2019 IEEE International Conference on Blockchain (Blockchain)

OpBench: A CPU Performance Benchmark for Ethereum Smart Contract Operation Code

***Abstract*—Ethereum and other blockchains rely on miners contributing 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, 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 Ethereum Virtual Machine. We implemented OpBench for the PyEthApp 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. The results also show a markable difference in performance between clients, with the Go client outperforming the Python client across machines.**

***Index Terms*—Blockchain, Ethereum, Smart Contract, Benchmark, Performance**

I. Introduction

The proper functioning of permissionless blockchains such as Bitcoin [1] and Ethereum [2] 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 [3]. 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 [4], [5]. Therefore, for miners and for the successful operation of the blockchain, it is critical to understand the relation between the fee received and 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 [2], 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 a number of 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 paper, 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 a 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., [3]. 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 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 paper 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 two implementations of OpBench: for the PyEthApp client (in Python) 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 libraries to measure performance accurately. To

978-1-7281-4693-5/19/$31.00 ©2019 IEEE
DOI 10.1109/Blockchain.2019.00043

be of practical use, OpBench runs independent from the live Ethereum blockchain.

The paper also presents the results for six system configurations, running the two different implementations on three different computers (and operating systems). These experiments demonstrate the validity of the OpBench approach across platforms. It also allows us to obtain results comparing PyEthApp with Go-Ethereum, both 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 experiments also show that the Go client outperforms the Python client across hardware and operating system configurations.

The paper is organized as follows. In Section II, we explain some of the fundamental Ethereum's concepts such as EVM and smart contracts. Section III offers a detailed design framework for our proposed OpBench system. In Section IV, we present the implementation of our system in both PyEthApp and Go-Ethereum. Section V presents the results and our observations. The related work is presented in Section VI, and we conclude our paper in Section VII.

II. Ethereum

In this section, we explain the main concepts and features of Ethereum essential to this work. These include the overall operation of the Ethereum blockchain, Ethereum Virtual Machine (EVM), and Ethereum's fee structure.

A. Ethereum Public Blockchain

A blockchain is a distributed ledger that maintains the history of all the transactions within a peer-to-peer network, see, e.g., [6]. It is a list of linked blocks, where each block includes a set of verified transactions. In the blockchain system, a set of nodes, referred to as miners, are responsible for expanding the blockchain ledger by appending new blocks containing transactions to the blockchain. Miners first collect unconfirmed transactions from the network, validate them, and then generate a block containing those transactions by engaging in a computational process, called proof of work (PoW). There are two types of blockchains: public and private. The former allows anyone to join the network, while the latter is restricted to users with a granted permission. Bitcoin and Ethereum are among the most popular examples of public blockchains. Ethereum, however, distinguishes itself by its ability to support complex distributed applications through Turing complete smart contracts [7].

A smart contract is a computer program deployed and executed on a blockchain like Ethereum, with the accuracy of execution being imposed by the consensus protocol. In Ethereum, a contract can include any type of agreements that can be expressed in high-level languages such as Solidity, Serpent and LLL. The source code of the smart contract needs to be transformed into bytecode so that the EVM can interpret it. A range of applications can be created using smart contracts including but not limited to financial applications, such as saving wallets or wills. According to Etherscan[1], the number of active Ethereum smart contracts at the time of writing is over 50k.

Ethereum is probably the most popular blockchain for smart contracts [5]. The currency of Ethereum is referred to as Ether. In Ethereum, there are two types of users, also 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. There are two forms of transactions in Ethereum: *Financial* and *Contract-creation/Execution*. 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 (Contract-execution).

For the contract-creation transaction, one needs to attach the compiled bytecode for the smart contract that needs to be deployed. Upon a 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 Contract-execution transaction that specifies the address of the contract, the functions to be executed, and possibly all input data required by those functions.

B. Ethereum EVM overview

The EVM is a Turing complete machine with a predefined set of instructions, referred to as opcodes, that handles the execution of smart contracts and their states. The current implementation of the EVM supports 150 different opcodes. The EVM relies on a stack-based language in which opcodes are executed by pushing and popping the required input values onto the stack.

