ISEngineering

Technische Universität Berlin
Fakultät IV
Institute of Software Engineering and Theoretical Computer Science
Information Systems Engineering

Supervisor: Jacob Eberhardt
WS 2019/20

OptiBench

Cloud Prototyping

Ankush Sharma
Stefanus Ardhito Prasetya
Karim El Tal
Fayan Leonardo Pardo Ladino
Santosh Dhirwani
Xhorxhina Taraj

2020-02-27

Contents

In Ethereum blockchain networks, throughput is a key metric to measure performance. In this project we aim to achieve maximum throughput given configuration values like the number of nodes and workload in private Ethereum networks following a pragmatic approach, tweaking the block interval and block gas limit parameter values. In order to find this maximum throughput, we propose a system design and an algorithm as contributions. We implemented a proof of concept to evaluate our contributions and even though it manages to get the maximum peak, these can be further improved. We also propose ideas to further improve our solution in the future.

1. Introduction

Since Nakamoto's paper in 2008, blockchain has become a very hyped technology, which has gained a lot of interest and attention from different domains, use cases, and enterprises (Beck, Avital, Rossi, & Thatcher, 2017) (Hughes, Park, Kietzmann, & Archer-Brown, 2019) (Konstantinidis et al., 2018). Blockchain is a special kind of data structure that performs and stores transactions in blocks (Thakkar, Nathan, & Viswanathan, 2018) (Gupta & Sadoghi, 2018). Each block is connected to a previous block with a timestamp and a hash link, creating like this a chain of blocks (Narayanan, 2016). The blocks provide data records, which cannot be altered retroactively without changing all the following blocks, which will require a consensus of the network majority (Malik, Manzoor, Ylianttila, & Liyanage, 2019). Blockchain can be considered as a distributed storage system where entities or also called the nodes of the networks, do not trust each other and they can rely on each other completely decentralized also containing a complete replica of the ledger (Dinh et al., 2017) (Thakkar et al., 2018). The process of creating a new block is called mining and the validity of a record has to be confirmed and when the majority agrees on it, meaning they reach a 'consensus' in a democratic way, which makes the system trustworthy and more strong, even if there are potential malicious attacks from outside (Weber et al., 2017)(Malik et al., 2019). According to Malik et al. (2019), blockchain protocols can be classified into the following types:

- **Public blockchain**: A blockchain where everyone can join, meaning that anyone can read, write, participate and may submit transactions to it (e.g. Bitcoin and Ethereum)
- **Private blockchain**: A blockchain places restrictions on who is allowed to participate in the network and who can submit transactions. (e.g. Hyperledger Fabric, private Ethereum)
- **Permissionless blockchain**: Anyone is allowed to create blocks (Malik et al., 2019)(Thakkar et al., 2018).
- **Permissioned blockchain**: Only entities included in a predefined list are allowed to create blocks (Malik et al., 2019)(Thakkar et al., 2018).

Some characteristics of the public blockchain are unsuitable for certain business goals. Therefore, a private blockchain concept was introduced to allow enterprises to use blockchain technology. In these blockchain types, a control layer runs on top of the

blockchain and manages the actions performed by the allowed participants (Pilkington, 2016). As a result, a private blockchain is more centralized in nature because only a small group controls the network (Dinh et al., 2017). They only allow certain entities to have access to a closed network. This means that a few participants whose identity is known and authenticated cryptographically can take part in the network. This leads to less time for the network to reach a consensus (Dinh et al., 2017) (Pongnumkul & Thajchayapong, 2017). Therefore, more transactions can take place. Enterprises aim also to be able to bring trust among untrusting parties since the credibility of a private blockchain network relies on the credibility of the authorized nodes (Thakkar et al., 2018). Further, enterprise applications work with very complex data models which can be supported using smart-contracts (Dinh et al., 2017) (Thakkar et al., 2018). While working with private blockchains for their business goals, they intend to get the best performance. However, constantly improving means for them to choose the right values of the parameters in private blockchains that can be adjusted to affect the level of throughput and latency.

Our paper focuses on private blockchain Ethereum with Proof of Authority (PoA) and how to achieve its best performance depending on throughput and when there are given certain parameters. Ethereum provides various configurable parameters such as block gas limit, block interval, workload and number of nodes, which according to Schäffer et al. (2019) they impact throughput noticeably. On the other hand, deciding for one benchmarking tool from the rest and then running it can be a very tiring, long and expensive process for any experimenter. During our research, we noticed that not every benchmarking tool provides explicit help and walk-through on how to benchmark a blockchain. The dependencies themself can take a long time to install, which makes the whole process even more uncomfortable.

We notice that even though well-established blockchain platforms have already been adopted to meet the demands of new applications and business requirements, there are still many challenges that lie ahead for the adoption of private blockchain platforms. Hence, the main challenges in setting up an efficient private blockchain network such as private Ethereum with PoA are: enable the automation of the System Under Test deployment even in multi-regions setups; create a benchmark plan to find through experiments the maximum throughput possible; monitor and collect the results to analyze them to offer the best configuration to obtain maximum throughput.

1.1. Contributions

To find a solution to the previously mentioned challenges, we advance the state-of-the-art with two main contributions:

1. System design of a tool that automates the benchmarking of private ethereum blockchain with PoA. Given certain configuration parameters, the tool automates the private Ethereum System Under Test (SUT) building, configuration, and the benchmark to find the block interval and block gas limit values that offer the maximum throughput possible. We choose Proof of Authority as a consensus mechanism with the algorithm of Clique, so any authorized signers can create new blocks at any time.
2. An algorithm to find maximum throughput; Given certain SUT configuration parameters and following a trial and error approach, the algorithm finds the block interval and block gas limit values that offer the maximum throughput.

The rest of the paper is organized as follows: The 2nd Section presents the background of our motivation and approach. The 3rd section describes the related work. The 4th and 5th sections explain our two contributions. Section 6th and 7th present the implementation and results we obtain. Next, we deliver our opinion on future works (Section 8), what could be done more, also how we organized our team (Section 9) and finally we conclude our paper with the conclusion (Section 10).

2. Background

In this section, we provide relevant information and definition of private Ethereum, Proof-of-authority and the blockchain benchmarking tool Hyperledger Caliper that we use for our study. Ethereum is an open-source, public, blockchain-based distributed computing platform allowing the user to define any complex computations in the form of smart contracts written with a built-in Turing-complete programming language (Hu et al., 2019) (Pongnumkul & Thajchayapong, 2017) (Malik et al., 2019). Ethereum was the first blockchain-based protocol to introduce smart contracts in 2015, which are autonomous agents for transactions among accounts. Once deployed, the smart contract is executed on all Ethereum nodes as a replicated state machine (Neiheiser, 2020). What encourages the users to run the software is to get Ether, which is a digital currency. When new blocks are being created, Ethereum uses transaction fees also called gas consumption

to prevent any cyber attack (Malik et al., 2019). The Ethereum transaction, in other words, is the amount of gas consumed multiply by gas price (Buterin, 2014).

Since this software is open-source, it allows software developers to download and configure the network to be private according to their needs and goals, where computer nodes that participate are those that are granted permission only. As stated by Dinh et al. (2017), public blockchain Ethereum uses the same consensus protocol as 90 % of public blockchain systems do (Proof-of-Work), though with different parameters. PoW is time-consuming and computationally expensive, making unfavorable for applications such as banking and finance, leading to an absence of privacy and security controls on data in the permissioned blockchains (Dinh et al., 2017) (De Angelis, Aniello, Lombardi, Margheri, & Sassone, 2018). But when the operating environment is more trusted, permissioned and private blockchains rely on message-based consensus schema, rather than on hashing procedures. This leads us to Byzantine fault-tolerant (BFT) algorithms, which have been investigated for permissioned or private blockchains to surpass PoW while providing fault tolerance (Tseng, 2016). A new family of BFT algorithms is Proof-of-Authority (PoA) which has recently drawn attention due to offering better and higher performance and having toleration to faults, requiring as well fewer message exchanges (De Angelis et al., 2018). Two well-known clients for the private setting of Ethereum, Parity, and Geth do use it. De Angelis et al. (2018) states that PoA was originally proposed as part of the Ethereum ecosystem for private networks and it is implemented into the clients: Aura and Clique.

PoA algorithms works on a set of N trusted nodes called the authorities, where each one of them has a unique id and a majority of them is assumed trustworthy, namely around (N/2 + 1) (De Angelis et al., 2018). These nodes run a consensus to order the transactions given by clients. The consensus in PoA algorithms is built on a mining rotation schema, which is a method that distributes the responsibility of block creation among authorities in a fair way (De Angelis et al., 2018) (Group & Garzik., 2015). Time is divided into steps, each of which has an authority chosen as a mining leader. Both PoA implementations work differently: they both have a first-round where the new block is proposed by the current leader (block proposal); then Aura requires another additional round (block acceptance), while Clique does not (De Angelis et al., 2018).

We mention in the introduction that measuring performance in blockchains means we need to evaluate its metrics, such as throughput, latency, and scalability. The focus of our paper is maximizing throughput, which Dinh et al. (2017) define it as the number

of successful transactions per second. The parameters that can affect throughput in private Ethereum are the type of workload or the so-called smart contract, network size or in other words the number of nodes, block gas limit, and block interval. In Ethereum, the block size is bound by how many units of gas can be spent per block. This limit is known as the block gas limit. Ethereum miners determine what the maximum gas limit should be by signaling it to the network each block (Schäffer, Di Angelo, & Salzer, 2019). Block interval is defined as the time between two succeeding blocks or the average time interval it takes for the network to generate one extra block in the blockchain (Schäffer et al., 2019) (Thakkar et al., 2018).

To better understand and measure the performance of blockchains, there are a lot of tools offered (e.g. Germlin, Hyperledger Caliper, Whiteblock Genesis, etc). Some of them are open source projects where everyone can contribute to improve and update, others are Software as a Service (SaaS), but mostly they are in-house solutions.

