Analyzing Performance in Blockchain-Based Systems

Anuj Das Gupta (anuj@stratumn.com)
Andrew Dickson (andrew@stratumn.com)

11/21/2017

We propose a methodology for analyzing performance in blockchain-based systems, evaluate the existing tools in the space, and define a set of software requirements for a new performance benchmarking tool. We invite input and participation in our ongoing efforts from the wider software development community.

1. Introduction

The scope of our work is to help the blockchain community develop a methodology for performance analysis of blockchain-based software systems with a special focus on Indigo Framework. We hope that this work can help close the performance gap between blockchain-based systems and comparable centralized systems. For further discussion of performance in centralized versus decentralized systems, as well as the general context for this work, see Appendix A.

The structure for the remainder of this paper is as follows:

In Section 2, we define the benchmark as a tool for measuring performance of blockchain-based software systems, and explore each of its component dimensions: layer, metric, and use case.

In Section 3, we examine how comparing benchmark results across blockchain applications can yield useful and actionable insights.

In Section 4, we highlight desirable types of performance reporting in benchmarking tools, and offer some graphical examples.

In Section 5, we analyze two existing open-source blockchain benchmarking tools.

In Section 6, we present the software requirements for the development of our own proposed benchmarking tool.

In Section 7, we conclude and lay out some possible next steps to continue our work.

2. Three Dimensions of Benchmarking

We first define a benchmark as:

The measurement of a single metric evaluated for a single use case at a specified system layer.

Next, we consider each of these dimensions — the system layer, the metric, and the use case — in more detail.

2.1. System Layers

In software engineering, a useful way to understand the functioning of a software system is to divide the system into several different conceptual layers. This abstraction is especially useful in the measurement and analysis of application performance, since it allows us to go deeper into understanding the sources of our performance bottlenecks, and help to optimize the parts of the application individually, in addition to merely optimizing the application as a whole.

In our work, we consider the following nine layers of blockchain applications:

	Layer	**Examples**
1	View	UTXO (BTC), Balance (ETH), Segments (Indigo)
2	Application	Contracts (ETH), Processes (Indigo)
3	Storage	DHT, Databases, CAN
4	Execution	Compilers, VMs, Dockers
5	Consensus	PoW, PoS, PBFT
6	Data Model	Blocks, Transactions, Indexing
7	P2P Propagation	Relay Network, Fast Relay Network, Falcon, Fibre
8	Side Plane	Sidechains, Payment Channels, Hubs, Oracles, Bridges to other networks
9	Network	TCP/IP, UDP

Table 1: System layers

2.2. Metrics

Each of the twenty-six metrics we present offers insight into a different aspect of the target application. We divide the metrics into eight functional categories, corresponding to areas of user interest:

	Category	**Description**
1	Traffic	Classic performance analysis.
2	Storage	How data is stored locally as well as in the network.
3	Execution	Any execution of code, be it locally

		on a client node or on the network level.
4	Structure	The structure of the network. This borrows heavily from Graph Theoretic and Network Complexity analysis.
5	Reliability	The various ways failure is detected, prevented, fixed, and evaluated.
6	Node Setup	Marginal costs of adding a new node to the network.
7	Network Setup and Governance	Costs related to starting a new network from scratch, as well as maintaining an existing network.
8	Economic	Performance analysis considering computational resources as scarce entities.

Table 2: Metric categories

In addition to the above approach to categorization, we recognize that some of our metrics involve different levels of theoretical difficulty, which result in a diversity of implementation costs and strategies, as seen in Appendix B.