As mentioned in Section II-A, the contract is complied into bytecode (a series of opcodes) before executing it on the EVM. The EVM, therefore, can be seen as a decentralized global machine where all smart contracts are executed on. The execution of a smart contract is handled by a transaction, in which the transaction contains the bytecode required to execute that contract.

The EVM executes all the opcodes that are associated with the contract's transaction one by one. Each opcode executed on the EVM has a pre-defined cost, measured in a gas unit, depending on the complexity of the opcode. It is worth noting that the gas cost of opcodes is set by the Ethereum foundation. For instance, `ADD` opcode costs 3 units of gas. The EVM keeps a record of the amount of gas consumed by the transaction's opcodes. This is to charge the submitter of that transaction a fee based on the total amount of gas consumed.

C. Ethereum Incentive Mechanisms

This section discusses the operations and motivation behind the incentive mechanism in Ethereum. As mentioned before,

[1]https://etherscan.io/

there are two types of transactions in Ethereum: *Financial* and *Contract-creation/Execution*. The cost of a *Financial* transaction is fixed, and it is 21k units of gas. The cost of a *Contract-creation/Execution* transaction is subject to the amount of gas that the EVM consumes during the execution of the transaction, in addition to the transaction cost, which is 21k units of gas.

Prior to submitting a transaction in Ethereum, the submitter of the transaction needs to set the highest level of gas payable for the transaction, called *GasLimit*. In addition, the submitter needs to set the *GasPrice*, the amount of ether to be paid per unit of gas. As explained in II-B, during the transaction execution, the EVM keeps track of the amount of gas consumed by the transaction. If the *GasLimit* provided is reached before finishing the execution of the transaction, the miner will terminate the execution of that transaction with an out-of-gas exception. Otherwise, the transaction will be executed successfully. Regardless of whether the transaction has been executed successfully or not, the miner who executed the transaction will collect its fee as compensation for their computational efforts. The fee is calculated as the multiplication of the amount of gas used by the transaction and the *GasPrice*.

Blockchain systems such as Ethereum rely on miners to operate them correctly and efficiently. Miners pay for the cost of the electricity and the cost of hardware to operate the blockchains, and thus, it is important to ensure that the incentive mechanisms are well aligned with the cost imposed. If not, miners might not be incentivized to run the blockchain properly as they are not able to optimize their profits, see for instance the modeling study in [3]. There are other reasons why the fee structure is considered critical. If the submitter places a limit on the transacted gas quantity, the execution is sure to terminate. This avoids the halting problem of Turing-complete systems, as termination is certain. Another important reason for the fee structure is that it avoids of denial of service attacks, which occur if excessive miner resources are used for no substantial fee [4], [5]. Therefore, preventing a user to submit such tasks by charging substantial fees is necessary for the blockchain to operate effectively. The 2016 attacks on Ethereum, i.e., the `EXTCODESIZE` attack, happened due to the small fees of particular opcodes with high computation levels. in response to this attack, Ethereum increased substantially the `EXTCODESIZE` fee to mitigate these attacks.

III. 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.

A. OpBench Overview

The usage of OpBench can be divided in four stages, as depicted in Fig. 1. In the first stage, we utilize PyEthApp [8] 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 requires 2 inputs and 1 output and it costs 3 unit of gas. Note that also if the EVM is in a different language, this Preprocessing stage can be done in Python.

In stage 2, for each opcode from stage 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 EVM state file in Fig. 1, showing generated Ethereum state, including account information (balance, address), environment information (timestamp, blocknumber) as well as execution information (code, gas and data).

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). For example, `ADD64` in Fig. 1 refer to 64 bits addition operation. 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.