Hyperledger Caliper is a more specialized solution. Caliper supports several blockchains at once, such as the Hyperledger family (Fabric, Sawtooth, Iroha, Burrow, Besu) as well as Ethereum and the FISCO BCOS network. Hyperledger Caliper produces reports containing several performance indicators, such as TPS (Transactions Per Second), transaction latency, resource utilization, etc (Hyperledger Caliper, 2019). The key to why multiple blockchain solutions can get integrated into the Caliper framework is its adaptation layer component. An adaptor is implemented for each blockchain system under test (SUT), the adaptor is responsible for the translation of Caliper NBIs into corresponding blockchain protocol (Hyperledger Performance and Scale Working Group, 2020). As explained in their official website, Caliper NBI is a set of common blockchain interfaces, which contains operations to interact with a backend blockchain system, for example, to install smart contracts, invoke contracts, query state from the ledger, etc. The NBIs can be used for upstream applications to write tests for multiple blockchain systems.

At last, after getting an important overview of private Ethereum, PoA and Hyperledger Caliper we state our research question: How can we systematically experiment with a given workload, number of nodes, block interval and block gas limit find the maximum throughput of a given private Ethereum with PoA blockchain?

The next section will provide us further insight into different studies related to our problem and challenges.

3. Related Work

In this chapter, we will discuss the related work that addresses our research question entirely or partially. Moreover, we will compare and contrast current state-of-the-art blockchain benchmarking tools: Blockbench, Fetch-Block and Hyperledger Caliper against our novel tool OptiBench.

Thakkar et al. (Thakkar et al., 2018) benchmarked Hyperledger Fabric using Fetch-Block as a tool by tweaking Block Size, Endorsement Policy, Channel, Resource Allocation and Ledger Database and collected output metrics such as throughput and latency. The paper did not address benchmarking Ethereum and did not propose any systematic method to tweak more than one parameter to analyze them with respect to each other. Moreover, Fetch-Block does not address the automation challenge we are solving. By automation, we mean, SUT infrastructure automation, installation of dependencies, ethereum client installation, workload modelling, and result collection.

Dinh et al. (Dinh et al., 2017) presented Blockbench as a framework for benchmarking Hyperledger Fabric, Ethereum, and Parity. Blockbench can measure performance in terms of throughput and latency. Additionally, they have conducted experiments to evaluate their tool against Ethereum, Parity and Hyperledger Fabric. However, similar to Fetch-Block, Blockben does not provide a degree of automation necessary to complete our challenge. Blockbench takes as parameter the IP address of the hosts where the SUT is to be installed. It assumes that the source code of Blockbench is cloned on the host to perform the blockchain client installations. The dependencies to the system are also not catered for resulting in a poorly reproducible and portable system. Finally, with SSH-based authentication, it accesses the machines to deploy ethereum and client nodes.

Hyperledger Caliper (Hyperledger Performance and Scale Working Group, 2020) is an open-source tool provided by the Hyperledger open-source community that benchmarks blockchain systems such as Ethereum. The white paper(Hyperledger Performance and Scale Working Group, 2020) explains the system design and the result collection and reporting provided by the tool. In contrast to the previously discussed tools, the tool offers an advanced workload modeling feature and the generated reports are human-readable and understandable as their HTML report could be rendered in any web-browser. The tool does not automate the SUT deployment but rather assumes a blockchain system is already up and running.

From our literature review, we can conclude that the existing tools are not sufficient

to solve our research question holistically, which is finding the best parameter for better performance. Tweaking parameters which require several experiments and several deployments. Thus a tool like Fetch-Block, Blockbench or Caliper alone would be inefficient to meet our goals. We can use one tool such as Caliper for workload generation because it provides scalability, machine and human-readable reports and advances workload modelling. However, we will need to build our SUT automation. A summary of our evaluation of tools is in Table 1.

Table 1: Comparison of Different Blockchain Benchmarking Tools

Feature	BlockBench	FetchBlock	Hyperledger Caliper
Infrastructure Automation	NA	NA	NA
SUT Deployment	Automate Ethereum nodes on remote hosts	NA	NA
Ethereum Account Creation	NA	NA	NA
Workload Scalability	Can scale on different threads	NA	Can scale on different threads
Workload Modelling	Rate of transactions can be adjusted	NA	Advanced workload modelling
Result Reporting	Console output and logs	Console output	Understandable HTML reports
Dependency Installation	NA	NA	NA

4. Optibench Algorithm

In this section we explain in detail our thinking process until designing our algorithm and its workflow step by step.

After experimenting with Ethereum we found out that the throughput is dependent on several factors, among them, the number of nodes, the block interval, the block gas limit and the workload(the smart contract transactions deployed in the network). We reached the conclusion that covering all of them to maximize throughput would be overwhelming

and too much work for the amount of time we had. For this reason we decided to assume that the final user, that would use our solution, knows the number of nodes that he can use and also knows the workload he wants to benchmark on his SUT, leaving us with deciding which block interval and block gas limit are the best for maximum throughput with given parameters.

Given that there is no established formula to calculate this maximum throughput in a private Clique Ethereum network, and that there are several factors influencing over it, we wanted to solve this problem following a pragmatic approach. Given a built SUT and a workload we wanted to know with real results where is the throughput peak that the user is searching for. This is the reason why the algorithm we created runs experiments over several configurations until it finds the desired result.

We also know after manually experimenting that certain block interval configurations can cause the Ethereum network to crash, and also that certain block gas limit configurations can cause block gas limit excess errors. We also know that for a given workload we can change the block gas limit to improve the throughput, but once the workload is "satisfied" with a certain block gas limit configuration, any more block gas limit will not improve the throughput. For this reason our solution will experiment by trial and error, taking into account workload crashes and throughput improvement stagnation to make decisions.

The algorithm will have several steps:

1. **Calculate the minimum block interval**: It is important that the algorithm first finds the minimum block interval which, most probable, will have the maximum throughput. In this step it will take a default configured block gas limit to run several block interval configurations until it successfully benchmarks.
2. **Calculate the minimum gas limit for a certain block interval**: The algorithm will, by trial and error, find the minimum gas limit. In this step it will find a working block gas limit and then try to narrow down accurately the minimum block gas limit taking into account the "accuracy parameter" given. The whole behavior can be seen in the figure 2.

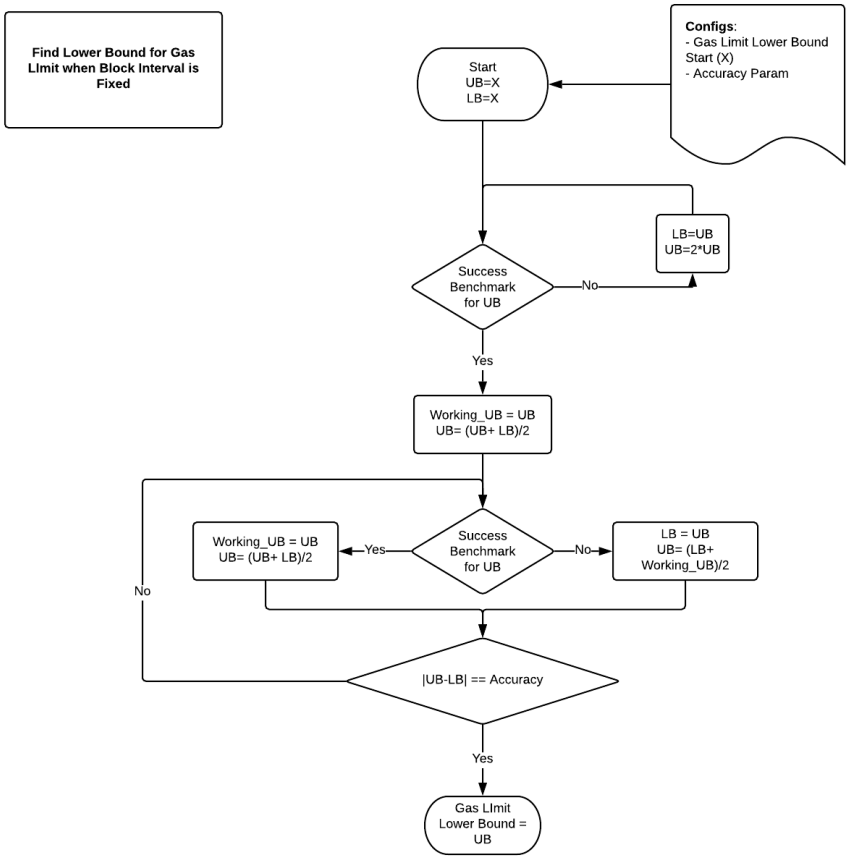

Figure 1: Algorithm diagram covering step 2

3. **Find the peak in a certain the block interval**: The algorithm will experiment with different block gas limit configurations until finding the peak for this certain block interval. It will start experimenting with the minimum block gas limit found in the previous step and every iteration it will increase the block gas limit adding the parameter value "block gas limit step "given by the user. If the throughput between the last experiment result and the previous results from this interval has an improvement of less percentage than the parameter "sensitivity" given or if certain configuration crashes, this iteration will stop and the algorithm will save the maximum throughput found in this block interval. This behavior can be seen in figure 3.
4. **Find the maximum throughput among all block intervals**: After every block interval iteration, the algorithm will compare all the throughput saved for

each block interval experimented. If the improvement of the last throughput is less than the "sensitivity" given, the algorithm will stop experimenting with more block intervals and find the maximum throughput of the peaks already gathered which will be the final result. If the improvement is higher than the "sensitivity" given, the tool will continue experimenting with the next block interval (increases the block interval by adding the parameter "block interval step" given by the user) and start again from step 2. This behavior can be seen in figure 3.