Our recommended list of metrics is presented below, along with their functional category and some applicable system layers:

	Metric Name	**Category**	**Applicable Layers**	**What it means**
1	I/O Latency	Traffic	Network	• Time to fetch segments/transactions (based on input parameters/query). • Time to commit segments/transactions.
2	Bandwidth	Traffic	Network	• The data rate that is supported by the network. This is the data for receiving and transmitting transactions, blocks, and metadata.
3	Throughput	Traffic	Network	• Transaction-Rate-Write (Transactions per second) at which the network can confirm transactions. • Data-Rate-Write (Data per second) at which the network can confirm a smaller number of transactions with

				large data payloads. • Transaction-Rate-Read (Transactions per second) at which the network can process read requests. • Data-Rate-Read (Data per second) at which the network can return large data blocks based on a smaller number of requests.
4	Consensus Latency	Traffic	Network, Side Plane, P2P Propagation, Consensus	A transaction is considered confirmed when it is included in a block which is the canonical one according to the network meta-consensus. There, we ask: • What is the probability that my transaction will be included in the next block? • How many blocks do I need before my tx is considered final? E.g. 6 for BTC as it is probabilistic, and just 2 for PBFT as it is deterministic (provided digital signatures behind the votes are secure enough).
5	MemPool Efficiency	Traffic	Side Plane, P2P Propagation	How fast do new or committed transactions travel through the network?
6	Block Size	Storage	Data Model	Disk space occupied by each block.
7	Disk Space	Storage	Data Model, Storage	The cost of storing • the necessary blockchain data for transaction validation. • the blockchain's entire historical data necessary to setup new nodes that join the network.
8	Data Structure	Storage	Data Model	How performant (space and time

				complexity) is the data structure used to record the transaction, e.g., UTXO (BTC), Contracts (Ethereum), and Chainscript (Indigo) — in regards to the operations of • search and filter • sort • merge • map
9	Peak Memory Usage	Execution	Data Model, Execution	RAM utilization (both on- and off-chain) for segment creation
10	Execution Time	Execution	Execution	Time to create a segment, i.e., • Off-Chain: CPU Cycles behind an agent/method execution. • On-Chain: CPU cycles for running the Validation Code (as we do in Indigo wrt the Manifest). This could be compared with the CPU cycles for a similar problem-statement solved by a Smart Contract (ETH) .
11	Availability	Reliability	Network, P2P Propagation, Consensus	• The ratio of uptime to total time.

12	Fault Tolerance	Reliability	Network, P2P Propagation, Consensus	How the throughput and latency change during node failure in one of the several ways:
				• Crash
				  ◦ Fail-stop: Nodes crash and never recover.
				  ◦ Fail-recover: Nodes crash and may at some later date recover from the failure and continue executing.
				• Omission: This is a special case of the previous one. The

				server is replying “infinitely late”.
				• Timing/Performance
				• Transient
				• Malicious
				  ◦ Authentification detectable byzantine failures.
				  ◦ Byzantine or arbitrary failures.
				Byzantine failures ⊃ authentification detectable byzantine failures ⊃ timing/performance failures ⊃ omission failures ⊃ fail-recover failures ⊃ fail-stop failures.

13	Fragility	Reliability	Consensus	• How quickly will the failure of any one node trigger failures across the network? • How likely is it that network malaise will spread?
14	Congestion (both partial and total)	Reliability	Network, P2P propagation	• Measuring Fairness Index. • Identifying choke-points. • Rate of spread of congestion.
15	Network Strength	Structural	Network, Consensus, Storage	• Concentration of nodes in the network.
16	Centrality (Node influence metrics)	Structural	Network, Consensus, Storage	• Centrality is the eigenvector corresponding to the largest eigenvalue for the adjacency matrix describing the network.
17	Network Diameter	Structural	Network, Consensus, Storage	• Longest shortest distance from a node to any other node, across all nodes.
18	Degree distribution	Structural	Network, Consensus, Storage	• The probability distribution of the number of connections each node has with other nodes over the