Stage 3 deploys the smart contract bytecode on the blockchain by submitting an Ethereum transaction. In this stage, we initiate a transaction that executes the deployed bytecode on the EVM, playing the roles of both the sender and miner. In the example, the EVM runs the `ADD` opcode by pushing 0 and 1 onto the stack (`PUSH1 0 PUSH1 1`) then adding these numbers (`ADD`), and finally popping the result (`POP`) from the stack.

The computation time of each bytecode is recorded in stage 4. 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 Fig. 1, the result of the execution of an opcode is reported as a cvs formatted file that shows the used gas, the name of the opcode and some statistics.

B. Classification of Opcodes and their Benchmarking

Different opcodes require different treatment when benchmarking their performance. Following the approach of [2], 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 [2]. 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.

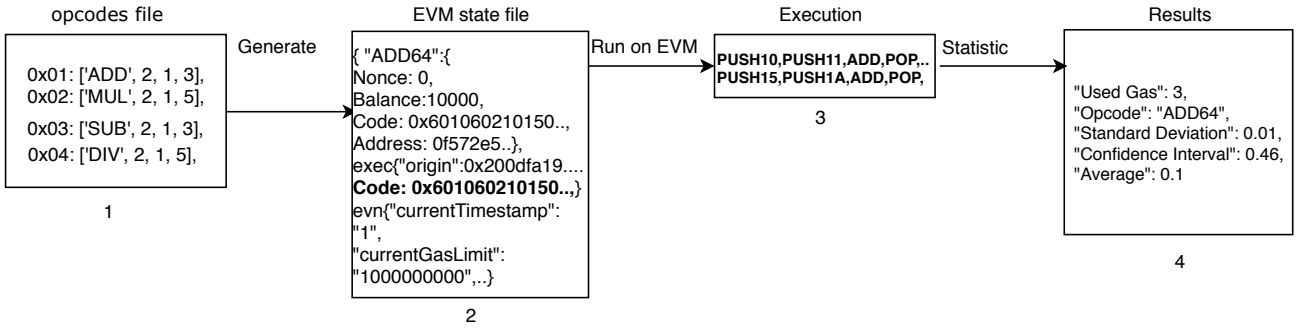


Fig. 1: OpBench overview

A third type of the opcodes have an associated fee that is not related to CPU usage, and it is therefore not useful to add them to OpBench. These opcodes fall in the system and log opcode categories, as described in [2]. For example, `CREATE`, which creates a new account with an associated code, has a fee of 32k gas, but this for creating (not running) the bytecode. In other words, this is not bound to the CPU usage. Similarly, log opcodes append a log record to the blockchain, which is based on the size of the disk, not on the CPU consumption.

1) Computation-based opcodes: This class of opcodes includes the ones in the *Stop and Arithmetic* operations, *Comparison* and *Bitwise Logic Operations*, using the categories identified in [2]. Table I provides the category names associated with all opcodes. 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 inputs: `ADD64, ADD128` and `ADD256`.

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 input size (256-bits). For these opcodes, first we push the required parameters onto the stack, and then we execute the actual opcode. Some opcodes push 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.

2) Formula-based opcodes: The formula-based opcodes are more intricate to benchmark, so we describe these in more detail in this section. 6 out of 150 opcodes belong to this type [2] of which we explain six representative ones here.

The `EXP` opcode is the exponential operation, and its formula is shown in Equation (1). `Exp` opcode pops two values form the stack and calculates the exponential (second value) for the first value, and pushes the result onto the stack. For this opcode, we repeat the experiments with different exponent sizes (64-bits to 256-bits).

$$gas = \begin{cases} 10, & \text{if } Exp = 0 \\ 10 + 50 \times (1 + \log 256(Exp)), & \text{if } Exp > 0 \end{cases} \quad (1)$$

The gas price for the `SHA3` opcode which computes the Keccak-256 hash for any input is calculated according to Equation (2). The `SHA3` opcode's parameters are the memory location 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. 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 the OpBench design, we benchmark this opcode with various parameters sizes that are 1, 2, 3 and 4 words, and each word is 32-bytes.

