Vite: A High Performance Asynchronous Decentralized Application Platform

Abstract

Vite is a generalised decentralized application platform that meets the requirements of industrial applications for high throughput, low latency and scalability while taking into account security. Vite uses DAG ledger structure, and transactions in ledgers are grouped by accounts. The Snapshot Chain structure in Vite can make up for the lack of security of the DAG ledger. The HDPoS consensus algorithm, through which writing and confirming of transactions are asynchronous, provides high performance and scalability. The Vite VM is compatible with EVM, and the smart contract language extended from Solidity, providing more powerful ability of description. In addition, an important improvement in Vite design is the adoption of a asynchronous Event Driven Architecture, which transmits information through messages between smart contracts, which greatly improves system throughput and scalability. In addition to built-in native tokens, Vite also supports users to issue their own digital assets, and also provides cross chain value transfer and exchange based on Loopring Protocol [1]. Vite realizes resource allocation by quotas, and light users do not have to pay transaction fees. Vite also supports contract scheduling, name service, contract update, block pruning and other features.

1 Introduction

1.1 Definition

Vite is a universal dApp platform that can support a set of smart contracts, each of which is a state machine with independent state and different operational logic, which can communicate by message delivery.

In general, the system is a transactional state machine. The state of the system $\boldsymbol{s} \in \boldsymbol{S}$, also known as the world state, is composed of the state of each independent account. An event that causes changes in account status is called transactions. The more formalized definition is as follows:

Definition 1.1 (Transactional State Machine) *a transactional state machine is a 4-tuple:* $(\boldsymbol{T}, \boldsymbol{S}, \boldsymbol{g}, \boldsymbol{\delta})$*,where* $\boldsymbol{T}$ *is a set of transactions,*$\boldsymbol{S}$ *is a set of states,*$\boldsymbol{g} \in \boldsymbol{S}$ *is the initial state, also known as* `genesis block`*,* $\boldsymbol{\delta} : \boldsymbol{S} \times \boldsymbol{T} \to \boldsymbol{S}$ *is a state transition function.*

The semantics of this transactional state machine is a discrete transition system, which is defined as follows:

Definition 1.2 (Semantics of Transactional State Machine) *The semantics of a transactional state machine* $(\boldsymbol{T}, \boldsymbol{S}, \boldsymbol{s_0}, \boldsymbol{\delta})$ *is a discrete transition system:* $(\boldsymbol{S}, \boldsymbol{s_0}, \to).\to \in \boldsymbol{S} \times \boldsymbol{S}$ *is a transition relationship.*

At the same time, the decentralized application platform is a distributed system with final consistency. Through some consensus algorithm, the final state can be reached between nodes. In realistic scenarios, what is stored in the state of smart contracts is a set of completed data in a decentralized application, with large volume and can not be transmitted between nodes. Therefore, nodes need to transfer a set of transactions to achieve the consistency of the final state. We organize such a group of transactions into a specific data structure, usually referred to as ledgers.

Definition 1.3 (Ledger) *Ledger is composed of a set of transactions, with an abstract data type recursively constructed. It is defined as follows:*

$$\begin{cases} l = \Gamma(T_t) \\ l = l_1 + l_2 \end{cases}$$

Among them,$T_t \in 2^T$*,representing a set of transactions,* $\Gamma \in 2^T \to L$*,represents a function of constructing a book through a set of transactions,*L *is a set of ledgers,*$+ : L \times L \to L$*,representing the operation of merging two sub ledgers into one.*

It should be noted that in such systems, ledgers are usually used to represent a group of transactions, rather than a state. In Bitcoin [2]and Ethereum [3],the ledger is a block chain structure, where transactions are globally ordered. To modify a transaction in the ledger, we need to reconstruct a sub ledger in the account book, thereby increasing the cost of tampering with the transaction.

According to the same group of transactions, different valid books can be constructed, but they represent a different order of transactions and may cause the system to enter a different state. When this happens, it is usually called "fork".

Definition 1.4 (Fork) *Assume $T_t, T_t' \in 2^T$,$T_t \subseteq T_t'$. if $l = \Gamma_1(T_t)$,$l' = \Gamma_2(T_t')$,and don't meet $l \preceq l'$,we can name l and l' are fork legers. $\preceq$ represents prefix relitionship.*

According to the semantics of the transactional state machine, we can easily prove that from an initial state, if the ledger is not forked, each node will eventually enter the same state. So, if a forked ledger is received, will it certainly enter a different state? It depends on the inherent logic of the transaction in the ledger, and how the ledgers organize partial orders between transactions. In reality, there are often some transactions that satisfy the commutative laws, but because of the problem of account design, they frequently cause forks. When the system starts from an initial state, receives two forked ledgers and ends up in the same state, we call these two ledgers a false forked ledger.