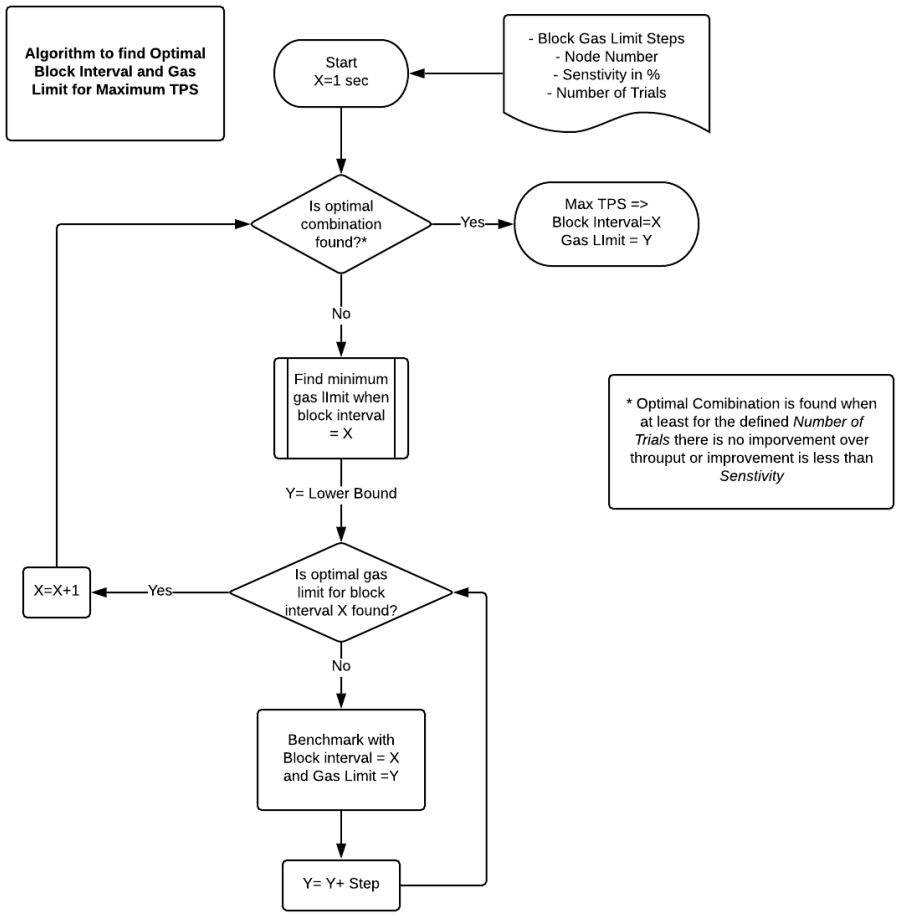

Figure 2: Algorithm diagram covering step 3 and 4

5. Optibench System Design

In this section we explain in detail our system design contribution. We wanted to fulfill the following requirements:

- **Modularity**: We want to bring the possibility to any user to decide if to use our implementation or to freely implement his own configuration. For that reason we decided to have components that are not tightly coupled, enabling interoperability and flexibility.
- **Reproducibility**: Because we are running benchmarks, we want our solution and executions to be reproducible. For this reason we designed a machine readable configuration that the final user can freely edit and which our solution will read and will execute following the configured parameters.
- **Interactivity**: We want our tool to be interactive with the final user and easy to use. For this reason we decided that our tool will have a CLI component and the reports generated will be as interactive as possible.

With these requirements in mind we created the system design seen in figure 1 that has the following components:

- **Configuration**: In this module the user will be able to give configuration parameters that the tool will read and use when executing. The idea is to have configuration for the SUT infrastructure, for the algorithm execution and for the Ethereum SUT.
- **Runner CLI**: This module will be the main executor of the solution. It will include the interactive implementation with which the final user will communicate. This module will include the algorithm explained in the previous section and will communicate with the rest of the main modules. It will communicate with SUT Builder module whenever the tool needs to build again the SUT, it will communicate with the Workload module whenever it needs to send workload to the SUT and needs to benchmark and it will communicate with the Analyzer module to get the last throughput achieved and to aggregate all the results gathered.
- **SUT Builder**: This module will take care of communicating with the cloud provider platform to build the SUT. By reading the user configuration and the parameters received from the Runner CLI it will automatically destroy the previous SUT (if any) and build a new SUT with the given configuration.
- **Workload**: This module will take care of running the workload and benchmark. It will receive parameters from the Runner CLI and benchmark and generate results that will be read by the Analyzer.

- **Analyzer**: This module will read the results generated by the workload, get the throughput values and aggregate all the results gathered during the tool execution. It will also generate the final report with the information gathered.

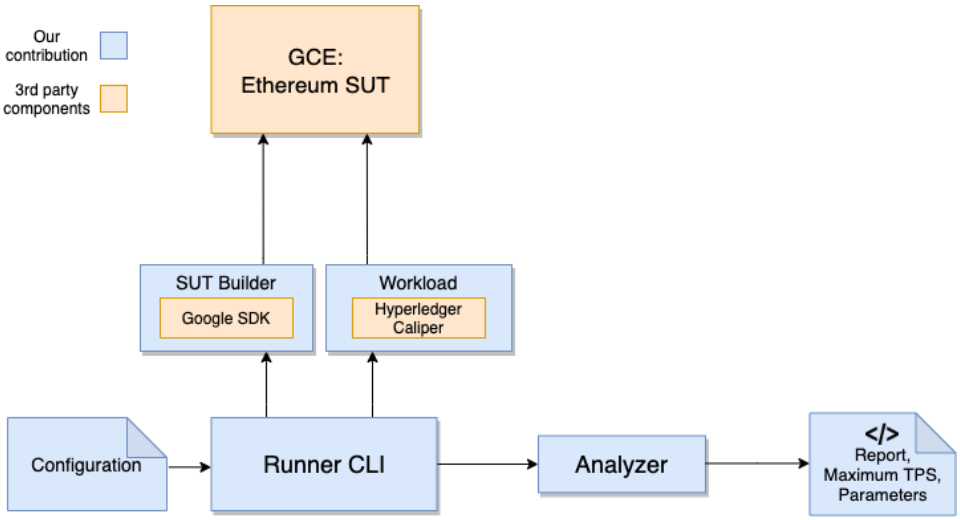

Figure 3: System Design

6. Implementation

After creating the algorithm and system design, we decided to implement them in a proof of concept tool. In this section we will explain the implementation decisions we took during this process.

The requisites of the implementation and instructions to run can be found in the appendix section A and B.

6.1. SUT Module

In this module we needed to implement the necessary logic to build the infrastructure, install and manage the SUT. In our case we decided to build the SUT in a public cloud platform. We chose Google Cloud Platform among the available options mainly because we could use trial credits to implement our solution and because it is well supported and documented. Google also offers the Google Cloud SDK that contains a huge library of commands to manage their platform, making our whole automation implementation easier.

Our first approach was to create container images and orchestrate them with Kubernetes Engine to build and run the SUT. However we had to change our infrastructure implementation to Virtual Machines because we faced problems when automatically building the network with containers as nodes could not communicate between them and we did not have enough skills in the team to make it work before the project deadline.

To run Ethereum nodes, we chose the Geth client. It is a command-line interface for running a full Ethereum node implemented in Go. We chose Geth over the rest of options because again it has a clear documentation, a huge community and support. Along with Geth, the tool also uses Puppeth to create the genesis file.

Moreover, we decided to give to the final user the option to configure nodes in the same region or in different regions.

The main logic of this module was implemented in the "deploy-sut.sh" script. This script will receive as arguments the number of nodes, block gas limit and block interval. Along with some configuration parameters from the configuration file, it will build first the infrastructure of the SUT in GCP, second it will create the bootnode, third create in each node VM the accounts that will be able to sign, fourth it will create locally the genesis file and send it to the VMs to at last run the geth processes in each machine. This script will also print in a file the machines IPs and Ethereum accounts that will be used by the benchmark tool.

It is also worth to mention that this script expects a template with Ethereum installed in the GCP project. In order to prepare this we created the script "create-template.sh" which would be automatically run by the prerequisite script and we also created "create-template-multi-region.sh" to prepare multi-region templates but is still not integrated in the prerequisite script.

6.2. Workload Module

In this module we needed to implement the necessary logic to run workload over the SUT. As already explained in the Related Work section, after comparing different benchmarking tools we chose Hyperledger Caliper for our implementation. It is an open-source tool that provides advanced features like workload modeling. The HTML reports that are generated by the tool are very easy to read and understand. The reports can be viewed using any web-browser. Hyperledger Caliper supports Ethereum it can be configured to

add several contracts as workload. Moreover the benchmark plan (functions to be called, number of transactions, transaction rate, etc.) can be defined in their configuration. For this reasons, we decided to use Hyperledger Caliper in this module not only to run the workload, but also to measure the experiments.

For our implementation we decided to use one simple contract transfer with 3 functions. This means that the reports generated by Hyperledger Caliper show the results for three types of transactions: open, query and transfer. We wanted to focus only in the transfer function throughput for our evaluation because the other two functions did not offer meaningful results and the results had mostly constant values.

The main logic of this module was implemented in the "run-caliper.py" script. This scripts will read the configuration file to know the number of tries of the benchmarking tool and will run Hyperledger Caliper installed previously with NPM. The result of the benchmarks are installed in the reports folder for a further analysis later in the execution.

6.3. CLI Module

The main idea behind this module is to implement the CLI interactive tool that the user would use to run our tool. Due to our iterative work and implementation decisions we also implemented in this module the algorithm logic which manages the execution flow of the tool.

During the initial experimentation's, we observed that the tool execution took a very long time to finish. On further observation, we saw that the step where the algorithm calculates the minimum block gas limit for a block interval was one of the reasons of this long execution time. We also observed that the minimum block gas limit between different block intervals, if not equal these were quite close. In order to run enough experiments before our project deadline, we decided to improve the algorithm by taking into account the previous block interval minimum block gas limit as starting point to save some time. This change improved the tool execution time significantly but still further improvements can be done.

The main logic of this module was implemented in the "main.py" script. This script will be executed by the final user and it will apply the algorithm and call iteratively the scripts of the SUT and Workload module as well as "get-last-throughput.py" of the Analyzer module. Once the algorithm has finished it will call the Analyzer module to generate the final output.