				whole network. For directed network, this is a two-dimensional distribution, one for in-degree and another for out-degree.
19	Connectivity/ Resilience	Structural	Network, Consensus, Storage	• The minimum number of elements (nodes or edges) that need to be removed to disconnect the remaining nodes from each other.
20	Network Topologies	Structural	Network, Consensus, Storage	The relationship between the nodes, both in terms of: • Organization: Is it a Delegated Consensus, is the governance spoke-n-hub or consortium like? • Infrastructure: Mesh network or trusted hubs.
21	Setup Cost	Setup	Application, Storage	• Time and space (memory and store) it takes a new node to sync up (download and replay/validate) as a relation to the block height and the size of a pre-existing blockchain network.
22	Membership Cost	Setup	View, Application, Storage	• Coordination Cost for network participants to accept the addition of a new member to a pre-existing network.
23	Bootstrapping	Governance	View, Application, Storage, Network	• Cost it takes to start a new network from scratch.
24	Fork Coordination Cost	Governance	View, Application, Consensus	• Resources spent to coordinate between nodes to fork the blockchain for governance.
25	Block Confirmation Probability	Governance	Consensus, Data Model	• Probability that a block will not be redacted.
26	Valuation of a Network	Economic	All layers	• How to value a network as the network scales with increased participation.

Table 3: Performance metrics

2.3. Use Cases

Each use case represents either a single state of the test network, or a set of a test network states. Many use cases involve generating a set of test network states by changing the test network along a single dimension. For a detailed definition of what we mean by "test networks" and an example of a test network please see Appendix C.

	Use Case Dimensions
1	Nodes in the network
2	Delay between nodes
3	Node diversity (cloud provider as well as geographical distribution)
4	Size of blockchain database
5	Computational demands of agent script (off-chain)
6	Nodes' network bandwidth
7	Segment and block size
8	Age of the blockchain (block height)
9	Computational demands of validation (On-chain)
10	Read request frequency for maps
11	Create request frequency for maps
12	Processes per network
13	Participants (per network and per process)
14	Signature schemes (for segment non-repudiation and block validation)
15	Access control levels between maps, participants and operation (read/write)
16	Encryption schemes
17	Proportion of trusted nodes (as influencers)
18	Congestion amongst nodes (influencers vs regular nodes)
19	Proportion of full nodes vs light clients

Table 4: Dimensions along which a test network might be varied

3. Interpreting the Benchmarks

The understanding of a benchmark through the triple-lens of layers, metrics and use cases presented in Section 2 implicitly assumes that we are measuring a single application in isolation. However, if one measures an application in isolation, a natural question quickly arises, namely: *what do the resulting numbers mean*?

Sometimes we will have an intuitive idea about what a "good", "bad" or "reasonable" number would be for a given benchmark, but more commonly we wish to make sense of benchmark results by comparing them with other results.

We see three useful categories of comparison:

	Comparison Type	**Example**
1.	Standards-Based	How do our results compare with an industry-accepted standard?
2.	Inter-Stack Benchmark	How does the performance of technology stack A compare with technology stack B, as a solution to the same high-level problem?
3.	Intra-Stack Benchmark	How does the performance of a given technology stack change as it configured with different options?

Table 5: Categories of benchmark comparison

Having access to these comparisons allows us to make practical decisions based on our benchmark results. For example, we can make a decision about which technological stack to use for our project, or how many servers we will need in our cloud infrastructure.

Because blockchain is an emerging technology without established standards, we focus here on the latter two types of benchmark comparisons.

3.1 Intra-Stack Benchmark Comparisons

(e.g, Indigo-Hyperledger VS. Indigo Tendermint)
Layer: any layer

Indigo Framework can be configured to run on top of one of several different underlying blockchain technologies. As an example, Indigo can currently be configured to utilize either Tendermint or Hyperledger for management of storage, consensus and P2P propagation of the underlying blockchain.

Therefore, an interesting question immediately arises: for a given test network and set of benchmarks of interest, which configuration performs better: Indigo/Tendermint, or Indigo/Hyperledger? Running intra-stack benchmarks against both configurations allows us to answer this question in a quantitative way.

3.2 Inter-Stack Benchmark Comparisons

(e.g. Indigo VS. Ethereum)
Layer: mostly application/view layer

Another type of comparison would evaluate one entire technology stack versus an entirely different technology stack to see how well they each perform at solving the same high-level problem. Note that because any two technology stacks may have significant differences in the details of how lower level layers are structured, this sort of comparison will probably work best with benchmarks that target higher-level layers (like the application or view layer)so that the comparisons can be as meaningful as possible.