Definition 1.5 (False Fork) *Initial state $s_0 \in S$,ledger $l_1, l_2 \in L$,$s_0 \xrightarrow{l_1} s_1, s_0 \xrightarrow{l_2} s_2$. if $l_1 \neq l_2$,and $s_1 = s_2$,we call these two ledgers l_1, l_2 as false fork ledgers.*

A well designed ledger should minimize the probability of false fork

When the fork occurs, each node needs to choose one from multiple forked ledgers. In order to ensure the consistency of the state, the nodes need to use the same algorithm to complete the selection. This algorithm is called the consensus algorithm.

Definition 1.6 (Consensus Algorithm) *Consensus algorithm is a function that receives a set of ledgers and returns the only ledger:*

$$\Phi : 2^L \to L$$

Consensus algorithm is an important part of system design. A good consensus algorithm should possess high convergence speed to reduce the sway of consensus in different forks, and have a high ability to guard against malicious attacks.

1.2 Current Progress

The Ethereum [4]took the lead in realizing such a system. In the design of the Ethereum, the definition of the world state is $S = \Sigma^A$, a mapping from the account $a \in A$ and the state of this account $\sigma_a \in \Sigma$.Therefore, any state in the state machine of the Ethereum is global, which means that a node can achieve the status of any account at any time.

The state transition function δ of Ethereum is defined by a set of program codes. Each group of code is called a smart contract. The Ethereum defines a Turing complete virtual machine, called EVM, whose instruction set is called EVM code. Users can develop smart contracts through a programming language Solidity similar to JavaScript, and compile them into EVM code, and deploy them on Ethereum [5]. Once the smart contract is successfully deployed, it is equivalent to defining contract account a receives the state transition function δ_a. EVM is widely used in such platforms, but there are also some problems. For example, there is a lack of library function support and security problems.

The ledger structure of the Ethereum is a block chain [2]the block chain is made up of blocks, each block contains a list of transactions, and the latter block refers to the hash of the previous block to form a chain structure.

$$\Gamma(\{t_1, t_2, ...|t_1, t_2, ... \in T\}) = (..., (t_1, t_2, ...)) \tag{1}$$

The greatest advantage of this structure is to effectively prevent transactions from being tampered with, but because it maintains the full order of all transactions, the exchange of two transaction orders will generate a new ledger, which has a higher probability of fork. In fact, under this definition, the state space of a transactional state machine is regarded as a tree: the initial state is the root node, the different transaction order represents different paths, and the leaf node is the final state. In reality, the state of a large number of leaf nodes is the same, which leads to a large number of false forks.

The consensus algorithm Φ is called PoW, which first proposed in Bitcoin protocol [2].The PoW algorithm relies on a mathematical problem that is easily verifiable but difficult to solve. For example, based on a hash function $h : N \to N$,finding the result of x,to meet the requirement $h(T + x) \geq d$,$d$is a given number, called the difficulty, T is a binary representation of the trade list contained in the block. Each block in the block chain contains a solution to such problems. Add up the difficulty of all blocks, which is the total difficulty of a block chain ledger:

$$D(l) = D(\sum_i l_i) = \sum_i D(l_i) \tag{2}$$

Therefore, when choosing the correct account from the fork, choose the fork with the highest difficulty:

$$\Phi(l_1, l_2, ..., l_n) = l_m \text{ where } m = \arg\max_{i \in 1..n}(D(l_i)) \tag{3}$$

The PoW consensus algorithm has better security and has been running well in Bitcoin and Ethereum. However, there are two main problems in this algorithm. The first is to solve a mathematical problem that requires a large amount of computing resources, resulting in a waste of energy. The second is the slow convergence speed of the algorithm, thus affecting the system's overall throughput. At present, the TPS of the Ethereum is only about 15, which is totally unable to meet the needs of decentralized applications.

1.3 Direction of Improvement

After the birth of the Ethereum, the Ethereum community and other similar projects began to improve the system from different directions. From the abstract model of the system, the following directions can be improved:

- Improving the system state $\boldsymbol{S}$
- Improving the state transition function $\boldsymbol{\delta}$
- Improving the structure of the ledger $\boldsymbol{\Gamma}$
- Improving the consensus algorithm $\boldsymbol{\Phi}$

1.3.1 Improve the state of the system

The main idea of improving the state of the system is to localize the global state of the world, each node is no longer concerned with all transactions and state transfers, and only maintains a subset of the whole state machine. In this way, the potentials of the set S and the set T are greatly reduced, thus improving the scalability of the system. Such systems include: Cosmos [6],Aelf[7],PChain and so on.

In essence, this side chain based scheme sacrifices the wholeness of the system state in exchange for the scalability. This makes the decentralization of each dApp running on it is weakened - the transaction history of a smart contract is no longer saved by every node in the whole network, but only by a part of the node. In addition, cross contract interaction will become the bottleneck of such a system. For example, in Cosmos, interactions in different Zone require a common chain Hub to complete [6].

1.3.2 Improve state transition function

Based on improving EVM, some projects provide more abundant smart contract programming languages. For example, a smart contract language Rholang is defined in RChain based on π calculus ; the smart contract in NEO is called NeoContract, which can be developed in the popular programming languages such as Java,C# etc; EOS is programmed with C/C++.