6.4. Analyzer Module

In this module we had to implement the logic to analyse the results obtained from the benchmark tool.

Because Hyperledger Caliper generates reports in HTML we needed to parse these to obtain the throughput values. We decided to aggregate all the experiment results generated and create one meaningful interactive report with Python modules such as Pandas and Matplotlib. The final report is generated in a Bootstrap HTML template.

The main logic behind this module was implemented in the "aggregate-html-reports.py" which will aggregate the results and generate the CSV files as well as the final HTML report. Also "get-last-throughput.py" is key in this module, it will be called iteratively by the CLI module to get the last benchmark throughput from the reports generated with the Workload module.

7. Evaluation

In this section we explain in detail the evaluation we did for to find out if our contributions are useful to reach our goal. To evaluate, we first implemented a proof of concept tool as detailed in the previous implementation section and afterwards we run in our implementation experiments and later analysed the results seen. Here we will explain the configuration of our evaluation, the metrics we observed to answer our question, the plots we used to analyse, the limitations we encountered, the results obtained and the observations we concluded from our results.

7.1. Experiment Goal

The goal of our experiment is to automate the process of benchmarking private ethereum blockchain with different variables such as gas limit and block interval by repeating these experiments multiple times to reach a conclusion of finding the maximum throughput for each type of contract and configuration. Based on our results, we want to guide the blockchain architect in an enterprise to implement private ethereum with the best values of block interval and gas limit for a blockchain solution.

7.2. Configuration

The configuration details for this evaluation are listed in the experiments appendix section. To sum up we decided to make our evaluation with 2 experiments: one of these experiments would run the tool configured to build nodes in one region and the other experiment would run the tool configured to build nodes in 3 different regions. Both of them would still share the tool, sut and workload configuration as well as the number of nodes of the network.

For the tool configuration, we decided that in order to have meaningful results, the tool must run at least 10 benchmark tool successful executions with at least 3 execution retries in case of unexpected failures.

For the SUT configuration, as our implementation has been designed around GCP, nodes have been created on Google Compute Engine Virtual Machines with Ubuntu 18.04 operating system and each Virtual machine of n1-Standard-1 Type and had 1 vCPU with 3.75 GB memory.

For the workload configuration, we have used a simple smart contract with a transfer function to test the throughput for a simple account transactions. As illustrated in Fig 4, this transfer function allows to send some amount of money from one account to another.

```
pragma solidity >=0.4.22 <0.6.0;

contract simple {
    mapping(string => int) private accounts;

    function open(string memory acc_id, int amount) public {
        accounts[acc_id] = amount;
    }

    function query(string memory acc_id) public view returns (int amount) {
        amount = accounts[acc_id];
    }

    function transfer(string memory acc_from, string memory acc_to, int amount) public {
        accounts[acc_from] -= amount;
        accounts[acc_to] += amount;
    }
}
```

Figure 4: Sample Smart Contract with transfer function

Snippet depicted in Figure 5 describes how a transaction would be triggered by in-

voking the transfer() in the smart contract through transfer.js file.

```
module.exports.run = function () {
    const account1 = account_array[Math.floor(Math.random() * (account_array.length))];
    const account2 = account_array[Math.floor(Math.random() * (account_array.length))];
    let args;

    if (bc.bcType === 'fabric') {
        args = {
            chaincodeFunction: 'transfer',
            chaincodeArguments: [account1, account2, initmoney.toString()],
        };
    } else if (bc.bcType === 'ethereum') {
        args = {
            verb: 'transfer',
            args: [account1, account2, initmoney]
        };
    } else {
        args = {
            'verb': 'transfer',
            'account1': account1,
            'account2': account2,
            'money': initmoney.toString()
        };
    }

    return bc.invokeSmartContract(contx, 'simple', 'v0', args, 10);
```

Figure 5: Transfer.js Javascript to invoke the contract

Users can decide the rate at which the transactions are made on the blockchain system. This rate is mentioned in the yaml file shown in figure 6. In this example, to test the performance, each transaction is called 1000 times through the transfer.js javascript associated with each type of transaction.

```
- label: transfer
  description: Test description for transfering money between accounts
  txNumber:
      - 100
  rateControl:
      - type: fixed-rate
        opts:
            tps: 50
  arguments:
      money: 100
  callback: caliper-config/scenario/simple/transfer.js
```

Figure 6: Yaml script to customize number of transactions

Benchmark Tool: In this project, we used Caliper which is a blockchain performance benchmark framework. This allows to test different blockchain solutions with predefined use cases, and get a set of performance test results[1]. The flow of process through the caliper framework is shown in Figure 7.

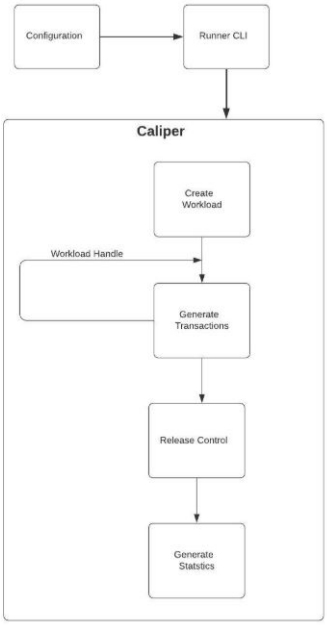

Figure 7: Caliper flow

Caliper receives the desired parameters mentioned in the configuration file through the runner CLI described in Section 3. Using these config files, caliper generates the workload and uses smart contracts which leads to the initialization of transactions. Once the first transaction is made, it consequently moves to the next workload created and continues until all transactions are completed. Once all experiments are run, it releases the control from the blockchain environment and generates statistics for each cycle.

7.3. Metrics

Based on our experiments the following metrics are used to reach a conclusion of the experiments.

- Transaction throughput: We are evaluating the transaction throughput which is the Total committed transactions per seconds through committed nodes. It also

[1]https://hyperledger.github.io/caliper/vLatest/getting-started/

means the rate at which valid transactions are committed by the blockchain SUT in a defined period of time.

- Gas Limit: We are drawing the conclusion about the effect of gas limit upon the throughput, it certainly means to check if there is any relationship between the given gas limit for each block interval over throughput.
- BlockInterval: We are observing the behavior of throughput getting Local maxima for each block interval.
- Maxima of throughput for any block interval (Climax):We are observing the Global maxima of throughput in which block interval and gas limit.
- Execution time: As there has been much needed work done to solve the problem of automation, we are observing the execution time it takes to perform through Optibench SUT.

7.4. Plots

To represent the results python libraries have been used to graphically illustrate the data observed and these graphical information is kept in the form of HTML with CSS pages for easy further analysis and understanding.

In these files, we have added two types of plots to describe results of the experiments. First one is a Bubble plot illustrated in Figure 8. It has block gas limit on X-axis, block interval on Y-axis and corresponding values of throughput are represented as bubbles. The size of this bubble is proportionate to the value of throughput. To know the exact throughput value at a point we can hover the cursor above the bubble.The intent of this plot is to show the changing value of throughput over different values of Block Intervals and Gas Limit.

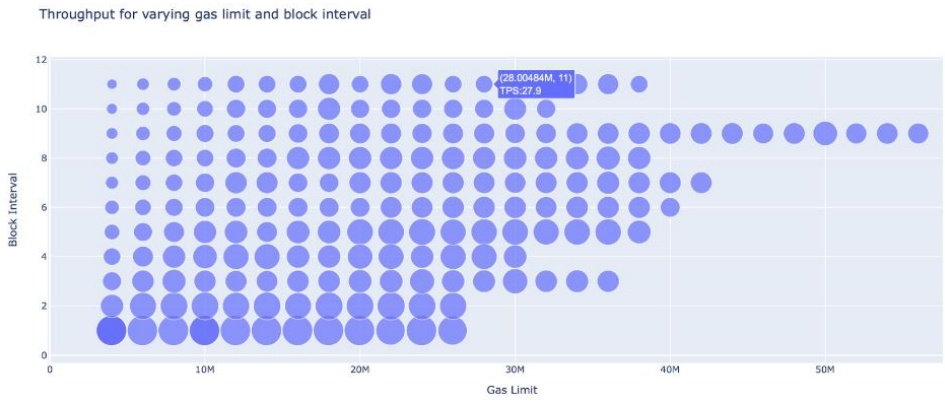

Figure 8: Interactive Bubble plot

The second one depicted in Figure 9 is an interactive line plot. This graph illustrates the value of throughput for each block interval over the different values of gas limit. Using the drop down menu user can select the block interval that it would like to observe or display on the line plot for a particular block interval.The intent of this graph is to show the changing response of throughput in a certain block interval and using the drop down menu allows you to observe values from selected relevant block intervals and compare their throughputs.