$$gas = \begin{cases} 30, & \text{if } input = 0 \\ 30 + 6 \times (input's\ size), & \text{if } input > 0 \end{cases} \quad (2)$$

For the `SSTORE` opcode that saves a word to storage, we repeat this opcode with random parameters, setting the storage to non-zero from zero and updating the current storage as well as setting the storage to zero from non-zero. The storage in Ethereum refers to a persistent memory area. The gas cost of the `SSTORE` opcode is $20k$ units if the storage is set to non-zero from zero, and $5k$ units for storage update as shown in Equation (3).

$$gas = \begin{cases} 20000, & \text{if } input \neq 0 \ \&\& \ storage = 0 \\ 5000, & \text{otherwise} \end{cases} \quad (3)$$

The `CALLDATACOPY` and `CODECOPY` opcodes to copy the input data in the current environment and the code running in the EVM to the memory, respectively. The `CALLDATACOPY` opcode pops the transaction's input size out from the stack and copies it to the memory. For this opcode, two sizes of parameters are used i.e., 1 and 2 words. The `CODECOPY` opcode also pops the code running from the stack and copies the message input to the memory. Similar the previous opcode, we execute this opcode with different input message and code sizes (i.e. 1 and 2 words). The gas calculation formulas for these opcodes are calculated according to Equation (4).

$$gas = \begin{cases} 2, & \text{if } word = 0 \\ 2 + 3 \times (\#\ words), & \text{if } word > 0 \end{cases} \quad (4)$$

The `EXTCODECOPY` opcode is to copy an account's code to the memory. 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. database), then copies the code to

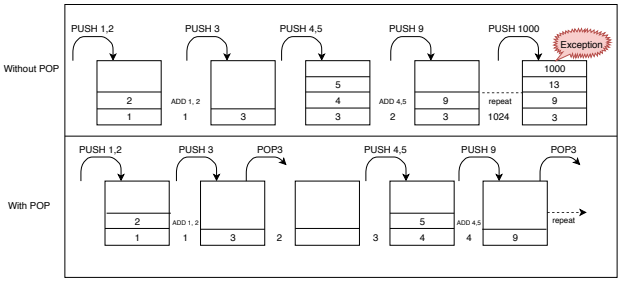


Fig. 2: Utilizing the POP opcode to overcome the stack size limitation.

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 5. We run this opcode with different code's size to report the results.

$$gas = \begin{cases} 700, & \text{if } word = 0 \\ 700 + 3 \times (\#\ words), & \text{if } word > 0 \end{cases} \quad (5)$$

C. 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 needed 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 [2] and a stack limit exception occurs when the stack size reaches 1024. For example, 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. When we want to repeat the ADD until the stack becomes full, 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. Fig. 2 illustrates an example of executing the ADD opcode on the EVM and how the stack is utilized. The top part of Figure 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 the OpBench is a list of opcodes and their execution times. The accuracy of setting the timer is achieved by utilizing the *Timeit* methods in Python, which runs the opcode snippet a high amount of times (default to 1m). In this way, we get the most accurate measurement of the execution time [9]. In Golang, we make use of the build-in *Benchmark* method in [10], with the same effect.

IV. Implementation

In this section, we present the implementations of our OpBench system for two Ethereum clients: PyEthApp [8] and Go-Ethereum [11]. We choose Go-Ethereum since it is one of the most popular platforms [12] and PyEthApp since the initial gas allocation for opcodes is based on a benchmark performed on the PyEthApp client.

A. PyEthApp

The OpBench system on the PyEthApp client has two phases. The first phase sets up the system by generating the smart contract bytecode, which contains a series of opcodes for an actual opcode and its required parameters. In the second phase, we create a transaction that deploys the smart contract bytecode to the blockchain. We also create another transaction that invokes the deployed bytecode and executes it on the EVM.

For the PyEthApp client implementation, we implement a new benchmarking (profiling) method (PyEth-Bench) to measure the performance of a single opcode utilizing the *timeit* method provided by Python.