1.3.3 Improve the ledger structure

The improvement direction of the ledger structure is the construction of the equivalent class. The linear ledger with the global order of multiple transactions is improved to a nonlinear ledger that only records partial order relations. This nonlinear ledger structure is a DAG (Directed Acyclic Graph). At present, Byteball [8],IOTA[9],Nano[10]and other projects have realized the function of encrypting money based on DAG's account structure. Some projects are trying to use DAG to implement smart contracts, but so far, improvements in this direction are still being explored.

1.3.4 Improve consensus algorithm

The improvement of consensus algorithm is mostly to improve the throughput of the system, and the main direction is to suppress the generation of false fork. Next we will discuss what factors are involved in false fork.

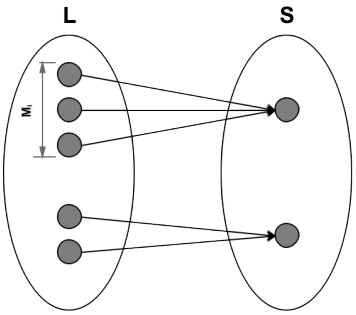

Figure 1: False Fork

As shown,L is a collection of all possible forked accounts for a set of transactions, and S is a collection of states that can be reached in different orders. According to definition ,1.4, mapping $f : L \to S$ is a surjective;And according to definition 1.5, this mapping is not a injective. Here we calculate the probability of the false fork:

Suppose that C users have the right to produce ledgers, $M = |L|$,$N = |S|$,$M_i = |L_i|$, where $L_i = \{l|f(l) = s_i, s_i \in S\}$. The probability of false fork is as follows:

$$P_{ff} = \sum_{i=1}^{N} \left(\frac{M_i}{M}\right)^C - \frac{1}{M^{C-1}} \tag{4}$$

From this formula, we can see that in order to reduce the probability of false fork, there are two ways:

- Establish equivalence relations on the L of the ledger set, divide equivalence classes into them, and construct fewer forked ledgers.
- Restrict users who have the right to produce ledgers, thereby reducing C

The first way is the important direction in Vite design. It will be discussed in detail later. The second ways have been adopted by many algorithms. In the PoW algorithm, any user has the right to produce a block; and the PoS algorithm limits the power of the production block to those with system rights; the DPoS algorithm [11] limits the user with the right to produce the block to be further restricted within a group of agent nodes.

At present, through improved consensus algorithm, some influential projects appeared. For example, Cardano uses a PoS algorithm called Ouroboros, and literature [12] gives a strict proof of the related characters of the algorithm; BFT-DPOS algorithm used by EOS[13], is a variant of the DPoS algorithm and improves system throughput by fast producing blocks; Qtum [14]'s consensus algorithm is also a PoS algorithm; The Casper algorithm adopted by RChain [15] is one of the PoS algorithms as well.

There are also other projects that put forward their own proposals for improving the consensus algorithm. NEO[16] uses a BFT algorithm, called dBFT, and Cosmos[6] uses an algorithm called Tendermint [17].

2 Ledgers

2.1 Overview

The role of ledgers is to determine the order of transactions, and the order of transactions will affect the following two aspects:

- **Consistency of status:** Since the state of the system is not a CRDT (Conflict-free replicated data types) [18], not all transaction is exchangeable, and the sequence of different transaction execution may lead to the system entering a different state.
- **Effectiveness of Hash:** In the ledger, the transaction will be packaged into blocks, which contain hash that is referenced each other. The order of transactions affects the connectivity of hash quoted in the ledgers. The greater the scope of this impact, the greater the cost of tampering with transactions. This is because any change to a transaction must be rebuilt by hash, which directly or indirectly refers to the block of the transaction..

The design of the ledger also has two main objectives:

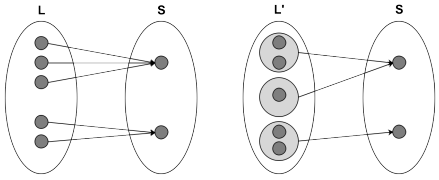

Figure 2: Ledger merge

- **Reducing the false fork rate:** as discussed in the previous section, the reduction of the false fork rate can be achieved by establishing an equivalent class and combining a group of accounts that lead the system into the same state into a single account. As shown above, according to the formula of false fork rate, the false fork rate of the ledger on the left is $P_{ff} = \left(\frac{3}{5}\right)^C + \left(\frac{2}{5}\right)^C - \frac{1}{5^{C-1}}$;after the merge of ledger space, the false fork rate of the right graph is $P_{ff}{}' = \left(\frac{2}{3}\right)^C + \left(\frac{1}{3}\right)^C - \frac{1}{3^{C-1}}$.It is known that when $C > 1$,$P_{ff}{}' < P_{ff}$. That is to say, we should minimize the partial ordering relationship between transactions and allow more transactions to be exchanged sequentially.
- **Tamper proof:** when a transaction t is modified in the ledger l,in the two sub ledgers of the book $l = l_1 + l_2$, the sub ledger l_1 is not affected, and the hash references in the sub ledger l_2 need to be rebuilt to form a new valid ledger $l' = l_1 + l_2{}'$. Affected sub ledger $l_2 = \Gamma(T_2), T_2 = \{x | x \in T, x > t\}$.Thus, to increase the cost of tampering with transactions, it is necessary to maintain the partial order relationship between transactions as much as possible in order to expand the scope of tampering $|T_2|$.