As an example of this type of comparison, consider three parties who wish to exchange data without sacrificing privacy. To concretize this scenario, we can record 100 MB worth of specific messages to be exchanged amongst the three parties in a predetermined order, and then enact this message exchange on two different networks of comparable size (say, Indigo and Ethereum, for example). We may then apply metrics such as throughput (how long did the whole exchange take?) and latency (i.e. how long did each message-write take to be confirmed, on average?) to both test networks.

4. Existing Benchmarking Tools

As a part of our performance research, we conducted a survey of existing open-source benchmarking tools in the blockchain space. We found two existing projects, Blockbench and Caliper, both of which support, in theory, plugin-based benchmarking for any blockchain-based software system.

After conducting a broad analysis of the capabilities of Blockbench and Caliper (see Appendix D), we also carried out a focused estimation effort to determine the development time requirements and feasibility of implementing a plugin for one of these two tools for benchmarking Indigo. The results of this analysis can be seen here. Our overall conclusions was that while it would be possible to implement such a plugin with an engineering effort of about 60-70 hours, each tool has significant enough drawbacks that we prefer to pursue own own development effort. We outline the basic software requirements for such a development effort in Section 6 of this paper.

5. Benchmark Reports to Analyze Performance

In this section we discuss two types of useful reports that benchmarking tools may output.

5.1 Single-Use-Case Reports

One interesting type of report, “Single-Use-Case”, considers how the value of one metric changes as we vary some parameter of the test network. Some examples of single-use-case reports that might be included are:

1. Network propagation rate (latency and throughput) vs. block size
2. Block Propagation Time vs block size
3. Throughput (tx/sec) and latency vs. number of nodes
4. Time to commit a segment vs. number of segment-add requests

5. Time to read a map vs. number of map-read requests
6. Execution Time (to validate a segment, to create a new segment) vs. peak memory usage

Graphical examples of single-use-case reports, taken directly from the Blockbench Research Paper, are shown below:

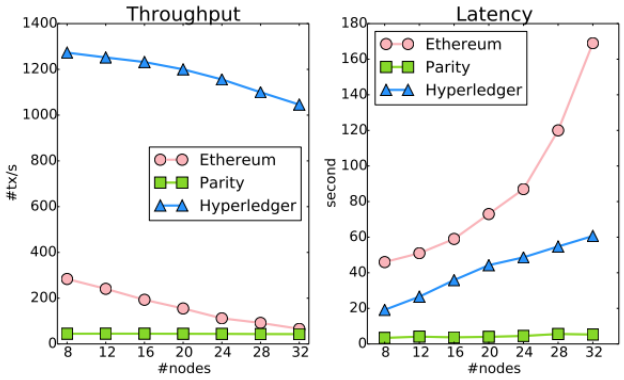


Figure 8: Performance scalability (with 8 clients).

A graph-style, single-use-case report that displays results for multiple blockchains at once. (from the Blockbench research paper)

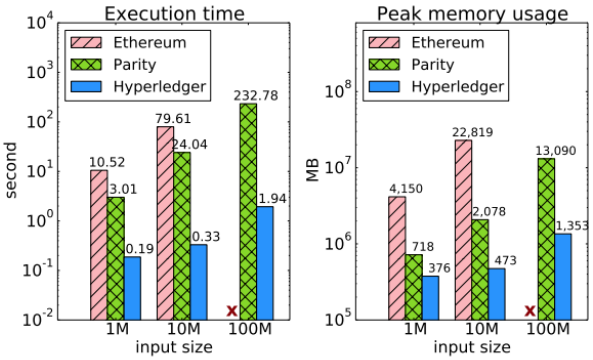


Figure 11: CPUHeavy workload, **'X'** indicates Out-of-Memory error.