B. Go-Ethereum

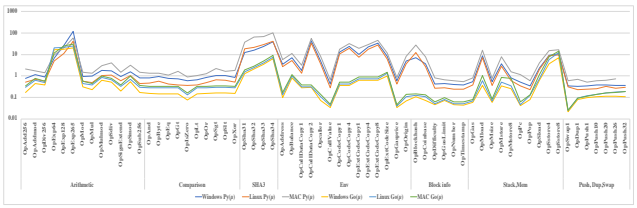
In order to benchmark opcodes in Golang, a separate method for each opcode must be implemented. Hence, for opcodes which require inputs with different sizes, we need to implement a separate method for each of them.

In the Go-Ethereum benchmark design, we use the smart contract created by Python (stage 2 of Fig. 1) and then utilize the *benchmark* method provided by Golang. The *benchmark* methods takes all the required parameters and executes the opcode on the EVM.

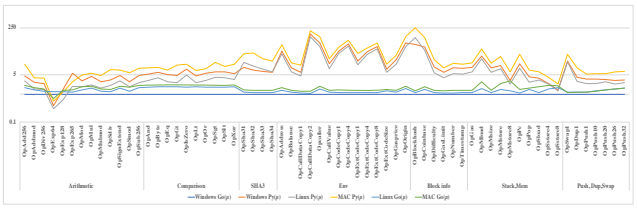
The execution of the opcode is repeated until the desired benchmark runtime is reached (default one second). We set the time high enough so that opcodes are executed sufficiently often and we check the accuracy by calculating a confidence interval.

V. Results and Discussion

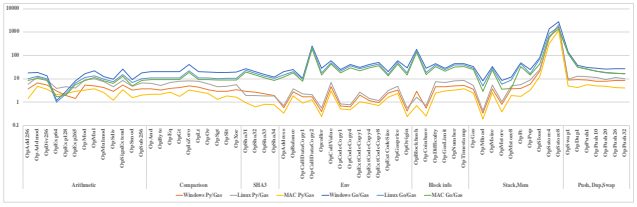
This section presents and discusses the results of experiments with the implementations of OpBench for two clients: Go-Ethereum and Python PyEthApp. We conduct our experiments using three different machines: (a) a MacBook Pro with a 2.8 GHz Intel i5 CPU and 8 GB RAM, (b) a desktop PC with a 3.20 GHz Intel i7 CPU and 16 GB RAM running on Ubuntu 16.04 and (c) a desktop PC with a 3.60 GHz Intel i5 (6 CPUs) and 32GB RAM running on Windows 10. Therefore, our results in this section concern six different platforms, denoted



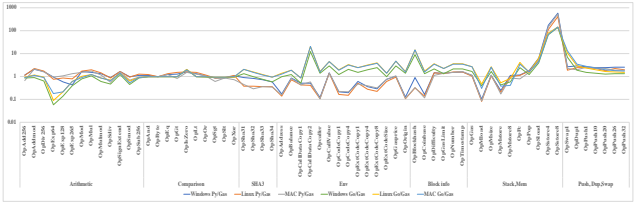
(a) CPU time (in microseconds) for each opcode.



(b) CPU time for each opcode, relative to the fastest platform (Windows Go 3.6GHz).



(c) Used Gas (per [2]) per CPU time unit (in Gas/microsecond). Reward and cost are proportional for a platform if the curve is a straight line.



(d) Normalized Used Gas per CPU time unit (results in 3c divided by the platform's result for opcode `Byte`). Reward and cost are proportional if the curve is a straight line at value 1.

as: Windows Go, Windows Python, Linux Go, Linux Python, Mac Go, and Mac Python. Note that our experiments aim to compare the Go and Python clients on different platforms, we do not aim to compare operating systems (this would require a different experiment setup).

The performance benchmark results for all opcodes are provided in Table I, in microseconds. We note that all confidence intervals are exceedingly tight, and are not provided in the table. As explained in Section IV, 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 a 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:

- The Go client is generally considerably faster (and thus more profitable) than the Python client, 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 3.6 GHz machine on the Python client, while the reverse is true for the Go client.
- 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.
- Python clients are able to gain higher fees for opcodes in the Arithmetic category compared to Go, and Go performs better across opcodes in the Environment category.