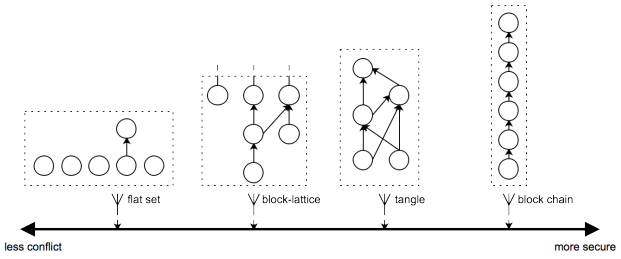

Figure 3: Ledger structure comparison

Obviously, the above two objectives are contradictory, and the necessary trade-offs must be made when designing the account structure. Since the account maintenance is a partial order between transactions, it is essentially a partial ordered set (poset) [19], if represented by Hasse diagram (Hasse diagram)[20], it is a DAG on the topology.

The above picture compares several common ledger structures, and the ledgers near the left are maintained with less partial order. Hasse diagram appears flat and has a lower false fork rate; the ledgers near the right side maintain more partial order relationships, and Hasse diagram is more slender and more tamper resistant.

In the picture, the most-left side is a common set based structure in a centralization system without any tamper proofing features; the most right side is a typical blockchain Ledger with the best tamper proof features; between the two, there are two DAG ledgers, the block-lattice account [10] used by Nano on the left; and the right side, the tangle book [9] is used by IOTA . In terms of characteristics, blocklattice maintains less partial order relations and is more suitable for the accounting structure of high performance decentralized application platforms. Because of its poor tampering characteristics, it can expose security risks, so far, no other projects adopt this ledger structure except Nano.

In order to pursue high performance, Vite adopts the DAG ledger structure. At the same time, by introducing an additional chain structure Snapshot Chain and improving the consensus algorithm, the shortcomings of block-lattice security are successfully made up, and the two improvements will be discussed in detail later.

2.2 Pre Constraint

First, let's take a look at the precondition of using this ledger structure for the state machine model. This structure is essentially a combination of the entire state machine as a set of independent state machines, each account corresponding to an independent state machine, and each transaction only affects the state of an account. In the ledger, all transactions are grouped into accounts and organized into a chain of transactions in the same account. Therefore, we have the

following restrictions on the state S and transaction T in Vite:

Definition 2.1 (Single degree of freedom constraint) *system state $s \in S$,is the vector $s = (s_1, s_2, ..., s_n)$ formed by the state s_i of each account. For $\forall t_i \in T$,after performing the transaction t_i, the system state transfers as follows: $(s_1', ..., s_i', ..., s_n') = \sigma(t_i, (s_1, ..., s_i, ..., s_n))$,need to meet:$s_j' = s_j, j \neq i$.This constraint is called a single degree of freedom constraint for a transaction.*

Intuitively, a single degree of freedom transaction will only change the state of an account without affecting the status of other accounts in the system. In the multidimensional space where the state space vector is located, a transaction is executed, and the state of the system moves only along the direction parallel to a coordinate axis. Please note that this definition is more stringent than the transaction definition in Bitcoin, Ethereum and other models. A transaction in Bitcoin will change the state of the two accounts of the sender and the recipient; the Ethereum may change the state of more than two accounts through a message call.

Under this constraint, the relationship between transactions can be simplified. Any two transaction is either orthogonal or parallel. This provides conditions for grouping transactions according to accounts. Here is an example to illustrate:

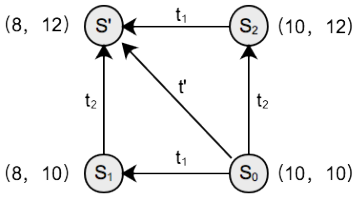

Figure 4: Single degree of freedom trading and intermediate state

As shown in the figure above, suppose Alice and Bob have 10 USD respectively. The initial state of the system is $s_0 = (10, 10)$. When Alice wants to transfer 2 USD to Bob, in the model of Bitcoin and Ethereum, a transactiont',can make the system go directly into the final state:$s_0 \xrightarrow{t'} s'$.

