A Simulator for Heavy-duty Smart Contracts in Blockchain

Jingqiao Xu

Follow this and additional works at: https://lib.dr.iastate.edu/creativecomponents

Part of the Computer and Systems Architecture Commons

Recommended Citation
https://lib.dr.iastate.edu/creativecomponents/827
Abstract

A blockchain is a distributed ledger in which participants who do not fully trust each other agree on the ledger’s content by running a consensus algorithm. It has been more popular and mature in recent years. Smart contracts on the blockchain system are usually redundantly executed by a large number of nodes for the purpose of verification, which can result in a large waste of computation. The waste is especially significant when the smart contracts are heavy-duty. The problem of reducing the computation has attracted a lot of interest from both the research community and the industry. In this creative component, we present a game theoretic design for efficient execution of heavy-duty smart contracts, and develop a simulator to implement the design and evaluate its performance in a large-scale blockchain system that simulates Ethereum. The simulator is built based on BlockSim, an emerging general-purpose blockchain simulator, but has enhanced it with detailed simulation of the heavy-duty smart contract protocol. The simulation results demonstrate the efficacy and efficiency of our design.

1 Introduction

A blockchain is a sequence of blocks holding a list of transactions. Figure 1 shows the structure of the blockchain. Each block has its own unique hash value and stores the unique hash value of its parent, used to link to its parent. The first block, called genesis block, has no parent. A block has two parts, block header and block body. A block header has several attributes, explained as follows. Block version defines the validation rule to follow. Merkle tree root hash is the root value of the Merkle hash tree built based on all the transactions in the block. Timestamp records the mining time of this block, with value 0 representing January 1, 1970. Parent block hash is the unique hash value of its parent block. Hash value is the unique hash value of the current block. Nonce indicates the difficulty level for miners to mine this block. Block body is composed of a transaction
counter and a list of detailed transactions. The number of transactions a block can contain depends on the sizes of a block and each transaction [1].

A blockchain system is a distributed network of many nodes, working as miners or non-miners. Both the miners and non-miners are required to collaborate in broadcasting/forwarding transactions and blocks, as well as to validate transactions and blocks that they receive. The miners are further expected to mine new blocks.

A smart contract is a computer program made by a set of rules and run in the blockchain system. It provides a high level of flexibility for developing and implementing some real-life problems without any trusted third party involved. The smart contract is self-verifying and self-executing. It takes transactions as input, executes the code in the smart contract, and outputs a result. The smart contract has many characteristics [2]: it can be machine readable code on blockchain; it can be used as an event driven program; it is autonomous without a monitor; and it can be part of an application program. Ethereum [3] is an open source decentralized software platform working atop a blockchain system. It supports smart Contracts and distributed applications to be built and run without any trusted third parties.

Most of existing blockchain system works based on the proof of work (POW) protocol. With the protocol, the miners are in a race, called mining, to solve a time-consuming cryptographic problem. The miner who first finds the solution of this problem gets block reward and all the transaction fees. When the miner finds the solution, it is called to have mined a new block successfully. The block has previous block hash. Thus, the system knows the parent block of this block. And then, the miner broadcasts the new block to the network. Other miners validate the block. If the block is valid, the block is added to the
chain. Sometimes, additional branches may appear; the phenomenon is called fork. The
miners continue to work on the longest chain because it implies the most work [4].

Heavy-duty smart contracts (HDSC), referring to the smart contracts that require a
large amount of computation to execute, becomes more common as more complex dis-
tributed systems are built atop blockchain systems. With the current mainstream blockchain
systems (such as Ethereum), all the nodes in the systems need to execute smart contracts
and verify them, which may incur high computational cost.

Many efforts have been applied to reduce the computational cost of smart contracts. For
example, the proof of stake (POS) [5] protocol has been proposed, with which the successful
miners are determined randomly based on the competitors’ stakes (e.g., balances or ages of
their accounts). Compared with the POW protocol, which selects block proposers based on
highly energy consuming competitions, POS randomly selects nodes to be block proposers
based on the stakes they own; thus, POS reduces computation significantly. But the disad-
vantage of POS is that it still requires every stakeholder to verify the result of execution.
Hence, each smart contract is still executed repeatedly and still incurs high computational
cost. Other researchers also propose sharding strategy to reduce the computation. The
sharding strategy can split the processing of transactions into smaller tasks and assign
them to smaller groups of nodes. These smaller tasks can work in parallel which improve
the throughput and reduce the computation [6]. However, The biggest problem is that,
most participants involved in the network cannot validate blocks in all shards, which leads
to a security problem: the participants cannot be sure that the state they are interacting
with is the result of some of the correct sequence of blocks.