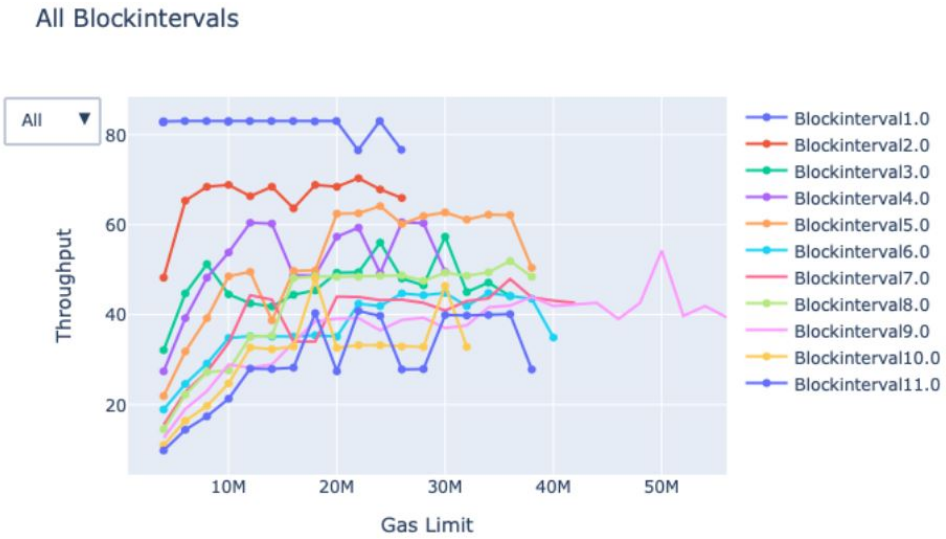

Figure 9: Interactive Line graph

We can also have summary of results from all the experiments performed in that particular run in the form of a table as depicted in Figure 10.

throughput	gasLimit	blockInterval
83.0	13964843.0	1.0
83.0	5964843.0	1.0
83.0	10000000.0	1.0
83.0	11964843.0	1.0
83.0	19964843.0	1.0
83.0	23964843.0	1.0
83.0	17964843.0	1.0
82.9	7964843.0	1.0
82.8	9964843.0	1.0
82.2	15964843.0	1.0
79.9	21964843.0	1.0
78.6	3964843.0	1.0
77.0	4000000.0	1.0

Figure 10: Sample result summary in the form of a Table

7.5. Limitations

Due to a trial account from Google Cloud Platform, experiments were restricted to utilize no more than 8 cores (or virtual CPUs) running at the same time. This limited the capability to test the throughput with a large number of nodes and effects on the blockchain transactions.

7.6. Results

In this section we present and discuss results based on the experiment design presented in above sub sections and appendix C.We visualize the select experiments using line and bubble plots and have found following observations.

- **Single region**: The maximum throughput found is 83 TPS with a block interval of 1 second and a block gas limit of 11.96 million. The execution time was close to 12 hours. As we can see in figure 11, our tool manages to analyse at least 10 benchmark tool executions and, after finding a drop in performance or stagnation it stops executing for that specific interval. We can observe that in our evaluation the maximum peaks of the whole execution are found in the block interval of 1 second. We can also see better in figure 12 that the more we increase the block interval, the more block gas limit steps it needs to find the peak and consequently the stagnation point.

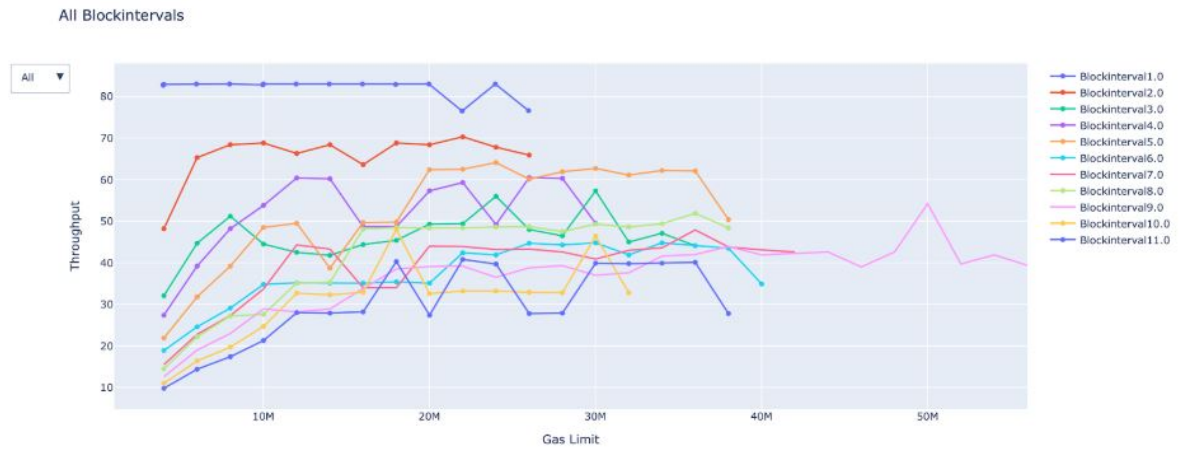

Figure 11: Interactive Line graph: Single

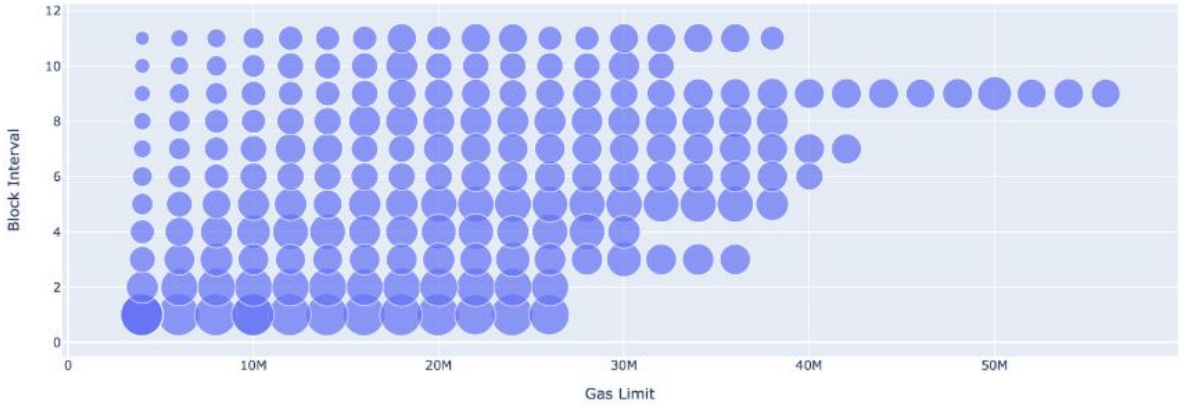

Figure 12: Interactive bubble graph: Single

- **Multi region**: The maximum throughput found is 77.7 with a block interval of 1 second and a block gas limit of 9.99 million. The execution time was more than 13 hours. As we can see in figure 13, our tool again manages to find the peak for each interval and stop after stagnation or drop of performance. In this experiment we can notice that, apart of having less throughput values than in the single region experiment, the throughput results are more chaotic and less "stable". The multi region extra latency may be the cause of these irregular results. In figure 14 we can again see the trend of the single region, the more we increase the block interval the more steps it takes to reach the peak.

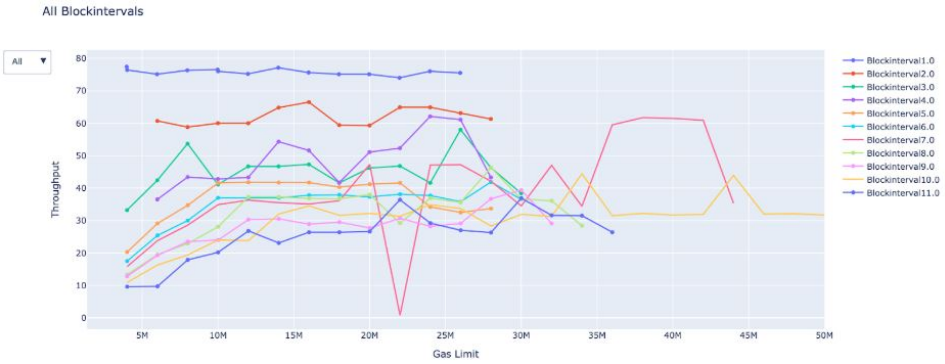

Figure 13: Interactive Line graph: Multi

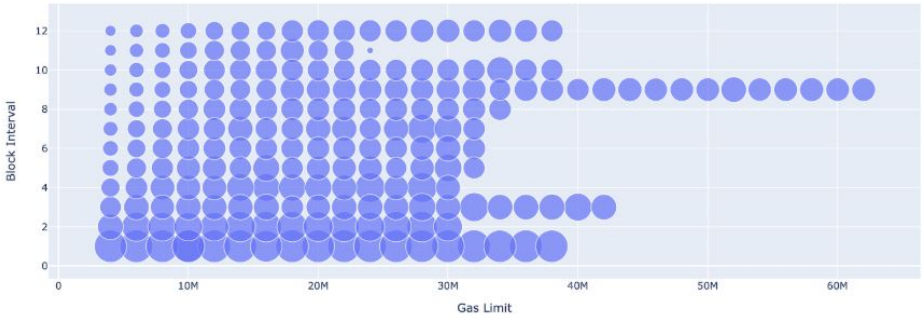

Figure 14: Interactive bubble graph: Multi

7.7. Observations

After analysing the results of our evaluation experiments we saw the following observations:

- **Throughput is inversely proportional to block interval:** our result showing that the less Block Interval we set, it has higher Throughput. As we could see in our plot, SUTs with 1 second of block interval has higher throughput values compare to others.
- **Block gas limit influences throughput more when block interval is increased until it becomes stagnant:** concerning block gas limit, the higher block interval we set then it needs higher block gas limit to get a higher Throughput and consequently to the peak we want to find. This condition was observed before the Throughput become stagnant.
- **Nodes in different regions affect the throughput negatively:** applying

nodes in different regions results in less throughput. The generated report after our experiment shows that the maximum throughput if the nodes are in the same region is 83 TPS meanwhile if the nodes are in different regions 77.7 TPS. This is expected, if we have the nodes in different regions the network latency increases which gives us in general less performance to execute our experiments and to execute transactions.

- **Throughput value is less "stable" in multi-region setups:** we can notice in the plots that the multi region results among the same block interval unexpectedly vary more than the single region results. On some occasions that the Throughput dropped far beyond. It happened on our experiment with the SUT is set to 11 seconds block interval, when it had around 24 million gas limit, the throughput was almost 0 TPS. Meanwhile when the same SUT setup with nodes deployed in the same region had around 40 TPS.

To conclude, our implementation built from our contributions, successfully finds the maximum throughput peak following a pragmatic approach given the workload and number of nodes. It can also stop executing experiments after finding no more improvements or after a complete crash of the SUT setup. However the execution time is quite long, our algorithm can be further optimized (this suggestion will be explained further in the next section), the implementation can be further improved to be more robust and intuitive too.