In the definition of Vite, transaction t' changed the status of two accounts of Alice and Bob as well, which did not conform to the principle of single degree of freedom. Therefore, the transaction must be split into two transactions:

1) A transaction t_1 that represents transferring of 2 USD by Alice

2) A transaction t_2 that represents receiving of 2 USD by Bob

In this way, from the initial state to the final state s' there could be two different paths $s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} s'$ and $s_0 \xrightarrow{t_2} s_2 \xrightarrow{t_1} s'$. These two paths are respectively passed through the intermediate state s_1 and s_2, and these two intermediate states are the mapping of the final state s' in the two account dimensions.In other words, if you only care about the state of one of the accounts, you only need to execute all the transactions that correspond to the account, and do not need to carry out the transactions of other accounts.

Next, we will define how to split transactions in Ethereum into the single degree of freedom transactions required by Vite:

Definition 2.2 (Transaction Decomposition) *Dividing a transaction with a degree of freedom greater than 1 into a set of single degree of freedom transactions, named Transaction Decomposition. A transfer transaction can be split into a sending transaction and a receiving transaction; a contract call transaction can be split into a contract request transaction and a contract response transaction; a message call within each contract can be split into a contract request transaction and a contractual response transaction.*

Thus, there would be two different types of transactions in the ledgers. They are called "trading pairs":

Definition 2.3 (Trading Pair) *a sending transaction or contract request transaction, collectively referred to as a "request transaction"; a receiving transaction or a contract response transaction, collectively referred to as "response transaction". A request transaction and a corresponding response transaction are called transaction pairs. The account for initiating the request for transaction t is recorded as $A(t)$; the corresponding response transaction is recorded as: $\widetilde{t}$, the account corresponding to the transaction is recorded as $A(\widetilde{t})$.*

Based on the above definition, we can conclude the possible relationship between any two transactions in Vite:

Definition 2.4 (Transaction Relationship) *There may exist for the following relations for two transactions t_1 and t_2:*

***Orthogonality**:If $A(t_1) \neq A(t_2)$,the two transactions are orthogonal , recorded as$t_1 \perp t_2$;*

***Parallel**:If $A(t_1) = A(t_2)$,the two transactions are parallel, recorded as $t_1 \parallel t_2$;*

***Causality**:If $t_2 = \widetilde{t_1}$,then the two transactions are causal,recorded as $t_1 \rhd t_2$, or $t_2 \lhd t_1$.*

2.3 Definition of Ledger

To define a ledger is to define a poset. First, let's define the partial ordering relationship between transactions in Vite:

Definition 2.5 (Partial order of transactions) *we use dualistic relationship $<$ to represent the partial order relation of two transactions:*

A response transaction must follow a corresponding request transaction :$t_1 < t_2 \Leftrightarrow t_1 \rhd t_2$*;*

All transactions in an account must be strictly and globally ordered:$\forall t_1 \parallel t_2$*, there must be :*$t_1 < t_2$*,or* $t_2 < t_1$*.*

Due to the partial ordering relationship established on the transaction set T meet the characteristics:

- Irreflexive: $\forall t \in T$,there is no $t < t$;
- Transitive:$\forall t_1, t_2, t_3 \in T$, if $t_1 < t_2, t_2 < t_3$, then $t_1 < t_3$;
- Asymmetric:$\forall t_1, t_2 \in T$, if $t_1 < t_2$, then it doesn't exist $t_2 < t_1$

In this way, we can define the Vite account in strict partial order set:

Definition 2.6 (Vite Ledger) *Vite Ledger is thestrict poset composed by set of T of the given transaction, and the partial poset $<$*

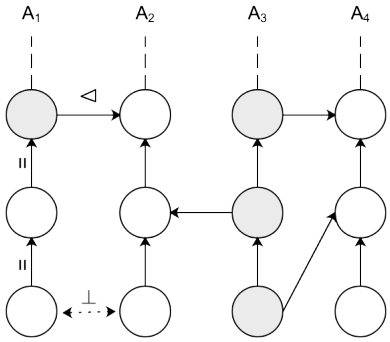

Figure 5: The relationship between the ledger and the transaction in Vite

A strict poset can correspond to a DAG structure. As shown in the figure above, circles represent transactions, and arrows denote dependencies between transactions. $a \rightarrow b$ indicates that a depends on b.

The Vite ledger defined above is structurally similar to block-lattice. Transactions are divided into request and response transactions, each of which corresponds to a separate block, each account Ai corresponds to a chain, a transaction pair, and a response transaction referencing the hash of its corresponding request transaction.

3 Snapshot chain

3.1 Transaction Confirmation

When the account is forked, the result of consensus may swing between two forked ledgers. For example, based on a blockchain structure, if a node receives a longer forked chain, the new fork will be selected as the consensus result, and the original fork will be abandoned and the transaction on the original fork will be rolled back. In such a system, transaction rollback is a very serious event, which will lead to double spend. Just imagine that a business receives a payment, provides goods or services, and after that payment is withdrawn, the merchant may face losses. Therefore, when a user receives a payment transaction, it needs to wait for the system to "confirm" the transaction to ensure that the probability of rolling back is low enough.