A bar-chart-style, single-use-case report that displays results for multiple blockchains at once. (from the Blockbench research paper)

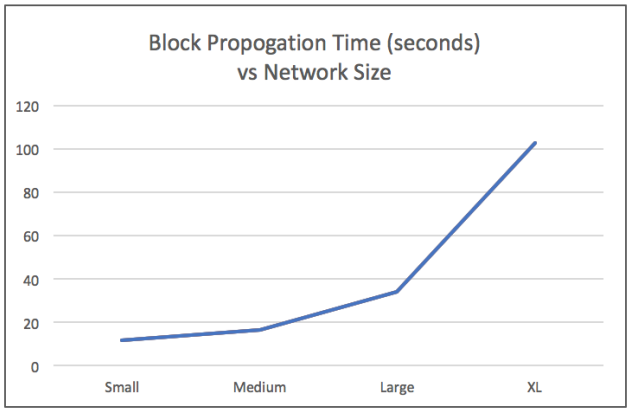
5.2 Scaled Test Network Reports

Because any test that we perform is, in reality, executed against an example/test network that represents an n-dimensional vector of values from the "Use Cases" section above (i.e. the test network has some number of client nodes, a number of server nodes, some block size, a size of the database, available bandwidth, etc.) the total space of possible test networks that we might benchmark is very large — something like X^n, where n is the number of use case attributes that we wish to consider, and X is the average number of discrete values of interest for each attribute.

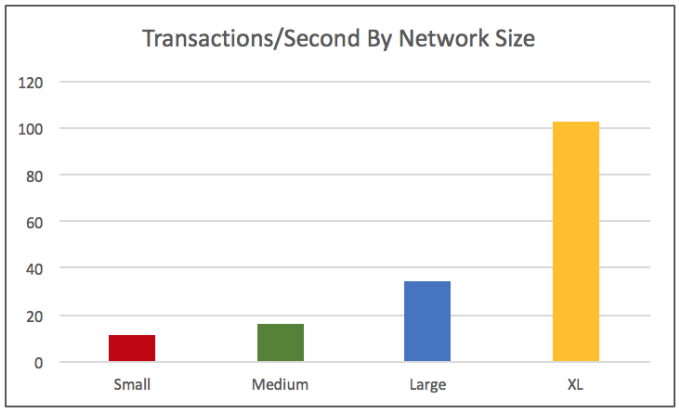
While we often want to hold $n - 1$ use case attributes constant, and vary the values of a single attribute to see how one or more metrics change as that attribute is scaled (this is what we have described as "Single-Use-Case Reports" above), a complementary approach involves measuring one or more metrics for a few discrete "test networks" of different sizes that represent scaled combinations of many of the use case attributes at once.

Often this can be a more realistic style of performance analysis, since, practically speaking, many use case attributes are more likely to scale together rather than in isolation (i.e. if you have more client nodes, you also have more server nodes, more maps/second, more reads/second, etc.), although this is certainly not true of all of the use cases we consider (e.g. block size).

Therefore, we must also consider a style of reporting that defines several test network sizes/configurations and compares metric values for these different-sized networks against one another.



Example graph report for scaled test network style data.



Example bar chart report for scaled test network style data.

6. Requirements for Our Ideal Benchmarking Tool

Here we present the software requirements for our proposed blockchain-based benchmarking tool.

1. **Automation**: The tool shall support running a full benchmark test with the execution of a single command, the results of which will include complete reports.

2. **Configuration**: The tool shall support configuration in terms of which benchmarks are to be run against which types of test networks, and the types of reports that shall be produced.

3. **Deployment**: The tool shall support automated setup, deployment and tear-down of test networks into virtualized environments such as Amazon AWS.

4. **Measurement**: The network performance tool must be able to measure results for metrics concerning traffic, execution and storage.

5. **Reporting**: The network performance tool must be able to produce single-use-case reports and scaled-test-network reports.

6. **Comparisons**: The network performance tool must be able to compare multiple configurations of a single technology stack (Intra-Stack) as well as multiple technology stacks (Inter-Stack).

7. Conclusion and Next Steps

We are passionate about blockchain software and its potential to contribute to the greater social good. However, in order for blockchain-based systems to gain wider adoption, we in the community must develop better methodologies and tools for conducting performance analysis of these software systems.

