking for ethereum opcodes. In *Computer Systems and Applications (AICCSA), 2018 ACS International Conference on*. IEEE, 2018.

[4] Amjad Aldweesh. Opbench full results. `https://github.com/mjod89`. Accessed on: 2019-01-20.

[5] Amjad Aldweesh, Maher Alharby, Mehrnezhad Maryam, and Aad van Moorsel. Opbench: A cpu performance benchmark for ethereum smart contract operation code. In *Proceedings of the 2019 IEEE International Conference on Blockchain (Blockchain-2019)*. IEEE, 2019.

[6] Amjad Aldweesh, Maher Alharby, Ellis Solaiman, and Aad van Moorsel. Performance benchmarking of smart contracts to assess miner incentives in ethereum. In *Dependable Computing Conference (EDCC), 2018 14th European*. IEEE, 2018.

[7] Amjad Aldweesh and Aad van Moorsel. A survey about blockchain software architectures. In *Proceedings of the 32nd Annual UK Performance Engineering Workshop & Cyber Security Workshop*, 2016.

[8] M. Alharby, A. Aldweesh, and A. Van Moorsel. Blockchain-based smart contracts: A systematic mapping study of academic research (2018). In *Cloud Computing, Big Data and Blockchain (ICCBB 2018), International Conference on*, 2018, 2018. IEEE.

[9] Maher Alharby, Amjad Aldweesh, and Aad van Moorsel. Blockchain-based smart contracts: A systematic mapping study of academic research (2018). In *Proceedings of the 2018 International Conference on Cloud Computing, Big Data and Blockchain*, 2018.

[10] Maher Alharby and Aad van Moorsel. The impact of profit uncertainty on miner decisions in blockchain systems. *Electronic Notes in Theoretical Computer Science*, 340:151–167, 2018.

[11] Amazon. Total cost of ownership (tco). `https://awstcocalculator.com/`, 2017. Accessed on: 2017.

[12] Elli Androulaki. et. al. *Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains, Cornell University Archive //arxiv.org/pdf/*, 1801:10228, January 2018.

[13] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, page 30. ACM, 2018.

[14] Andreas M Antonopoulos. *Mastering Bitcoin: unlocking digital cryptocurrencies.* " O'Reilly Media, Inc.", 2014.

[15] E. Arjomandi and M. J. Fischer and. N.a.lynch. Efficiency of Synchronous Versus Asynchronous Distributed Systems. *Journal of the ACM*, 30(3):449–456, July 1983.

[16] Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B Jones, and Bodo Parady. Specomp: A new benchmark suite for measuring parallel computer performance. In *International Workshop on OpenMP Applications and Tools*, pages 1–10. Springer, 2001.

[17] Jörg Becker, Dominic Breuker, Tobias Heide, Justus Holler, Hans Peter Rauer, and Rainer Böhme. Can we afford integrity by proof-of-work? scenarios inspired by the bitcoin currency. In *The economics of information security and privacy*, pages 135–156. Springer, 2013.

[18] Mira Belenkiy, Melissa Chase, C Chris Erway, John Jannotti, Alptekin K, and Anna Lysyanskaya. Incentivizing outsourced computation. In *Proceedings of the 3rd international workshop on Economics of networked systems*, pages 85–90. ACM, 2008.

[19] V. Buterin. Transaction spam attack: next steps. `https://goo.gl/uKi9Ug`, 2016. Accessed on: 2017-05-02.

[20] Vitalik Buterin et al. Ethereum white paper. *GitHub repository*, pages 22–23, 2013.

[21] Francesc Campoy. Analyzing the performance of go functions with benchmarks. `https://cutt.ly/AwOYBLJ`. Accessed on: 2018-01-27.

[22] M. Carlsten, H. Kalodner, A. Weinberg, and A. Narayanan. On the instability of bitcoin without the block reward. In *ACM SIGSAC Conference on Computer and Communications*, 2016.

[23] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

[24] Ting Chen, Xiaoqi Li, Ying Wang, Jiachi Chen, Zihao Li, Xiapu Luo, Man Ho Au, and Xiaosong Zhang. An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. In *International Conference on Information Security Practice and Experience*, pages 3–24. Springer, 2017.