Definition 3.1 (Transaction Confirmation) *when the probability of a transaction being rolled back is less than a given threshold ϵ, the transaction is called confirmed.* $P_r(t) < \epsilon \Leftrightarrow t$ *is* $\boldsymbol{confirmed}$*.*

Confirmation of transactions is a very confusing concept, because whether a transaction is recognized depends in fact on the implicit confidence level of $1 - \epsilon$. A merchant selling diamonds and a coffee seller suffered different losses when they were attacked by double spend. As a result, the former needs to set smaller ϵ on the transaction. This is also the essence of the number of confirmations in Bitcoin. In Bitcoin, the confirmation number indicates the depth of a transaction in the block chain. The greater the number of confirmations, the lower the probability of the transaction being rolled back [2]. Therefore, merchants can indirectly set the confidence level of the confirmation by setting the waiting number of confirmation numbers.

The probability of transaction rollback decreases with time due to the hash reference relationship in the account structure. As mentioned above, when the design of the ledger has better tampering characteristics, rolling back a transaction needs to reconstruct all subsequent blocks of the exchange in the block. As new transactions are constantly being added to ledgers, there are more and more successive nodes in a transaction, so the probability of being tampered with will decrease.

In the block-lattice structure, as the transaction is grouped by account, a transaction will only be attached to the end of the account chain of its own account, and the transaction generated by most other accounts will not automatically become a successor node of the transaction. Therefore, it is necessary to design a consensus algorithm reasonably to avoid hidden dangers of double spend.

Nano adopts a voting based consensus algorithm, [10], transaction is signed by a set of representative nodes selected by a group of users. Each representative node has a weight. When the signature of a transaction has enough weight, it is believed that the transaction is confirmed. There are following problems in this algorithm:

First, if a higher confidence degree of confirmation is needed, the threshold of the voting weight needs to be raised. If there are not enough representative nodes online, the intersecting speed can not be guaranteed, and it is possible

that a user will never collect the number of tickets necessary to confirm an exchange;

Second, the probability that transactions are rolled back does not decrease with time. This is because at any time, the cost of overthrowing a historical voting is the same.

Finally, the historical voting results are not persisted into the ledger, and are stored only in the local storage of nodes. When a node gets its account from other nodes, there is no way to reliably quantify the probability of a historical transaction being rolled back.

In essence, the voting mechanism is a partial centralization solution. We can regard the voting results as a snapshot of the status of the ledgers. This snapshot will be distributed in the local storage of each node in the network. In order to have the same tamper proof ability with the block chain, we can also organize these snapshots into chain structures, which is one of the kernel of the Vite design - the snapshot chain [21].

3.2 Definition of snapshot chain

Snapshot chain is the most important storage structure in Vite. Its main function is to maintain the consensus of Vite ledgers. First, we give the definition of the snapshot chain:

Definition 3.2 (Snapshot block and snapshot chain) *a snapshot block that stores a state snapshot of a Vite ledger, including the balance of the account, the Merkle root of the contract state, and the hash of the last block in each account chain. The snapshot chain is a chain structure composed of snapshot blocks, and the next snapshot block refers to the hash of the previous snapshot block.*

The state of a user account contains the balance and the hash of the last block of the account chain; in addition to the above two fields, the state of a contract account contains the Merkle root hash of it, The structure of the state of an account is as follows:

```
struct AccountState {
  // account balance
  map<uint32, uint256> balances;
  // Merkle root of the contract state
  optional uint256 storageRoot;
  // hash of the last transaction
  // of the account chain
  uint256 lastTransaction;
}
```

The structure of the snapshot block is defined as follows:

```
struct SnapshotBlock {
  // hash of the previous block
  uint256 prevHash;
  // snapshot information
  map<address, AccountState> snapshot;
  // signature
  uint256 signature;
}
```

In order to support multiple tokens at the same time, the structure of recording the balance information in Vite's account state is not a `uint256`, but a mapping from the token ID to the balance.

The first snapshot block in the snapshot chain is called the "genesis snapshot", which saves snapshots of the genesis block in the account.

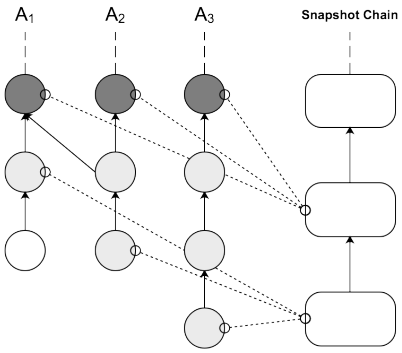

Figure 6: snapshot chain

Since each snapshot block in the snapshot chain corresponds to the only fork of the Vite ledger, it is possible to determine the consensus result of the Vite ledger by the snapshot block when the snapshot block does not fork in the snapshot block.

3.3 Snapshot chain and transaction confirmation

After introducing the snapshot chain, the natural security flaws of block-lattice structure have been remedied. If an attacker wants to generate a double spend transaction, in addition to rebuilding the hash reference in the Vite ledger, it also needs to be rebuilt in the snapshot chain for all the blocks after the first snapshot block of the transaction, and need to produce a longer snapshot chain. In this way, the cost of attack will be greatly increased.