A. Absolute CPU time

Fig. 3a shows the graph with the CPU time required for each of the opcodes on all the six platforms, as given in Table I. The *x*-axis shows the same opcodes as in the table, and note that the CPU time on the *y*-axes is depicted in logarithmic scale. Many of the opcodes take in the order of one microsecond or less to execute, but some take considerably longer, in the order of 0.1 millisecond. There are several examples in 3a, e.g., the exponential and the hash operation code.

From Fig. 3a it is clear that the Go client outperforms the Python client on all three machines: Mac 2.8Ghz, Linux 3.2GHz and Windows 3.6GHz. This gap is particularly clear in opcodes belonging to the SHA3, Env and Block info categories. For example, the three machines consume between 40 and 102 μs to execute `SHA3` on the Python client, while they consume less than 10 μs on the Go client. Similarly, the machines consume between 30 and 60 μs to execute `CALLDATACOPY2` on the Python client, while they consume less than half a microsecond executing on the Go client.

B. Relative CPU time

In order to better compare the CPU usage results for different platforms, Fig. 3b shows the CPU time relative to the fastest platform, Windows Go 3.6Hz. In other words, we divided the results of each individual platform by the results of Windows Go, and, as a consequence, Windows Go shows as a straight line at 1 in Fig. 3b. Note that the Windows Go platform was not the fastest for some `EXP` opcodes, as is visible on the left side of the range in Fig. 3b.

As mentioned, the Go client generally outperforms the Python client: the three higher lines in Fig. 3b correspond to the Python clients and the three lower ones to the Go clients. The exceptions are `EXP64` and `EXP128` opcodes, where the Python client performs better. The Mac 2.8 GHz machine is