[25] JBAMJ Clark, ANJAK Edward, and W Felten. Research perspectives and challenges for bitcoin and cryptocurrencies. *url: https://eprint. iacr. org/2015/261. pdf*, 2015.

[26] Jeremy Clark and Aleksander Essex. Commitcoin: Carbon dating commitments with bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 390–398. Springer, 2012.

[27] Michael Crosby, Pradan Pattanayak, Sanjeev Verma, Vignesh Kalyanaraman, et al. Blockchain technology: Beyond bitcoin. *Applied Innovation*, 2(6-10):71, 2016.

[28] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. Step by step towards creating a safe smart contract: Lessons and insights from a

cryptocurrency lab. In *International Conference on Financial Cryptography and Data Security*, pages 79–94. Springer, 2016.

[29] Python developers. Mathematical statistics functions. `https://www.scipy.org/`. Accessed on: 2017-08-20.

[30] SciPy developers. Scientific computing tools for python. `https://www.scipy.org/`. Accessed on: 2017-07-11.

[31] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. Blockbench: A framework for analyzing private blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1085–1100. ACM, 2017.

[32] GoLang Documentation. Benchmarks. `https://golang.org/pkg/testing/`. Accessed on: 2017-07-12.

[33] Python Documentation. Measure execution time. `https://docs.python.org/2/library/timeit.html`. Accessed on: 2017-07-23.

[34] C. Dong, Y. Wang, A. Aldweesh, P. McCorry, and A. Van Moorsel. Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing. In *ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[35] Partha Dutta, Rachid Guerraoui, and Bastian Pochon. Fast Non-blocking Atomic Commit: An Inherent Trade-off. *Information Processing Letters*, 91(4):195–200, August 2004.

[36] Yasuhiro Endo, Zheng Wang, J Bradley Chen, and Margo I Seltzer. Using latency to evaluate interactive system performance. *ACM SIGOPS Operating Systems Review*, 30(si):185–199, 1996.

[37] Paul Ezhilchelvan, Amjad Aldweesh, and Aad van Moorsel. Non blocking two phase commit using blockchain. In *Proceedings of the 1st Workshop on Cryptocurrencies and Blockchains for Distributed Systems*, pages 36–41. ACM, 2018.

[38] Paul Ezhilchelvan, Amjad Aldweesh, and Aad van Moorsel. Non-blocking two-phase commit using blockchain. *Concurrency and Computation: Practice and Experience*, page e5276, 2019.

[39] Zsolt Felfldi. Introduction of the light client for dapp developers. https://blog.ethereum.org/2017/01/07/introduction-light-client-dapp-developers/. Accessed on: 2017-03-12.

[40] International Organization for Standardization. *Systems and Software Engineering: Systems and Software Quality Requirements and Evaluation (SQuaRE): Measurement of System and Software Product Quality*. ISO, 2016.

[41] Ethereum Foundation. Ethereum ropsten. https://ropsten.etherscan.io/. Accessed on: 2018-01-30.

[42] Ethereum Foundation. Evm implementation for parity. https://github.com/paritytech/parity-ethereum/tree/master/evmbin. Accessed on: 2018-05-12.

[43] Ethereum Foundation. Evm instructions gas cost. https://docs.google.com/spreadsheets/d/. Accessed on: 2016-09-30.

[44] Ethereum Foundation. Official go implementation of the ethereum protocol. https://geth.ethereum.org. Accessed on: 2017-09-19.

[45] Ethereum Foundation. Parity-ethereum. https://www.parity.io/. Accessed on: 2017-09-11.

[46] Ethereum Foundation. Python-based client implementing the ethereum. https://github.com/ethereum/pyethereum. Accessed on: 2017-01-12.

[47] Ethereum Foundation. The solidity compiler. https://solidity.readthedocs.io/en/v0.5.3/installing-solidity.html#versioning. Accessed on: 2018-02-11.

[48] Ethereum Foundation. Solidity language. https://goo.gl/jdgoYi. Accessed on: 2018-06-20.