Researchers recently have also proposed the on-chain/off-chain hybrid approach, which
combines off-chain execution of heavy-duty or private tasks with on-chain execution of
lightweight and privacy-insensitive tasks. However, the state-of-the-art designs have several
limitations, such as difficult-to-satisfy requirements posed to users, no consideration of
collusion, and inefficient use of on-chain operations, which may hinder their application in
practice. For example, the zKSNARKS proposed in [7] has a setup phrase which is more
expensive than original execution. Another design zkSTARKs also proposed in [7]; it does
not need the setup phrase, but the proof size and verification grows quickly with complexity
which requires much computation. Some designs such as TrueBit [8] require at least one
honest executor to make sure the system is safe. The designs in [9] require the executor
to run secure multiple-party computation protocols which makes the phrase really complex
for blockchain users.

To address the above limitations, Liu et al. [10] propose a new solution. In the solution,
a user who has a heavy-duty task simply posts to the blockchain a heavy-duty smart contract
(HDSC) that follows a carefully-designed template, where the desired number of executors
and other optional parameters are specified; then, the user deposits money that should be
paid to the honest executors. Once the HDSC gets ready for execution, the executors who
are either selected from the blockchain nodes following a default rule or picked by the user,
make deposits to be responsible for their executions, execute individually or collectively the
computation task embedded in the HDSC, commit the execution results, reveal the results, and finally get rewards (if honestly executing the computation) or lose deposits (otherwise). With a low and adaptable frequency, a trusted mechanism (e.g., the underlying blockchain, available TEE, or other mechanism optionally picked by the user) also executes the HDSC, which serves as a deterrence to executors’ misbehavior. A game theoretic scheme is designed to regulate the above operations such that, economically-rational executors have to act honestly or collude only in a benign way where computation cost can be saved but the computation result is guaranteed correct.

To comprehensively evaluate this design, a question is raised: which factors affect the execution time of heavy-duty smart contracts. To explore this problem, we need a tool to evaluate computation cost in HDSC. One evaluation method is based on emulation, which imitates the behaviour of a system in a set of machines. The paper [10] evaluates the factors such as the number of nodes, the number of executors that influence the computation in real life, and so on. However, the approach lacks scalability because only a small number of nodes can be emulated.

Another method is to conduct simulation based on discrete event simulators. Existing simulators are for mainstream blockchains. For example, the simulator [11] proposed by Carlos Faria and the simulators proposed in Maher Alharby [12]. Both of the simulators can simulate the basic operation of blockchain. These models can simulate the process to create transactions, broadcast transactions, and pack transactions. Another simulator is proposed in [13]. The simulator can easily change the parameters of nodes, including the number of the connections, the block size, etc. However, these simulators do not take the computation of HDSC into consideration, and they do not simulate the process to handle transactions in real life such as executing transactions and committing the result. They cannot be used to simulate the operations on HDSC and evaluate the computation. In this creative component, we fill this gap by developing a simulator that can simulate the processing of HDSC in a mainstream blockchain system (such as Ethereum).

Organization: In the following, Section 2 presents the architecture of our simulator. Section 3 provides our proposed HDSC process simulation. Section 4 presents the setup and results of the experiments. Section 5 analysis some supplementary problems. Finally, Section 6 concludes the paper.

2 Architecture of Simulator

In this section, we introduce the architecture of our simulator which follows the architecture proposed in BlockSim [11].

2.1 Simulation framework

Our simulator is a discrete-event simulation model. It is sufficient to specify a blockchain system, whose state changes at a random time. The simulation framework we use is called
BlockSim. It provides the basic simulation blockchain models including network, block, and transaction. The user can extend these models to implement and evaluate their own design. Figure 2 shows the architecture of our BlockSim framework. The core of this framework is the Discrete event simulation engine. The engine can simulate each process as a discrete event. The discrete-event simulation engine we use is called SimPy, which is based on Python. SimPy can be used to model active events, schedule events, queue events, and process events, as well as control the communication among events to simulate asynchronous networking. In the beginning, we give inputs to the simulation world class. These transactions factory creates transactions. The transactions are collected by nodes to create new blocks. The new block will be added to the chain after validated by nodes.