8. Future Work

Our contributions and implementation are far from perfect, in this section we detail several improvements that can be implemented. The set of improvements that can be done are the following:

- **Optimizing the algorithm:** with the configuration detailed in the evaluation section our tool needs almost 12 hours of execution. Our algorithm can be further improved by predicting workload gas consumption or predicting behaviors with the Ethereum network stats (such as block propagation or pending transactions). We already tried to optimize the algorithm execution time as explained in the implementation section. However, even with this improvement done, the algorithm execution time can be further improved.

- **Container orchestration:** in the current implementation, our SUT is built in Virtual Machines (VMs). Our initial approach was to build the SUT in containers to save execution time, but due to our amount of time and skills we needed to reduce the complexity of the implementation by using VMs. This is the reason why we already have several Docker images ready to use but the orchestration automation is not yet implemented. Images that already created are for creating: bootnode, Ethereum node, and a sealer Ethereum node.
- **A higher level of modularity:** our system design and implementation can be even more modular than the current state. At the moment runner CLI still executing most of the functionality for the analyzer. In the future, it would be better to divide the CLI module into Client for the CLI and Experiment Controller for the Optibench algorithm and the SUT module into Infrastructure Controller to build the SUT infrastructure and Software Controller to install and manage the SUT software.
- **Improve the throughput analysis:** we evaluated our implementation with one simple contract as workload. In the contract, there are three functions: open, query, and transfer. However, we were only measuring throughput from the transfer function since the other functions did not offer meaningful results. A further improvement would be to implement a solution that analyses the contract workload configured and choose the functions to analyse (some query functions offer no meaningful information for our tool). This implementation would also need a procedure to aggregate all the functions throughput and give a general throughput value to measure and compare.
- **User Interface and User Experience:** our current tool is only using the CLI command. It would be better if we can support it by creating a web-based tool. This could help the final user to have a better experience while using Optibench. Then providing more meaningful plots and information of the benchmarking result, could help the user to analyze the report better.
- **Integration with other public cloud services:** at the moment our tool implementation is dependent on the Google Cloud Platform (GCP). The tool's scripts are compatible with GCP service only. In the future, it would be better to have the scripts running with other public cloud providers that commonly used like Amazon Web Service or Microsoft Azure. This will help the user to choose their preference public cloud service to use our benchmarking tool.

- **Fix implementation bugs:** currently, in our tool, there are still bugs or incomplete functions. The GCP flag has a current bug with the multi-region deployment and the monitor script works only with sudo permissions in Linux based systems.

9. Project Organization

In this section we detail how we organized our work as a team in order to fulfill our course goals. Here we explain our important deadlines, team roles, team scheduling, task organization and scheduling, code organization and team communication.

9.1. Important Deadline

Kickoff Meeting (23.10.2019):
In the Kickoff Meeting, all members introduced themselves and Jacob as a supervisor also introduced himself. Team members elaborated on their experience of blockchain technology. Then Jacob gave a brief explanation about this project.
Weekly Team Meeting (every Tuesday at 16.00):
Every Tuesday all team members held a meeting with the supervisor. In this meeting we discussed the current state of the project, sharing obstacles that we found, planning for the next week's milestones, and getting input from the supervisor. All members must attend to this meeting.
Mid-Week Team Meeting (every Saturday at 14.00):
Every Saturday at 14.00 we held a call. The purpose of the call is to update our work in current tasks. All team members shared their progress and obstacles to try to solve problems if any.
Midterm Presentation (11.12.2019):
In December we presented our work to the audience. After presenting our achievements we shared what would be the next steps and the goal of this project. The audience that was attended were members of The Chair of Information System Engineering Technische Universitaet Berlin.
Final presentation (12.02.2020):
The final presentation is the time when our development is finished. Then we presented our tool. In the presentation, we shared how our tool was built and how it works, what are the findings, and discuss the results found with the audience.

9.2. Team Structure

As a team, we all worked as developers in this project. There is only one person who had a double role which acted also as Project Manager. Our team structure was the following:

- Project Manager: Leo
- Software Developer: Ankush, Ardhito, Karim, Leo, Santosh, Xhorxhina

9.2.1. Project Manager

Project Manager had the main task to make sure the project is still on track. To accomplish it, the Project Manager constantly reminding about the sprints and tasks that our team has to achieve. Furthermore, the Project Manager also arranged meetings either for an update within internal members or with the supervisor. Then together with team members, he defined tasks and set the priority of these tasks.

9.2.2. Software Developer

As a Software Developer, the main responsibility is to finish the issue (task) that had assigned in the current sprint. The Software Developer also defined tasks together with all team members. Developers could choose the task which suited best for him/her based on his/her skills. At the end of the development, every Software Developer had to send a pull request to the Github repository for the other developers to review the code. If the code passed the reviews, then it would be merged into the main branch. Otherwise, the pull request should be updated until it meets the task requirements.

9.3. User Stories

After we decided on our roles and the technology stack, we defined our user story. The user story explains what to expect as a user when using our tool. We divide the stories into three main parts: SUT Deployment, Workload Deployment, and Analyzer.

9.3.1. SUT Deployment

In SUT Deployment, as a user, he can build new Ethereum nodes automatically by setting up a configuration and by running the SUT module that takes care of building and destroying the SUT software and infrastructure. The configuration helps the user to decide to automatically deploy the SUT on the desired number of nodes, maximum gas limit, minimum gas limit, determining the nodes in which the region will be deployed, etc.

9.3.2. Workload Deployment

As a user, he expects that this tool can automatically run a workload over the SUT. After the nodes are deployed, then it will run experiments to the network by running a workload and gathering results.

9.3.3. Analyzer

As a user he expects to find their block interval and gas limit combination that gives maximum throughput for the configuration and workload given and from the results gathered from the experiments.

9.4. Kanban

In the first place, our team was using Kanban methodology as our development workflow. At this time our tasks were posted in our AirTable. We define our tasks, then we simply choose it and take another task if already done. After we chose our task the Project Manager will put our name on the assignee. But then we realized this method is not working well for us, so we changed to Scrum Methodology.

9.5. Scrum

After finding out Kanban methodology not working well for us, we chose Scrum Methodology for our development. When using Scrum we defined our goals for each sprint which duration was one week. For every sprint we had list of tasks that should be finished in this period of time. Every sprint, each member takes one or more tasks. At the end of

the sprint, after analysing the task implementations, if these were not finished it would be moved to the next sprint.

We decided to put our tasks, sprints, pull requests, and code stored on our Github page. It shows more transparency for all members and the supervisor.

9.5.1. Task

We list our tasks on Github issues list. Our tasks are created based on our discussion. Then each member has to choose the task to do.

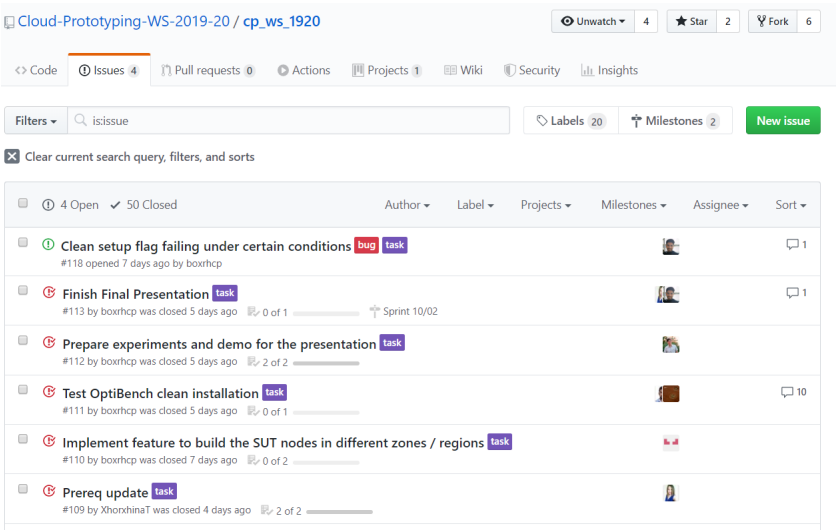

Figure 15: List of tasks on issues page.

Our team task templates. We explained what to do, set acceptance criteria, give the associated story to make sure the task is aligned with our goal, and side notes.

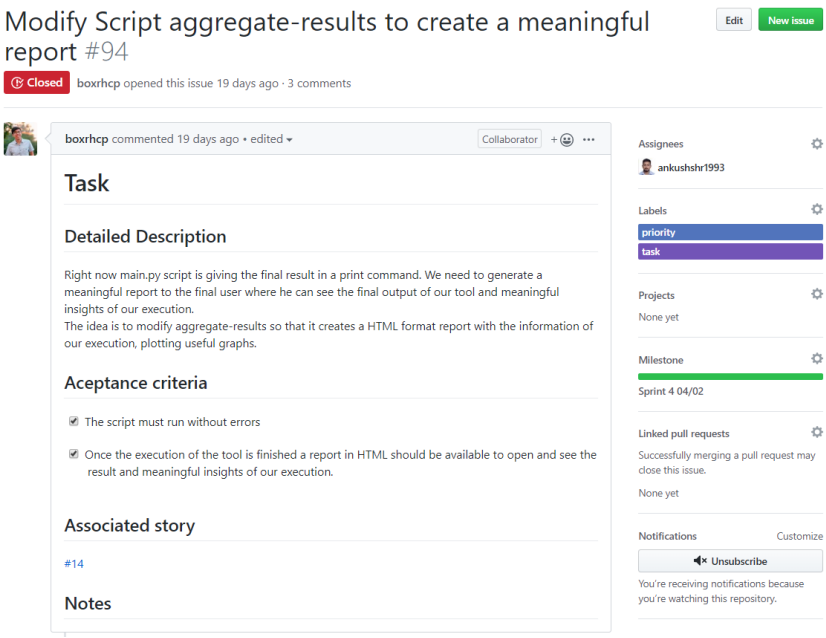

Figure 16: Detail of the task with the template.

To do the task, as a developer we had to make our own branch to keep the master branch clean from unfinished code. The developer would make a pull request before being merged and, after merging with master, close the issue.