Category	opcode	Windows		Linux		MAC		Gas
		Py μ	Go μ	Py μ	Go μ	Py μ	Go μ	
Stop and Arithmetic	Add256	0.79	0.17	0.51	0.30	2.06	0.35	3
	Addmod	1.18	0.44	0.68	0.63	1.70	0.76	8
	Div256	0.91	0.38	0.56	0.49	1.41	0.58	5
	Exp64	6.56	16.66	5.05	20.41	9.47	11.90	20
	Exp128	26.01	17.45	11.06	21.68	20.80	24.41	50
	Exp265	118.80	20.83	41.03	25.22	59.17	29.91	170
	Mod	0.92	0.30	0.56	0.45	1.49	0.58	5
	Mul	1.00	0.23	0.50	0.37	1.32	0.44	5
	Mulmod	1.86	0.64	1.06	0.83	3.05	0.99	8
	Sdiv	1.73	0.52	1.09	0.65	4.00	0.79	5
	SignExtend	0.92	0.20	0.59	0.33	1.46	0.38	5
	Smod	1.53	0.54	1.03	0.71	3.22	1.00	5
	Sub256	0.81	0.17	0.44	0.30	1.47	0.36	3
Comparison & Bitwise	And	0.81	0.15	0.48	0.28	1.37	0.33	3
	Byte	0.92	0.15	0.58	0.27	1.36	0.32	3
	Eq	0.77	0.15	0.43	0.28	1.10	0.33	3
	Gt	0.72	0.15	0.39	0.27	1.67	0.33	3
	IsZero	0.59	0.07	0.36	0.14	0.90	0.16	3
	Lt	0.68	0.15	0.38	0.27	1.06	0.33	3
	Or	0.86	0.15	0.49	0.28	1.29	0.33	3
	Sgt	1.04	0.16	0.66	0.29	2.27	0.34	3
	Slt	1.04	0.16	0.63	0.29	1.61	0.34	3
	Xor	0.86	0.15	0.53	0.29	1.85	0.33	3
SHA3	Sha31	12.25	1.31	18.30	1.58	37.02	1.88	36
	Sha32	15.62	2.09	21.70	2.42	65.44	2.88	42
	Sha33	23.20	3.54	28.82	4.11	67.43	4.95	54
	Sha34	40.19	6.47	41.91	7.48	102.00	8.96	78
Environmental Information	Address	3.49	0.10	2.77	0.13	5.83	0.17	2
	Balance	7.23	0.81	5.27	0.97	10.93	1.13	20
	CallDataCopy1	1.79	0.29	1.34	0.31	3.23	0.38	5
	CallDataCopy2	42.56	0.29	33.80	0.31	56.39	0.38	72
	caller	4.93	0.07	3.64	0.11	8.32	0.14	2
	CallValue	0.43	0.03	0.28	0.04	0.65	0.05	2
	CodeCopy1	13.02	0.36	10.56	0.42	17.83	0.51	9
	CodeCopy4	23.65	0.36	19.39	0.42	32.26	0.51	15
	ExtCodeCopy1	10.02	0.64	7.37	0.74	19.28	0.89	20
	ExtCodeCopy4	22.43	0.64	17.63	0.74	29.73	0.89	26
	ExtCodeCopy8	33.93	0.64	27.42	0.74	45.07	0.89	32
	ExtCodeSize	8.13	1.00	6.20	1.32	11.98	1.50	20
	GasPrice	0.59	0.03	0.41	0.04	0.86	0.05	2
	Origin	4.87	0.07	3.63	0.11	8.12	0.14	2
Block Info	Blockhash	6.89	0.11	12.29	0.13	27.46	0.15	20
	Coinbase	3.63	0.07	2.78	0.11	7.88	0.13	2
	Difficulty	0.43	0.04	0.27	0.05	0.80	0.06	2
	GasLimit	0.44	0.07	0.28	0.08	0.65	0.10	2
	Number	0.40	0.04	0.25	0.05	0.60	0.06	2
	Timestamp	0.39	0.04	0.24	0.05	0.54	0.06	2
Stack, Memory & storage	Gas	0.55	0.06	0.38	0.07	0.81	0.08	2
	Mload	8.82	0.37	6.95	0.59	15.71	1.04	3
	Msize	0.54	0.06	0.37	0.07	0.77	0.08	2
	Mstore	3.70	0.34	2.80	0.52	7.86	0.85	3
	Mstore8	0.80	0.25	0.59	0.34	1.58	0.77	3
	Pc	0.52	0.04	0.35	0.05	1.13	0.06	2
	Pop	0.35	0.08	0.22	0.12	0.61	0.14	2
	Sload	2.37	0.62	1.99	0.72	4.00	1.17	50
	Sstore1	8.91	3.69	8.80	5.68	15.40	7.75	5000
	Sstore2	10.00	7.17	9.50	12.52	16.92	14.63	20000
PUSH	Swap1	0.34	0.02	0.31	0.03	0.60	0.02	3
	Dup1	0.32	0.08	0.23	0.09	0.70	0.09	3
	Push1	0.34	0.10	0.24	0.12	0.52	0.12	3

TABLE I: The average CPU time for each of the opcodes for all six platforms. The right-most column provides the gas cost.

slower than other machines on both clients and this is expected since it has lower specifications. However, the Mac machine does outperform in some of the `EXP` opcodes (e.g., `EXP64` on the Go client). In addition, the Mac machine is as fast as the Linux machine in the Go client for `DUP`, `PUSH` and `SWAP` opcodes.

There is a very interesting difference between the Python and Go client in terms of the operating system on which they perform best. For Python, the Linux 3.2 GHz machine performs better than the Windows 3.6 GHz machine, whereas for the Go client, 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.

C. Absolute gas/CPU

The critical issue for the successful operation of Ethereum is not the CPU time required for opcode, but the ratio between the fee obtained and the CPU invested. This is displayed in Fig. 3c, 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 I and is calculated by the EVM from values set in [2]. The higher the used gas, the more profit a miner makes.