Specifically, let us think about prospective users who currently rely on traditional, centralized software systems, but are considering migrating to blockchain-based systems. For many of these users, particularly in enterprise, an important part of their evaluation process will include understanding the performance

characteristics of the various systems in question. If useful comparative benchmarks are not readily available, it will be much less realistic for them to make the switch, or even consider developing new software based on the blockchain.

With this in mind, Stratumn is undertaking a long-term research effort to analyze the performance qualities of blockchain-based software systems. This paper, which is the first major result of our work, lays out a detailed conceptual framework for analyzing performance, based on our definitions of benchmark, layer, metric, and use-case. We also discuss the reporting, comparison and interpretation of benchmark results, and include a survey of existing work in the space.

Our next step will be an open-source engineering effort to develop a new benchmarking tool based on the requirements outlined in Section 6. We believe that this tool will offer current and prospective users actionable insights into the performance qualities that they can expect.

At part of this effort, we are seeking community support and participation around both the development of this new tool, as well as the creation of standards for interpreting benchmark results. We hope that this standardization will bring a shared language to the technological choices we make.

APPENDICES

APPENDIX A: The Context for Our Performance Analysis

We are aware that historically speaking, heavily centralized systems with embedded applications enjoy far greater performance than distributed ones. But they do so by sacrificing security, interoperability and inclusiveness that are characteristics of well-designed distributed systems like Ethereum, Hyperledger and Indigo. So while we *aim* toward achieving the performance of centralized systems, our actual *engineering goal* will be to build “comparable” solutions, all things considered.

In order to understand what we mean by “comparable” solutions, let us take the example of three different solutions to the same problem-statement, “three parties want to exchange data without sacrificing privacy”:

- Solution 1. A typical centralized system would be a website that encrypts the data for all of the three parties, provided the website is operated by neutral party external to those three.
- Solution 2. A blockchain such as Hyperledger using “channels” between the parties, thus we would need three channels in total.
- Solution 3. An Indigo application using cryptographic protocols such as Zero-Knowledge Proof Systems.

Now we can compare the performance of these solutions from either of two perspectives:

Business Results: From the perspective of the application owner, or the user, without looking at the internal workings of the underlying technical stack. That is, looking at how well each solution solves the problem by

focusing on measurements such as the success rate of the solution being built, the costs of the solution (financial, operational, transactional), the conversion rate of new users, or other user experience metrics.

Technical Results: By looking into the internal workings of the technology stack for each solution. This means comparing metrics such as transactions per second or the efficiency of the mempool.

Our research and engineering efforts will focus on measuring performance from this latter perspective ("Technical Results").

APPENDIX B: Measurement Techniques

We see three broad measurement techniques: automated monitoring, theoretical solutions, and heuristic techniques. Some metrics end up using a combination of these techniques, such as measuring network latency.

Automated Measurement
Some metrics, such as the speed of transactions being committed or the blockchain database size, can be measured via automated tools. These metrics are the most practical to implement in real world benchmarking software, so we will focus our engineering efforts on this type of metric first.

Theoretical
Metrics that involve a theoretical parameter (such as network centrality) must be calculated mathematically (e.g. with a pen and paper or Matlab). We include a broad range of theoretical metrics from the fields of graph theory and network complexity, as these offer us a way to assess the structure and reliability of the networks we are be building.

Heuristic
Finally, there are those metrics which must be approximated through simulated runs, such as measuring the coordination costs needed to change the rules for the network. These heuristic metrics provide the only means to measure the performance in system setup and governance.

While there is not a 1:1 mapping between metric categories and measurement, we can approximately link them as follows:

	Measurement Technique	**Metric Category**
1	Automated Measurement	Traffic
2	Automated Measurement	Storage
3	Automated Measurement	Execution
4	Theoretical	Structure
5	Theoretical	Reliability
6	Heuristic	Setup

7	Heuristic	Governance
8	Heuristic	Economic

Table 6: Measurement techniques and metric categories

APPENDIX C: Test Blockchain Networks

When we refer to "test networks" in this document, we mean something different than shared public blockchain test networks like the Ropsten or Rinkeby Ethereum test networks. For us, "test networks" will be temporary example networks configured to simulate real-world solutions as closely as is practical, and against which our benchmarks will be executed.