2.2 Simulation world

The Simulation world class is used to handle the input files. These files are necessary input to run the model successfully. There are four kinds of input files: (1) configuration file which contains the name of the blockChain, node location, and the number of nodes; (2) the probability distribution of delay time to validate transactions; (3) the probability distribution of latency between nodes; (4) the probability distributions of reception throughput and sending throughput between nodes at different locations. A user can adjust these parameters, and give these files to simulation world as the input to simulate the model. The users also need to specify the duration time and start time for the model. The class will return an object denoted as world, which is used to store the information of blocks,
transactions, and nodes during the simulation.

2.3 Node Factory class

The Node Factory class is used to create nodes, which are used to simulate the operations of nodes. The node has three main operations: First, these nodes can listen to other nodes they connect with; second, the nodes can apply time delays when receiving and sending messages; third, the nodes can broadcast messages to the network or send messages to some specific nodes they connect with. Each node must be a miner or non-miner. A miner has a transaction queue that packs new transactions. The miner creates a new block by collecting pending transactions in its transaction queue and fitting them into the candidate block. After the block is created, the miner broadcasts this block if it is selected by the network. The process simulates the mining of a new block. A Non-miner only needs to validate new blocks and transactions, as well as broadcast them. Instead of executing cryptographic validations in real world, the simulator adds a time delay to simulate the process.

2.4 Transactions Factory

The Transactions Factory class is used to create transactions that are broadcast by a random selected node. The batched transactions are created in specific time intervals. This factory can control the number of transactions created every minute. Every transaction generated by the factory has a unique hash value calculated based on the transaction’s content. The owner of the transaction is also recorded in this transaction.

2.5 Chain model

The model simulates the behavior of the chain by verifying the hash value of each new block and adding the block to the chain. A new block can be added to the chain in the following three cases:

- In the first case, the previous hash value of the new block is equal to the hash value of current head of the chain (i.e., the block most recently added to the chain), thus the model adds the new block as the head of the chain.

- In the second case, no block’s hash value on the chain is equal to the previous hash value of the new block. This happens when the new block, as a child of its parent block, is received before the parent block due to uncontrollable network delay. In this case, the new block is added to a buffer. Every time the chain adds a new block, it checks the buffer to find if its child block is there. If there is one in the buffer, the child block is also added as the head of the chain;
In the third case, the previous hash value of the new block is not equal to the hash value of the current head of the chain, but an older block’s hash value. Thus, the model will fork the chain to create a secondary chain, and then calculate the difficulty of the secondary chain to check whether it is a new head.

2.6 Consensus class

The consensus class implements rules for validating blocks and transactions. In the simulator, no real validation is conducted, as all the transactions and blocks are assumed valid; instead, the model adds a time delay to simulate the process to validate blocks and transactions. The class also provides rules to calculate the difficulty of new block. Specifically, it uses a simple calculation to simulate the proof of work protocol rule:

$$\text{Difficulty} = P_d + (B_{ts} - P_{ts}),$$

where $P_d$ is the difficulty of the parent block, $B_{ts}$ is the timestamp of the new block, and $P_{ts}$ is the timestamp of the parent block. The difficulty indicates the minimum work required by a node to mine a block successfully.

2.7 Network class

The network class creates a connection channel between nodes, and the nodes connected by a channel can exchange messages through the network channel. Our model applies the TCP protocol to connect each pair of nodes and adds a three-way latency delay to simulate the three-way handshaking. Our model applies a network latency when the nodes exchange the message. During the simulation, the network is responsible for selecting one node to broadcast its candidate block. Each node has a hash rate. The probability of a node being selected depends on the hash rate of the node; the node with a larger hash rate is more likely to be selected. This way, the mining process POW [4] is simulated.

2.8 Block class

The block class is used to record the information of each block on a blockchain, including the index of the block, the coinbase address of the block miner, the time to mine this block, the hash value of this parent block, the hash value of this block, and the difficulty of mining this block.

2.9 Report

The class records the necessary information when simulation is running, including the time to create blocks, the number of blocks, and the information of all transactions in blocks.
2.10 Database

Database, denoted as DB, is a new class designed in the simulator. It is used to remove duplicate transactions and store necessary information. It is a global hashMap structure containing $<Key, Value>$ tuples, where the key is unique in this hashMap. The database has the following functions:

- Record whether the transactions are packed. The key is "packed" + hash of these transactions. The value is a boolean flag to record whether it is packed, and an array to record the index of blocks where the replacement happens.