Ideally, the curves in Fig. 3c for each platform would be straight lines, since then fee and cost are proportional to each other across all opcodes. Unfortunately, there is considerable deviation from 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 Go outperforms Python. The range of values for the respective clients is significantly different. For the Go client, the average collected gas per microsecond of CPU usage varies between 54 and 98 units of gas, while it ranges from 26 and 47 in the Python client. In other words, the average amount of the awarded gas in the Go client is about twice that of the Python client.

In both 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 Python 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 microsecond. In the Go client, opcodes such as `EXP64` and `EXP128` are the most expensive ones.

D. Normalized gas/CPU

To remove the platform-specific element from the results, we introduce normalized results for the gas user per microsecond in Fig. 3d. In this figure, 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 from the left in the Comparison category). For each platform, we took the Gas/CPU value of opcode `BYTE` and divided all other opcode results for that platform by the value for opcode `BYTE`.

Interestingly, for both 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 Python client than with the Go client. At the same time, with respect to the environment (Env) category, Go receives the higher fee. This implies that miners who run Python get better profit than Go when executing smart contracts that have opcodes in the Arithmetic category, while miners that use Go perform better than Python miners if the smart contract has more opcodes in the Env category.

VI. 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 [13] states that the most researched

aspects of smart contract based systems is in new applications and in software engineering approaches, while performance and scalability is 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 [2], 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. [14], 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 [15], 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. [12] 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. [12] 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 [16], the authors introduced a tool called (GasReducer) that detects suboptimal 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 paper.

VII. Conclusion

In this paper 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 two different clients, Python-based PyEthApp and Go-based Go-Ethereum and conducted experiments for these two clients on different machines and operating systems.

The work demonstrates the feasibility to benchmark opcodes, discussing both design and implementation. The results obtained from OpBench establish if the award obtained when executing smart contracts is proportional to the cost of executing. This is important for several reasons described in the paper'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.

Our results 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 Go client outperforming the Python client.

Future work should aim to expand on the reported experiments, to further compare across clients, operating systems and CPU specification. In addition, 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 will be made freely available to others.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] G. Wood, "Ethereum: *A secure decentralised generalised transaction ledger*," *Ethereum Project Yellow Paper*, 2018. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[3] M. Alharby and A. Van Moorsel, "The impact of profit uncertainty on miner decisions in blockchain systems," *Electronic Notes in Theoretical Computer Science*, vol. 340, pp. 151–167, 2018.

[4] 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.

[5] V. Buterin, "Transaction spam attack: Next steps," 2016. [Online]. Available: https://goo.gl/uKi9Ug

[6] A. Narayanan, J. Bonneau, E. Feltem, A. Miller, and S. Goldfeder, *Bitcoin and Cryptocurrency Technologies-A Comprehensive Introduction*. Princeton University Press, 2016.

[7] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *White paper*, 2014.

[8] "PyEth client." [Online]. Available: https://github.com/ethereum/pyethereum

[9] "Measure execution time." [Online]. Available: https://docs.python.org/2/library/timeit.html

[10] "Benchmarks." [Online]. Available: https://golang.org/pkg/testing/

[11] "Official Go implementation of the Ethereum protocol." [Online]. Available: https://geth.ethereum.org

[12] T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, X. Luo, M. H. Au, and X. Zhang, "An adaptive gas cost mechanism for Ethereum to defend against underpriced DoS attacks," in *International Conference on Information Security Practice and Experience*. Springer, 2017, pp. 3–24.

[13] 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: IEEE, 2018.

[14] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "Blockbench: A framework for analyzing private blockchains," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 1085–1100.

[15] A. Aldweesh, M. Alharby, E. Solaiman, and A. Van Moorsel, "Performance benchmarking of smart contracts to assess miner incentives in Ethereum," in *Dependable Computing Conference (EDCC), 2018 14th European*. IEEE, 2018.

[16] P. Zheng, Z. Zheng, X. Luo, X. Chen, and X. 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*. ACM, 2018, pp. 134–143.