9.5.2. Pull Request

After developers finished the task, they needed to create a pull request related to the issue which was needed to merge the working branch into the master branch. The developer had to follow the pull request template created in Github.

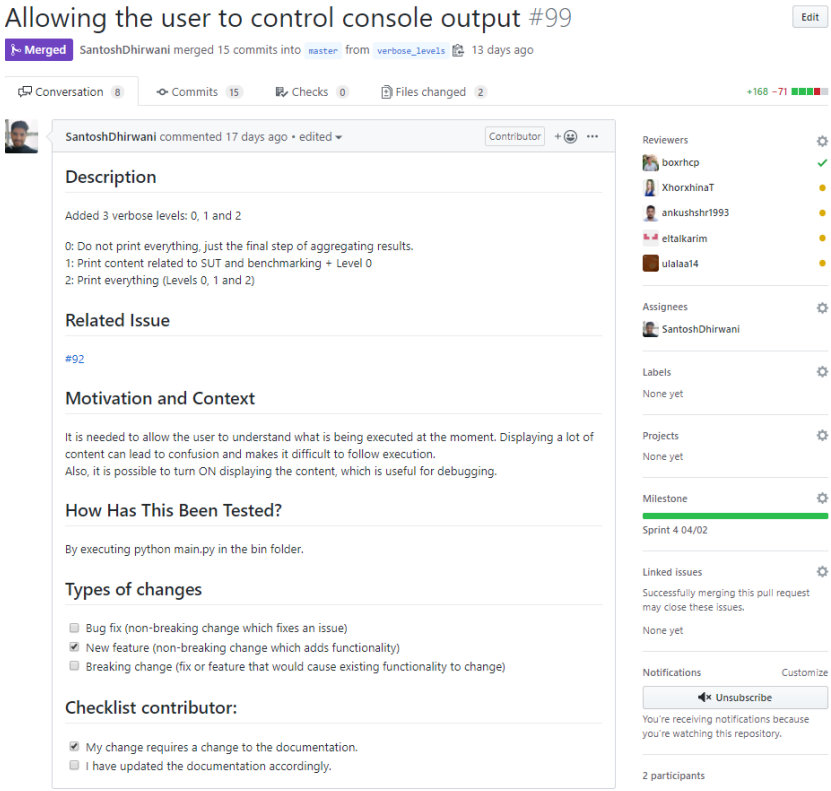

Figure 17: Detail of Pull Request.

Developers had to describe what were the changes or fixes. Then they gave a clear motivation why these changes were needed. As a developer, it was also necessary to describe how the changes can be tested to help the reviewer to test the pull request. Furthermore, they had to tick a checklist of whether the changes were conflicting with the current features or not and checklist if the documentation needed to an update or not.

9.5.3. Review

Pull requests had to be approved by at least one developer before it could be merged. To review the pull request, developers needed to follow the pull request instructions to correctly test it. Then they needed to see the result and check whether it fulfilled the acceptance criteria or not. The reviewer could give comments or ask for a revision if the code was not enough to fulfill the criteria. If the reviewer approved the pull request, the branch could be merged into the main branch.

9.6. Communication With The Team

To manage our communication, we decided to use Slack as the main platform. In slack we have several channels to communicate specific matters :

- Development: this channel is connected with our Github repository to send notification about new/closed issues and new/closed pull requests.
- General: used for communicating with all team members including the supervisor.
- Literature: used for sharing related literature that we found.
- Presentations: this channel was used to share the presentation slides and discuss about them.
- Team Talk: was used for internal team members to communicate, coordinate about the tasks, or to have an internal discussion.

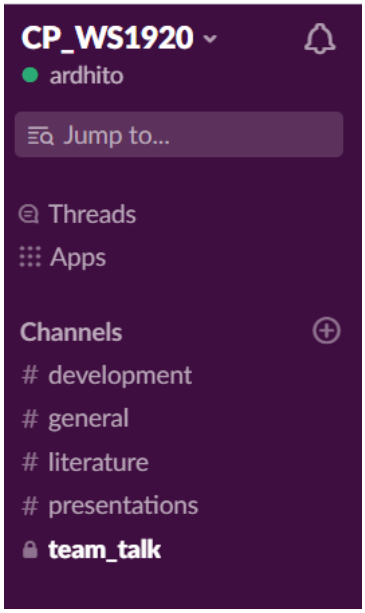

Figure 18: Slack Channels.

9.7. Lesson Learned

From this project, we have learned things about how to work in a team. There were difficulties that we faced.

We found that coordinating amongst the team is quite difficult if we are not commu-

nicating well. Since each team member has their own personal schedule, it was hard to expect us to work together at the same time. Sometimes we have to discuss during our development. Also, sometimes we found out some of us were working on the same task. Therefore, we decided to discuss these coordination issues and solve them on our Slack channel. We centralized our communication platform and we were able to communicate with team members and the supervisor easily.

Moreover defining a task was not an easy job. There has been a long discussion to set priorities for tasks, the whole team needed to clearly agree which tasks were the priority. Also while working on it, we noticed that task descriptions were not clear or the solution implemented did not meet the expectation with the task given. Therefore, we needed to change the wording in the task description. Thankfully these kind of problems could be handled during our midweek team meeting. On Saturday we would also update our progress before the deadline of each sprint to see if there are any issues with the tasks. Discussing our progress helped us to understand the current state of each member. If there was some misunderstanding we could point out the issue and solve it to make a clearer expectation of the task.

10. Conclusion

To sum up, the blockchain technology Ethereum is broadly known, companies want to make the most of it thus some of them want to use it as a private network. Even though it has a big community and public support, it is still complicated to find information to configure an Ethereum network for an specific use case. Benchmarking tools exist to measure and find the ideal configuration but it still can be painful and tiresome to manually benchmark to find the desired metrics. One of these desired metrics is the throughput which can be critical to achieve a better performance.

In order to find the maximum throughput the user needs to experiment with the parameters or configurations that influence it, for example, the block interval, block gas limit, number of nodes and even the workload expected to use.

To ease this process, we stated our contributions: an algorithm that will apply a pragmatic benchmarking plan to find the maximum throughput for given number of nodes and workload by tweaking the block interval and block gas limit; and a system design to automatically perform the experiments and gather the results to give the

maximum throughput.

A proof of concept following these contributions was implemented and was evaluated. The results showed us how this implementation successfully found the throughput peak for a given workload and number of nodes, giving to the user the output of the block gas limit and block interval parameters that would achieve this maximum throughput. We also could observe throughput trends: the less the block interval, the highest the throughput; the more block interval, the more block gas limit influences the throughput; having the nodes in different regions decreased the throughput values compared to having the nodes in the same region; and the results in a multi region setup are less "stable".

From this project, apart of learning about Ethereum, we have learned about how to work in a team. We found coordinating amongst the team is quite difficult if we are not communicating well. Using Slack we are able to communicate within team members and the supervisor easily.

Another thing is defining a task not an easy job. We had to do long discussions to decide which task is our priority. More importantly, writing a task on the issue page had to be clear so there would be no misunderstanding within the members. Somehow, we found out that some tasks needed to rephrase to make it clearer for the assignee.

References

Beck, R., Avital, M., Rossi, M., & Thatcher, J. (2017). Blockchain technology in business and information systems research. *Business and Information Systems Engineering*, *59*, 381–384.

Buterin, V. (2014). Next-generation smart contract and decentralized application platform. *white paper*, *3*, 37.

De Angelis, S., Aniello, L., Lombardi, F., Margheri, A., & Sassone, V. (2018). *Pbft vs proof-of-authority: applying the cap theorem to permissioned blockchain.*

Dinh, T. T. A., Wang, J., Chen, G., Liu, R., Ooi, B. C., & Tan, K.-L. (2017). *Blockbench: A framework for analyzing private blockchains.*

Group, B., & Garzik., J. (2015). *Public versus private blockchains.*

Gupta, S., & Sadoghi, M. (2018). *Blockchain transaction processing.*

Hu, Y., Liyanage, M., Manzoor, A., Thilakarathna, K., Jourjon, G., & Seneviratne, A. (2019). *Blockchain-based smart contracts - applications and challenges.*

Hughes, A., Park, A., Kietzmann, J., & Archer-Brown, C. (2019). *Beyond bitcoin: What blockchain and distributed ledger technologies mean for firms.*

Hyperledger Caliper. (2019). *Tool.* `https://www.hyperledger.org/projects/caliper`.

Hyperledger Performance and Scale Working Group. (2020). *Hyperledger blockchain performance metrics white paper.* `https://www.hyperledger.org/resources/publications/blockchain-performance-metrics`.

Konstantinidis, I., Siaminos, G., Timplalexis, C., Zervas, P., Peristeras, V., & Decker, S. (2018). *Blockchain for business applications: A systematic literature review.*

Malik, H., Manzoor, A., Ylianttila, M., & Liyanage, M. (2019). *Performance analysis of blockchain based smart grids with ethereum and hyperledger implementations.*

Narayanan, J. F. E. M. A. G. S., Arvind; Bonneau. (2016). *Bitcoin and cryptocurrency technologies: a comprehensive introduction.*

Neiheiser, I. G. R. L. e. a., R. (2020). *Hrm smart contracts on the blockchain: emulated vs native.*

Pilkington, M. (2016). *Blockchain technology: principles and applications. in research handbook on digital transformations.*

Pongnumkul, C. S., Suporn, & Thajchayapong, S. (2017). *Performance analysis of private blockchain platforms in varying workloads.*

Schäffer, M., Di Angelo, M., & Salzer, G. (2019). *Performance and scalability of private*

ethereum blockchains.

Thakkar, P., Nathan, S., & Viswanathan, B. (2018). Performance benchmarking and optimizing hyperledger fabric blockchain platform. *n 2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 264-276.

Tseng, L. (2016). Recent results on fault-tolerant consensus in message-passing networks. in int. colloquium on structural information and communication complexity. *Springer*, 92-108.