[49] Ethereum Foundation. Get ether. ttps://ethereum.org/ethe, 2016. Accessed on: 2016.

[50] Ethereum Foundation. Introduction — ethereumfrontier guide. https://ethereum.gitbooks.io/frontier-guide/content/mining.html, 2016. Accessed on: 2016.

[51] Ethereum Foundation. Ethereum Mist. https://github.com/ethereum/mist/releases, 2018. Accessed on: 2018-03-02.

[52] The Ethereum Foundation. The ethereum nodes explorer. https://www.ethernodes.org/network/1. Accessed on: 2018-02-12.

[53] Gartner. Gartner says by 2020 cloud shift will affect more than $1 trillion in it spending. http://www.gartner.com/newsroom/id/338472, 2017. Accessed on: 2017.

[54] A. Gervais, G. Karame, K. Wuest, V. Glykantzis, H. Ritzdorf, and S. Capkun. On the security and performance of proof of work blockchains. In *ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[55] SPEC Glossary. Standard performance evaluation corporation (spec). http://spec.org/spec/glossary/. Accessed on: 2017-11-12.

[56] Jim Gray. Notes on Data Base Operating Systems. In Michael J. Flynn, Jim Gray, Anita K. Jones, Klaus Lagally, Holger Opderbeck, Gerald J. Popek, Brian Randell, Jerome H. Saltzer, and Hans-R"udiger Wiehle, editors, *Operating Systems, An Advanced Course*, pages 393–481. Springer-Verlag, London, UK, 1978.

[57] R. Guerraoui. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distributed Computing*, 15:17–25, 2002.

[58] V. Hadzilacos. On the relationship between the atomic commitment and consensus problems. In B. Simons and A. Spector, editors, *Fault-Tolerant Distributed Computing*, pages 201–08. LNCS 448, Springer, 1987.

[59] Doug Hellmann. Performance analysis of python programs. https://pymotw.com/2/profile/index.html#module-profile. Accessed on: 2017-09-03.

[60] Alyssa Hertig. Ethereums big switch: The new roadmap to proof-of-stake. https://bit.ly/2xRmPjB. Accessed on: 2017-08-13.

[61] Stan Higgins. Ibm invests $200 million in blockchain-powered iot. https://www.coindesk.com/ibm-blockchain-iot-office. Accessed on: 2016-12-03.

[62] Stan Higgins. Russias central securities depository tests blockchain assets exchange. https://www.coindesk.com/russia-central-securities-depository-blockchain-assets. Accessed on: 2017-01-03.

[63] Ethereum Homestead. Why are there multiple ethereum clients? http://ethdocs.org/en/latest/ethereum-clients/choosing-a-client.html#why-are-there-multiple-ethereum-clients. Accessed on: 2017-02-03.

[64] IEC ISO. Ieee, systems and software engineering–vocabulary. *IEEE computer society, Piscataway, NJ*, 2010.

[65] Bob Jenkins. A hash function for hash table lookup. http://www.burtleburtle.net/bob/hash/doobs.html. Accessed on: 2016-11-12.

[66] Paul John. Golang vs python. https://www.edureka.co/blog/golang-vs-python/#perf. Accessed on: 2018-09-11.

[67] Tom Kaitchuck. Rust versus go fastest programs. https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust-go.html. Accessed on: 2018-03-23.

[68] Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

[69] Butler W. Lampson. Atomic Transactions. In An Advanced Course, B. W. Lampson, M. Paul, and H. J. Siegert, editors, *Distributed Systems - Architecture and Implementation*, pages 246–265. Springer-Verlag, London, 1981.

[70] Iuon-Chang Lin and Tzu-Chun Liao. A survey of blockchain security issues and challenges. *IJ Network Security*, 19(5):653–659, 2017.

[71] Robert C Marshall and Leslie M Marx. *The economics of collusion: Cartels and bidding rings*. Mit Press, 2012.

[72] Patrick McCorry. *Applications of the Blockchain using Cryptography*. PhD dissertation, School of computing, 2018.