- Record the first time to execute the HDSC and the last time to reveal the result. To record the first time to execute HDSC, the key in the hashMap is "find" + hash value of this transaction, the value is the first time to execute this HDSC. To record the last time to reveal HDSC, the key in the HashMap is "revealed" + hash value of this HDSC, the value is the last time to execute the HDSC.

- Record the committed times of the execution result of HDSCs. The key is "committed + hash of HDSC"+, and the value is the times to commit the transactions.

**Design goals** We aim to design a simulator based on the above-mentioned component so as to meet the following goals: (i) the simulator can simulate the operations of blockchain; (ii) the simulator can simulate the behaviors of executing HDSC transactions, committing the result and revealing the result; (iii) the simulator can output the executing time of HDSCs.

3 Detailed Design

![flowchart](image)

**Figure 3: HDSC processing Scheme**

Figure 3 shows the flowchart of the process of HDSCs. First, the owner of HDSC posts an HDSC on the chain. The node needs to identify the state of HDSCs to ensure that it is ready for execution. After the excution, the executor creates a cryptographic result of the execution tx called commitment. The executor then commits tx by broadcasting it to the
network. When all the commitments are stabilized on the chain, the executors reveal the result. Finally, the simulator outputs the average delay time to process HDSC. The details are explained as follows.

### 3.1 Post a new smart contract

Like a regular smart contract, a HDSC should be posted to the blockchain before execution. The HDSC owner broadcasts this HDSC and waits for it to be posed and stabilized. The owner makes a deposit to the account of the HDSC to pay the executors. To facilitate the process, each HDSC in our model contains the following:

- An attribute denoted as txType, which is used to identify the type of transactions.

<table>
<thead>
<tr>
<th>Type of transactions</th>
<th>txType value</th>
</tr>
</thead>
<tbody>
<tr>
<td>general transactions</td>
<td>general</td>
</tr>
<tr>
<td>general HDSC transactions before any execution</td>
<td>HDSC</td>
</tr>
<tr>
<td>the commitments posted by the executors</td>
<td>committed</td>
</tr>
<tr>
<td>the result revealed by the executors</td>
<td>revealed</td>
</tr>
</tbody>
</table>

- The state of the HDSC: unready, the transaction is in the unstable block and can not be executed; ready, when the HDSC’s block is stable; committed, the commitments posted by the executors; revealed, the result revealed by the executors.

- An attribute of the HDSC denoted as ExeNumber for the "committed" transaction which is the number of times the same node executes this HDSC. It is used to calculate the number of times of the commitments of the HDSC.

- System parameters of this HDSC, including: N, the number of desired executors; M, the number of blocks that is required to make sure one block is stable.

- One extra hash value for HDSC transactions denoted as Hd to identify the origin HDSC of the commitment. For example, when the node executes a HDSC transaction $tx_0$, it creates a commitment $tx_1$. Hd of $tx_1$ is equal to the Hash of $tx_0$. We can check the Hd of $tx_1$ to know whether $tx_1$ is a commitment of $tx_0$. $tx_0$ and $tx_1$ are linked by Hd. The Hds of the general HDSCs and general transactions are empty.

- Functions called to process the HDSC: execute, called by the node to execute the HDSC; commit, called by executors to broadcast the result of HDSC to the network; reveal, called by the executors to reveal their execution result.

### 3.2 Identifying ready HDSCs

The state of a HDSC is required to transit to be ready for execution. When the block is stable, the node identifies the state of HDSC in this block as ready. The block added
to the block chain starts to become stable when another $M$ new blocks are added to the blockchain. Suppose we are at block $X$, when the new block $X + 1, X + 2, \cdots, X + M$ are added to the chain, block $X$ becomes stable.

### 3.3 Executing a ready HDSC and Committing the result

When a HDSC transits to a ready state at block $Y$, the node starts to execute the computation code in this HDSC and commits the result. We define the node that executes the HDSC as the executor. Unlike the mechanism in the traditional blockchain that all nodes have to execute and verify the transactions, the simulator selects the nodes which are the miners of the next new $N$ blocks to execute the HDSC. By default, the nodes which mine $Y + 1$th, $Y + 2$th, $\cdots$, $Y + N$th blocks are the executors of HDSCs in block $Y$. To prevent other executors from coping the result, the executors do not reveal the result immediately. Instead, the executors commit a cryptographic result.