Weber, I., Gramoli, V., Ponomarev, A., Staples, M., Holz, R., Tran, A. B., & Rimba, P. (2017). *On availability for blockchain-based systems.*

Appendices

A. Requirements to run

The following tools are required for our solution:

- Python 3 and libraries: pandas, lxml, matplotlib, numpy.
- web3: 1.2.6 (version 1.2.0 or above)
- pm2: 4.2.3
- Google Software Development Kit (SDK)
- Geth
- Puppeth
- jq-command[2]
- npm (Node.js)[3]: to run caliper a bigger version than Node.js v8.X LTS is needed
- Caliper[4]

B. Execution

To run the tool, the user needs to first clone our repository on GitHub and then open the repository. On executing the prerequisites.sh script, the user has the environment ready for running OptiBench. The next step is to set the value of parameters in the config file. It can be found inside the config folder. The user executes the tool by navigating to the bin folder and executing the main script. It is the most important script in this tool and is responsible for triggering all other scripts in the tool. It is executed along with 3 flags as:

python main.py [–**verbose**] [–**notbuildsut**] [–**monitor**]

where,

–**verbose**

[2] `https://stedolan.github.io/jq/`
[3] `https://nodejs.org/en/download/`
[4] `https://hyperledger.github.io/caliper/vLatest/installing-caliper/`

It represents the verbose levels. We have defined 3 verbose levels: 0,1 and 2, where '0' will print only the result, '1' will print the execution steps and '2' will print everything (recommended for debugging).

–notbuildsut

It is a flag when enabled, the sut will not be built (or rebuilt) in the Google Cloud project setup.

–monitor

It enables network monitoring by using ethstats.

C. Experiments

In this appendix section we show in detail the configuration parameters used in our evaluation process. This process is explained in the evaluation section. We run 2 different experiments, the common configuration for both experiments is the following:

- Workload:
  - Smart Contract: Simple transfer contract with 2 functions
  - Benchmark Tool: Hyperledger Caliper
  - Number of Transactions: 1000 per function
  - Transaction rate: 100 tps per function
- SUT Configuration:
  - Number of nodes: 3
  - VM: GCE n1-standard-1
- OptiBench Configuration:
  - Block Interval Step: 1s
  - Block Gas Limit Step: 2m
  - Block Gas Limit Accuracy: 10k
  - Sensitivity: 5
  - Minimum number of benchmark tool successful executions: 10

For the single region experiment we just configured 3 number of nodes under the configuration file. For the multi-region experiment we configured the following nodes:

- Node 1: europe-west1-b zone.
- Node 2: us-east1-b zone.
- Node 3: asia-east1-b zone.

The host were our implementation ran our evaluation had the following specs:

- OS: Ubuntu 18.04
- vCPU: 2
- RAM: 4GB
- Memory: 50GB

The json configuration in our tool implemented is at it follows:

```
{
  "tool_config": {
    "maxGas": 15000000,
    "minGas": 2000000,
    "defaultGas": 10000000,
    "gasStep": 2000000,
    "gasLimitAccuracy": 10000,
    "maxInterval": 15,
    "minInterval": 1,
    "intervalStep": 1,
    "numberTrials": 10,
    "sensitivity": 0.05
  },
  "sut_config": {
    "nodeNumber": 3,
    "templateName": "ethtemplate",
    "nodes": [
      {
        "Region": "europe-west1",
        "Zone": "europe-west1-b"
      },
```

```
      {
        "Region": "us-east1",
        "Zone": "us-east1-b"
      },
      {
        "Region": "asia-east1",
        "Zone": "asia-east1-b"
      }
    ]
  },
  "workload_config": {
    "attempt": 3
  },
  "eth_config": {
    "username": "cloudproto",
    "password": "cloudproto",
    "network_id": 123
  }
}
```

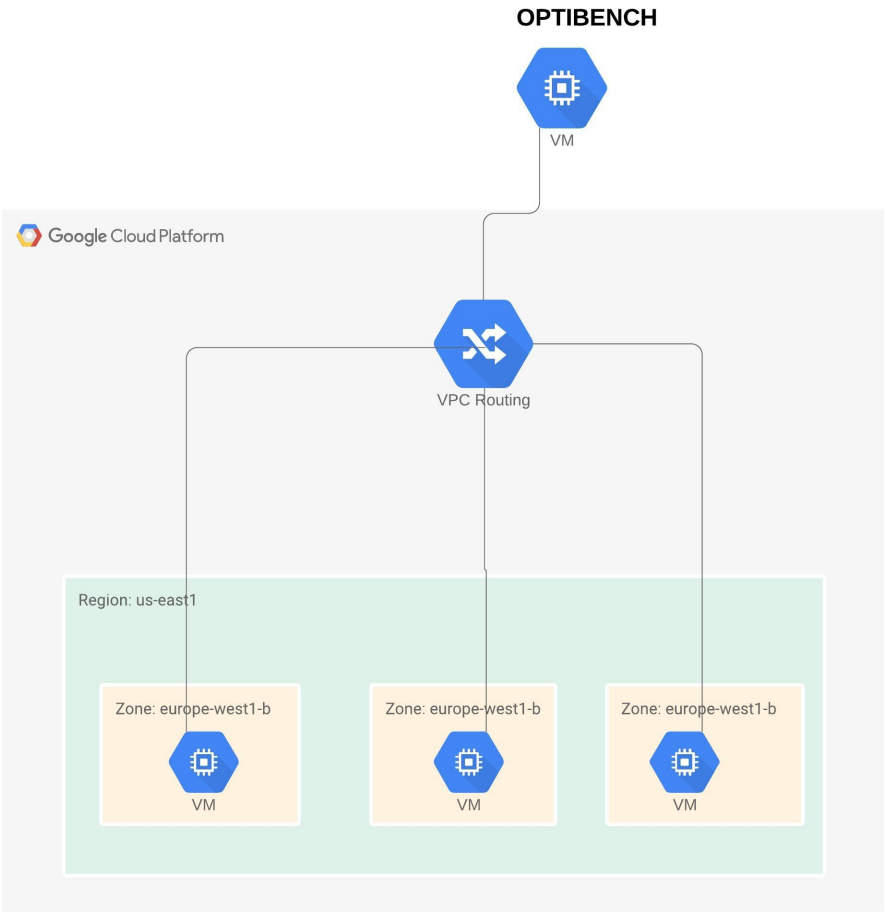

Figure 19: Single Region Scenario

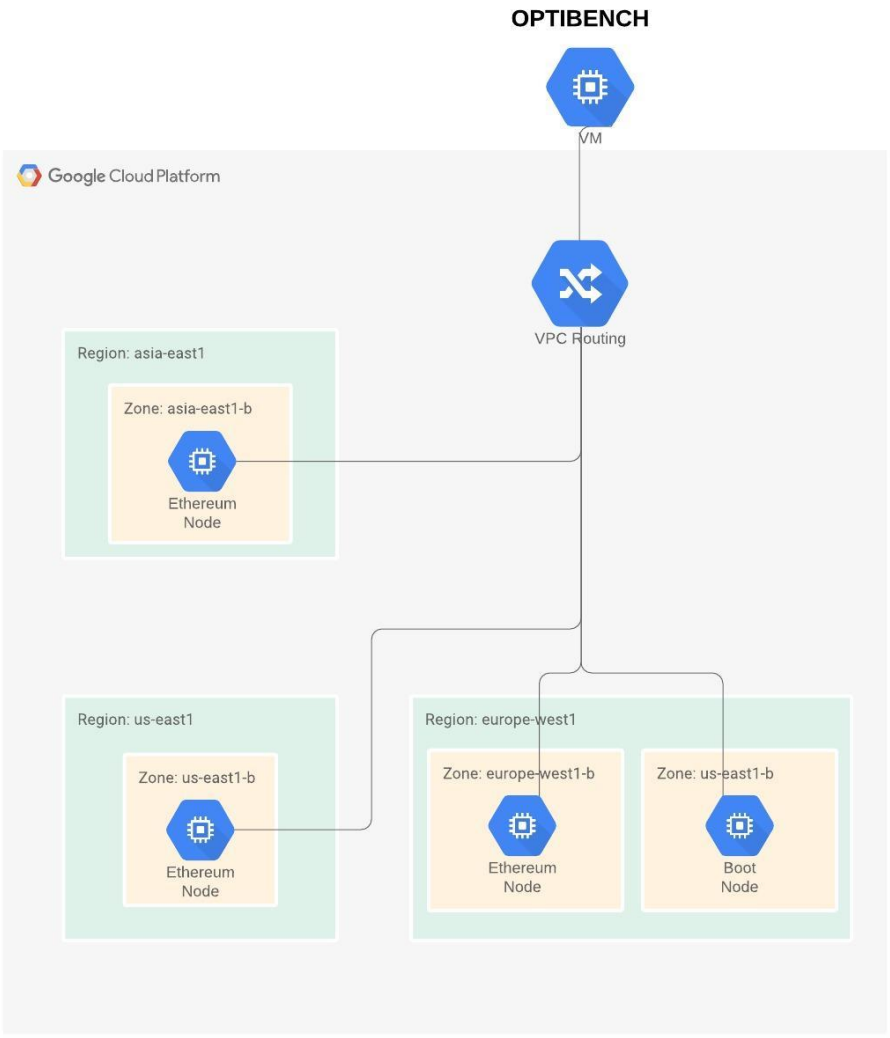

Figure 20: Multi Region Scenario

D. Report Authorship

In this section we will list the authors of the document and the sections written by each of them:

- Ankush Sharma: Evaluation (Goal, Configuration, Metrics, Plots, Limitations).
- Ardhito: Evaluation (Observations), Future Work, Project Organization.
- Leo: Abstract, Optibench Algorithm, Optibench System Design, Implementation

(SUT Module, Workload Module, CLI Module, Analyzer Module), Evaluation (Configuration, Results), Conclusion.

- Karim El Tal: Related Work.
- Santosh Dhirwani: Implementation (SUT Module. Workload Module), Requirements to run, Execution.
- Xhorxhina Taraj: Introduction, Background.