[73] Patrick McCorry, Siamak F Shahandashti, and Feng Hao. A smart contract for boardroom voting with maximum voter privacy. In *International Conference on Financial Cryptography and Data Security*, pages 357–375. Springer, 2017.

[74] Jeffrey C Mogul. Brittle metrics in operating systems research. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pages 90–95. IEEE, 1999.

[75] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.221.9986*, 2008.

[76] pirapira. [wip] metropolis: elliptic curve precompiled contracts. `https://github.com/ethereum/yellowpaper/pull/297`, 2017. Accessed on: 2017.

[77] Meikel Poess and Chris Floyd. New tpc benchmarks for decision support and web commerce. *ACM Sigmod Record*, 29(4):64–71, 2000.

[78] Michel Raynal. A case study of agreement problems in distributed systems: non-blocking atomic commitment. In *Proceedings 1997 High-Assurance Engineering Workshop*, pages 209–214. IEEE, 1997.

[79] Michel Raynal and Mukesh Singhal. Mastering agreement problems in distributed systems. *IEEE Software*, 15(4):40–47, 2001.

[80] RightScale. Rightscale 2016 state of the cloud report. `http://assets.rightscale.com/uploads/pdfs/RightScale-2016-State-of-the-Cloud-Report.pdf`, 2016. Accessed on: 2017.

[81] Ameer Rosic. What is ethereum gas? [the most comprehensive step-by-step guide ever!]. `https://blockgeeks.com/guides/ethereum-gas/`. Accessed on: 2018-06-20.

[82] Margo Seltzer, David Krinsky, Keith Smith, and Xiaolan Zhang. The case for application-specific benchmarking. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pages 102–107. IEEE, 1999.

[83] Dale Skeen. Nonblocking commit protocols. In *ACM SIGMOD international conference on Management of data*, pages 133–142. Proc. (SIGMOD 81, 1981.

[84] Synergy. Review shows $148 billion cloud market growing at 25% annually. https://www.srgresearch.com/articles/2016-review-shows-148-billion-cloud-market-growing-25-annually., 2016. Accessed on: 2016.

[85] Ahmad Syukri. The computer language benchmarks. `https://benchmarksgame-team.pages.debian.net/benchmarksgame/program/regexredux-python3-1.html`. Accessed on: 2019-04-16.

[86] Alex Tapscott and Don Tapscott. *How Blockchain Is Changing Finance*. Harvard Business Review, 2017.

[87] Rust Team. Learn rust. `https://www.rust-lang.org/learn`. Accessed on: 2019-01-27.

[88] Lester G Telser. *Competition, collusion, and game theory*. Routledge, 2017.

[89] Walter F Tichy. Should computer scientists experiment more? *Computer*, 31(5):32–40, 1998.

[90] M. Vieira, H. Madeira, K. Sachs, and S. Kounev. Resilience benchmarking. In *Resilience Assessment and Evaluation of Computing Systems, Wolter et al.* Springer, 2012.

[91] Michael Walfish and Andrew J Blumberg. Verifying computations without reexecuting them. *Communications of the ACM*, 58(2):74–84, 2015.

[92] I. Weber. et. al. In *Symposium on Reliable Distributed Systems*. On Availability for Blockchain-based Systems. In Proceedings of the 36th (SRDS17). IEEE, 2017.

[93] Ethereum Wiki. Ethash. `https://github.com/ethereum/wiki/wiki/Ethash`, 2015. Accessed on: 2016.

[94] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.

[95] Xiwei Xu. et. al. In *Working IEEE/IFIP Conference on Software Architecture*. The Blockchain as a Software Connector. In proceedings of the 13th (WICSA, April 2016.

[96] A. Zamyatin, K. Wolter, S. Werner, C. Mulligan, P. Harrison, and W. Knottenbelt. *Swimming with Fishes and Sharks: Beneath the Surface of Queue-based Ethereum Mining Pools*. IEEE Mascots, 2017.

[97] Peilin Zheng, Zibin Zheng, Xiapu Luo, Xiangping Chen, and Xuanzhe Liu. A detailed and real-time performance monitoring framework for blockchain systems. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 134–143. ACM, 2018.