**Algorithm 1**

```
execute(HDSC, selfExecutor)
1: Yield timeout
2: Create the new result transaction $tx$ whose $ExeNumber = selfExecutor & Hd = Hash of HDSC
3: Start a new process $P$ to broadcast $tx$
```

Algorithm 1 line 1 shows we add a time delay to simulate the time we spend on executing the transaction. Then, we create a new transaction which is the commitment of the HDSC. The $Hd$ of the new transaction should equal the hash value of the HDSC to link the HDSC with its commitment. Considering the same executors can execute the same transactions multiple times but only commit the result one time, we denoted a variable $selfExecutor$ which is the number of times of this HDSC executed by the current executor. If $selfExecutor$ is larger than 0, it indicates the current node is the executor of the HDSC because it executes this HDSC at least once. It will be recorded in $ExeNumber$ of $tx$ to calculate the number of times of commitments.

Algorithm 2 shows the process of committing HDSC. Because only the stable executors can execute the ready HDSC, the node can only execute the transaction in the block $x - M - N - 1$ when the $x$th block is added to the chain. If these transactions are HDSCs and the node is the executor, the node executes them. The simulator stores $T_{start}$ (the first time for different executors to execute the HDSC) in DB.

### 3.4 Revealing the result

After all the executors have posted the commitments and these commitments are stabilized, the executors compare the result. If all the result are the same, the executors reveal the result (in our model, we assume all the result are the same).
Algorithm 2 process of executing committing HDSC

Require: $M, N, x$,
1: if $x - M - N - 1 > 0$ then
2: for transaction in block indexing at $x - M - N - 1$ do
3: if transaction is HDSC then
4: for $i$ in range$(1, N + 1)$ do
5: if the current node is the miner of block indexing at $i + x - M - N - 1$ then
6: selfexecutor++
7: end if
8: end for
9: if selfExecutor > 0 then
10: execute(HDSC, selfExecutor)
11: record the $T_{start}$ to execute this HDSC in DB
12: end if
13: end if
14: end for
15: end if

Algorithm 3 shows that whenever we find a commitment that is ready, we store it in DB to record the number of commitments stabilized on the chain. The key of DB is "committed" + Hd, and the value is the sum of ExeNumber of commitments of this HDSC. When the value is equal to N, it indicates all the commitments are posted on the chain. the executor will reveal the result to the network once all the commitments are stabilized. At the same time, Our simulator records the $T_{end}$ (the last time for different executors to reveal the result) in DB.

3.5 Calculating the average time

Our goal is to calculate the average delay time of each HDSC. The definition of delay time is the time difference between $T_{start}$ and $T_{end}$. $C$ is the number of HDSC transactions. The delay time $T$ to execute this transaction is denoted as:

$$ T = \frac{(T_{end} - T_{start})}{C} $$

The final average delay time should be:

$$ T_{average} = \frac{\sum_{i=1}^{S} T_i}{S} $$

$S$ is the number of the tests under the same condition. $T_{average}$ is the average delay time.
Algorithm 3 reveal the result

Require: $M$, $x$, $N$,
1: for transactions in block indexing $x - M$ do
2:     if txTyPe of transaction == committed then
3:         if the committed time of $HDSC == N$ and selfexecutor > 0 then
4:             create a new "revealed" transaction
5:         Start a new process to broadcast this transaction
6:     end if
7: end if
8: end for

3.6 Optimization

In addition, we optimized the traditional blockchain simulator.

- First, we only used miners to build our network in order to increase the efficiency of our model.
- Second, we use DB to avoid the nodes to add duplicate transactions on the chain.

In the origin design, duplicate transactions exist on the chain because the nodes pack all the transactions they receive. For example, when Node A broadcasts a transaction to Node B, node A will record this event. When B receives the transactions, and broadcasts this transaction again back to A, node A will check whether there is a record to broadcast this transaction to Node B. if there is, it will not broadcast this transaction to B again. However, node A packs the transaction again.

The original design leads to huge numbers of duplicate transactions on the chain. The average delay time would be very long (about 10,000 seconds) because there are many duplicate transactions in the delay time interval. Compared with hundreds of seconds in real life, the result is far beyond expectation. Thus, we optimize the model:

The broadcast process is still the same, but we optimized the process of packing transactions. We record whether a transaction is packed in DB. If it is, this transaction will not be packed any more. One special case we need to consider is when a secondary chain appears. Because the replacement happens in all nodes, we need to consider the processes as asynchronous. Only one flag to record whether the transaction is packed or not in our database is not enough. Figure 4 demonstrates this situation. We have node A and node B. Suppose the replacement happens at 3th block, when the system takes out all the transactions in block 3, the state of these transactions would change from "packed" to "unpacked" in our database. When node A adds a new block, it will pack these "unpacked" transactions back to the chain. The same process happens in node B. In this case, these transactions are added repeatedly to the chain. Thus, the DB also needs to record the index of the block where the replacement happened before. If the index of this block is in our database, our
system does not need to add the transactions in this block back to the chain. In this way, we make sure the same transactions only appear once on the chain.