In Vite, the confirmation mechanism of transactions is similar to Bitcoin, which is defined as follows:

Definition 3.3 (Transaction Confirmation in Vite) *in Vite,if a transaction is snapshot by snapshot chain, the transaction is confirmed., the depth of the snapshot block in the first snapshot, is called the confirmation number of the transaction.*

Under this definition, the number of confirmed transactions will increase by 1 when the snapshot chain grows, and the probability of the double spend attack decreases with the increase of the snapshot chain. In this way, users can customize the required confirmation number by waiting for different confirmation numbers according to the specific scenario.

The snapshot chain itself relies on a consensus algorithm. If the snapshot chain is forked, the **longest** fork is chosen as a valid fork. When the snapshot chain is switched to a new fork, the original snapshot information will be rolled back, that means the original consensus on the ledger was overthrown, and replaced by the new consensus. Therefore, snapshot chain is the cornerstone of the whole system security, and needs to be treated seriously.

3.4 Compressed storage

Because all account states need to be saved in every snapshot block in snapshot chain, the storage space is to be very large, the compression to the snapshot chains is necessary.

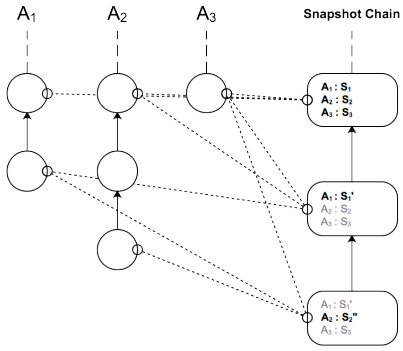

Figure 7: Snapshot before compression

The basic approach of compressing snapshot chain storage space is to use incremental storage: a snapshot block only stores data that is changed compared to the previous snapshot block. If there is no transaction for one account between the two snapshots, the latter snapshot block will not save the data of the account.

To recover snapshot information, you can traverse the snapshot block from the beginning to the end, and cover the data of every snapshot block by the current data.

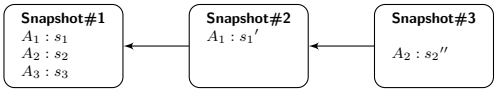

Figure 8: Snapshot after compression

Only the final status of each snapshot of an account is saved when snapshotting, the intermediate state will not be taken into account, so only one copy of the data in the snapshot will be saved, no matter how many transactions generated by an account between the two snapshots. Therefore, a snapshot block takes up to $S * A$ bytes in maximum. Among them, $S = \text{sizeof}(s_i)$, is the number of bytes occupied for each account state, and A is the total number of system accounts. If the average ratio of active accounts to total accounts is a, the compression rate is $1-a$.

4 Consensus

4.1 Goal of Design

When designing a consensus protocol, we need to take full account of the following factors:

- **Performance**.The primary goal of Vite is fast. To ensure high throughput and low delay performance of the system, we need to adopt a consensus algorithm with higher convergence speed.
- **Scalability**.Vite is a public platform that is open to all decentralized applications, so Scalability is also an important consideration.
- **Security**.The design principle of Vite is not pursuing the ultimate safety, however, it still needs to ensure enough safety base line and effectively guard against all kinds of attacks.

Compared with some existing consensus algorithms, the security of PoW is better, and a consensus can be reached if the computing power of malicious nodes are below 50%. However, the intersecting speed of PoW is slow and can not meet the performance requirements; PoS and its variant algorithms remove the steps to solve mathematical problems, improve intersecting speed and single attack cost, and reduce energy consumption. But the Scalability of PoS is still poor, and the "Nothing at Stake" problem [22] is difficult to solve; BFT algorithms has better performance in security and performance, but its Scalability is a problem, usually more suitable for private chain or consortium chain; the DPoS [11] series algorithm effectively reduces the probability of false fork by limiting the permissions of generating blocks. The performance and scalability are good. As a consequence, DPoS has a slight sacrifice in security, and the number of malicious nodes should not be more than 1/3 [23].

Generally, the DPoS algorithm has obvious advantages in performance and scalability. Therefore, we choose DPoS as the basis of the Vite consensus protocol and expand it properly on the basis of it. Through Hierarchical Delegated consensus protocol and asynchronous model, the overall performance of the platform can be further improved.

4.2 Hierarchical Consensus

The consensus protocol of Vite is HDPoS (Hierarchical Delegated Proof of Stake). The basic idea is to decompose the consensus function Φ (functional decomposition):

$$\begin{aligned}\Phi(l_1, l_2, \ldots, l_n) = \Psi(&\Lambda_1(l_1, l_2, \ldots, l_n),\\ &\Lambda_2(l_1, l_2, \ldots, l_n), \ldots\\ &\Lambda_m(l_1, l_2, \ldots, l_n))\end{aligned} \tag{5}$$

$\Lambda_i : 2^L \to L$,is called as local consensus function,the returned result is called the local consensus; $\Psi : 2^L \to L$, known as the global consensus function, it selects a unique

result from a group of candidate in local consensus as the final consensus result.