The test networks will include both client and server nodes, and will also include software to support the measurement of system variables of interest, such as processor usage and memory consumption on the nodes. For example, we might define a "Small Indigo Test Network" to be a network composed of four M4-2XLarge virtual machine instances running on Amazon AWS, with one Indigo Framework server node running on each VM, and two M4-Large instances running test clients that utilize the Indigo Framework SDK. The Small Indigo Test Network, will be automatically deployable onto Amazon AWS.

Type	Blockchain Nodes	Local Agents
Instance Count	4	2
CPU	8 x 2.4 GHz Intel Xeon® E5-2676	2 x 2.4 GHz Intel Xeon® E5-2676
Memory	32 gb	8 gb
Network Bandwidth	1000 mb/s	450 mb/s
Software	• Indigo Framework w/ Tendermint • System variables measurement software • Benchmark software framework and server node benchmark implementation code	• Indigo Framework SDK • System variables measurement software • Benchmark software framework and client node benchmark implementation code • Indigo agent test code

Table 7: An example test network ("Small Indigo Test Network")

APPENDIX D: Existing Benchmarking Tools — Detailed Analysis

BlockBench
BlockBench is an open source project for benchmarking blockchain network performance for private blockchains.

Github https://github.com/ooibc88/blockbench
Research paper https://arxiv.org/pdf/1703.04057.pdf

License: *Apache License 2.0*
Contributors: 2
Commits: 10
Dependencies: C++, NodeJS

Blockchains Benchmarked: Ethereum, Parity, Hyperledger

Measures Implemented: Throughput, Latency, Code Execution Performance, Scalability, Fault-Tolerance, Security

Discussion:

A significant feature of Blockbench, is that its creators have chosen to isolate different *layers* of the blockchain applications for testing. The layers that they consider are Consensus, Data, Execution and Application.

A potential downside to Blockench is that the metric implementation code is written in C++, which makes cross-platform use somewhat more difficult than it would be with a VM-based language like Javascript, C# or Python due to the need to compile a new version for each new deployment platform. Additionally, we do not know if Blockbench has included platform-specific dependencies that may limit straightforward support for certain platforms.

Caliper
Caliper is another open source effort to measure blockchain performance. This one has a larger community, with 5 contributors and 100 commits. It also, in theory, offers extensibility for supporting testing of any blockchain project.

Github https://github.com/Huawei-OSG/caliper
Docs https://github.com/Huawei-OSG/caliper/blob/master/README.md
https://github.com/Huawei-OSG/caliper/blob/master/docs/Architecture.md
License: *Apache License 2.0*
Contributors: 5
Commits: 100
Dependencies: NodeJS 6.x, Docker, Docker-compose

Blockchains Benchmarked: Hyperledger Sawtooth, Ethereum (planned), Hyperledger Fabric (planned)

Measures Implemented: Success Rate, Throughput (TPS), Transaction Confirmation Latency, CPU, Memory, Network IO

Discussion:
Caliper looks like a nice project being actively developed by a small community. From what we can tell, they have made reasonable choices in terms of design and tools used. The documentation for the project is well-written, but also relatively minimal, so we were not able to evaluate what the product's reporting layer looks like.

One issue with Caliper is that, at this time, it only supports benchmarking for Sawtooth, which limits its usefulness in terms of being able to compare performance among multiple blockchains.

APPENDIX E: Additional Resources

1. Hyperledger has a Performance and Scalability Working Group whose mission is to discuss blockchain performance metrics and benchmarking. The group is open to anyone who is interested, and conducts monthly conference calls. While they are moving somewhat slowly, it will be very interesting to keep an eye on their work, since their goal is to establish broad industry consensus around best practices, use cases, and metrics of interest. They have also stated the eventual goal of developing a spec for an open-source project similar to the project that we outline in section 6.

2. The Blockbench Research Paper is probably the best overall academic resource that we have found on the topic of performance benchmarking for blockchains.

3. The paper On Scaling Decentralized Blockchains is another nice research paper on the topic of blockchain performance, although it is less focused on measurement.