4 Experiments

Our experiment is to explore how these six parameters affect the average executing time:

- number of nodes: the number of nodes that build our blockchain network.
- number of stable nodes: M.
- number of executors: N.
- HDSC percentage: our model creates a fixed number of transactions per batch, and HDSC percentage is the percentage of HDSC in each batch of transactions.
- Transaction rate: the number of transactions created per minute.
- Computing time for each HDSC: the time for the nodes to execute HDSC.

We change one variable while keeping the default value of another variables to evaluate how the average executing time changes.

Configurations: Simulation evaluation is based on following parameters:

1. number of nodes: 10*2, 20*2, 40*2, 80*2, and 160*2, where 80*2 is default.
2. number of stable nodes: 10, 12, 14, 16, 18, and 20, where 10 is default.
3. number of executors: 6, 8, 10, 12, 14, 16, 18, and 20, where 10 is default.
4. HDSC percentage: 10%, 20%, 30%, and 40%, where 20% is default.
5. Transaction rate (per 60 seconds): 5, 10, 15, 20, and 25, where 15 is default.
6. Computation time for each HDSC: 10s, 20s, 40s, 80s, 160s, and 320s, where 80s is default.
Figure 5: Six Factors vs Average Time
Figure 5(a) shows the change of average delay time as the number of nodes increases. When the number of nodes increases, the average time also increases. The trend matches our expectations. As the number of nodes increases, the frequency of validation and broadcasting increases which incurs a longer time delay.

Figure 5(b) shows the change of the average delay time as the number of stable blocks increases. When the number of stable blocks increases, the wait time needed to reveal the result will increase. An increase of the delay time is expected. Figure 5(b) shows that we need to choose an appropriate value. If the value is small, our model can not make sure the block is indeed stable. If the value is too large, there would be a computation waste.

Figure 5(c) shows the change of the average delay time as the number of executors increases. As expected, when the execution time of a node increases, the average delay time should increase. The reason is that when more nodes execute HDSC, more computation resource is wasted which leads to a longer delay.

Figure 5(d) shows the change of the average delay time as the percentage of the HDSC changes. The HDSC percentage is the percentage of the number of HDSC over the number of transactions. As the percentages of HDSC increase, there are more HDSC transactions for nodes to execute, which also causes an increase in the average time.

Figure 5(e) shows the change of the average delay time as the number of transactions increases. As expected, when the number of transactions increases, both the delay of broadcast transactions and wait time for HDSC will increase.

Figure 5(f) shows the change of the average time as the computing time increases. As the computation time increases, the systems will need to spend more time on executing HDSCs. Thus, the average delay time will increase. That is why we try to make the heavy-duty contracts lightweight in real life.

5 Discussion

In this experiment, we can easily notice how the delay time changes with the factors (number of nodes, HDSC percentage, transactions rate, number of executors, transaction rate, and computing time). We use average delay time to indicate the time our simulators should spend on the HDSC. The whole process starts with executing the HDSC and ends with all the nodes agreeing with the final result of the execution. Therefore, it can be used to indicate how much computation (includes the time for communication) our simulator spends on it.

To get the delay time of executing a HDSC, we calculate the value 2-5 times. There are two reasons why we calculate it 2-5 times: One is that the result starts to become stable after 2-5 times calculation. The other one is that the average delay time matches our experience in real life.
6 Conclusion

In this work, we develop a simulator to simulate the processing of heavy-duty smart contracts (HDSCs) through their whole lifetime, which includes the phases of HDSC posting, execution, result committing, and result reveal. The simulator is built on top of BlockSim, but has enhanced the simulator. Compared with the methods of evaluating the performance of blockchain with a large set of machines, our study method extends higher scalability when exploring the effects of different factors such as the number of nodes, number of transactions on the computations. The results of our experiments show the impacts of various factors on the performance of processing HDSCs. In the future, we will further study the impacts of these factors and explore more strategies to reduce the computation in the blockchain system.

References