After this separation, the consensus of the whole system has become two independent processes:

Local consensus generate the blocks corresponding to request transactions and response transaction in the user account or contract account, and writes to the ledgers.

Global consensus snapshots the data in the ledger and generates snapshot blocks. If the ledger is forked, choose one of them.

4.3 Right of Block Generation and Consensus Group

Then, who has the right to generate the transaction block in the ledger and snapshot block in the snapshot chain? What consensus algorithm is adopted to reach a consensus? Since the ledger structure of Vite is organized into multiple account chains according to different accounts, we can conveniently define both the right of production of the blocks in the ledger according to the dimension of the account, and the production right of the snapshot block belong to to a single group of users. In this way, we can put a number of account chains or snapshot chains into a consensus group, and in the consensus group, we can use a unified way to produce the block and reach a consensus.

Definition 4.1 (Consensus Group) *Consensus group is a tuple* (L, U, Φ, P)*, describing the consensus mechanism of a portion of the account or snapshot chain.,*$L \in A|\{A_s\}$*, represents one or a number of account chains, or snapshot chains of the consensus group in the ledger; U represents the user with the block production right on the chain specified by the L;* Φ *specifies the consensus algorithm of the consensus group; and P specifies the parameters of the consensus algorithm.*

Under this definition, users can set up consensus groups flexibly and select different consensus parameters on their needs. Next, we will elaborate on different consensus groups.

4.3.1 Consensus Group of Snapshot

The consensus group of snapshot chains is called snapshot consensus group, which is the most important consensus group in Vite. The consensus algorithm Φ of snapshot consensus group adopts the DPoS algorithm and corresponding to Ψ in the hierarchical model. The number of agents and the interval of the block generation are specified by the parameter P.

For example, we can specify snapshot consensus groups with 25 proxy nodes to produce snapshot blocks at intervals of 1 second. This ensures that the transaction is confirmed to be fast enough. Achieving 10 times transaction confirmation need to wait 10 seconds in maximum.

4.3.2 Private Consensus Group

The private consensus group is only applicable to the production of transaction blocks in ledgers, and belongs to the account chain of private consensus group. The blocks can only be produced by the owner of the private key of the account. By default, all user accounts belong to the private consensus group.

The greatest advantage of the private consensus group is to reduce the probability of fork. Because only one user has the right to produce blocks, the only possibility of fork is that the user initiate a double spend attack personally or a program error.

The disadvantage of the private consensus group is that the user nodes must be online before they can pack the transaction. This is not very suitable for the contract account. Once the owner's node fails, no other node can replace the response transaction that it produces contracts, which is equivalent to reducing the service availability of dApp.

4.3.3 Delegate Consensus Group

In the delegate consensus group, instead of user account ,a set of designated proxy nodes is used to package the transaction through the DPoS algorithm. Both user accounts and contractual accounts can be added to the consensus group. Users can set up a set of separate agent nodes and establish a new consensus group. There is also a default consensus group in Vite to help package transactions for all the other accounts that haven't established their delegate consensus group individually, which is also known as the **public consensus group**.

The delegate consensus group is suitable for most of the contract accounts, because most of the transactions in the contract account are contract response transactions, in which higher availability and lower delays are needed than the receivable transactions in the user account.

4.4 The Priority of the Consensus

In the Vite protocol, the priority of global consensus is higher than that of local consensus. When the local consensus is forked, the result of global consensus selection will prevail. In other words, once the global consensus selected a fork of the local consensus as the final result, even a longer fork of a certain account chain in the future accounts occurs, it will not cause the roll back of the global consensus results.

This problem needs more attention when implementing cross chain protocol. Because a target chain may roll back, the corresponding account chain of the relay contract mapping the chain also needs to roll back accordingly. At this moment, if the local consensus of the relay chain has been adopted by the global consensus, it is impossible to complete the rollback, which may cause the data between the relay contract and the target chain to be inconsistent.

The way to avoid this problem is to set a parameter *delay* in the consensus group parameter P, which specifies the snapshot consensus group to take a snapshot only the local consensus is completed after *delay* blocks. This will greatly reduce the probability of inconsistency of relay contracts, but it can't be avoided completely. In the code logic of relay contracts, it is also necessary to deal with the rollback of the target chain separately.

4.5 Asynchronous Model

In order to improve system throughput further, we need to support a more perfect asynchronous model on the consensus mechanism.

The life cycle of a transaction includes transaction initiation, transaction writing and transaction confirmation. In order to improve the performance of the system, we need to design these three steps into asynchronous mode. This is because at different times, the quantity of transactions initiated by users is different, the speed of transaction writing and transaction confirmation processed by system is fixed relatively. Asynchronous mode helps to flatten the peaks and troughs thus improve the overall throughput of the system.

The asynchronous model of the Bitcoin and the Ethereum is simple: the transaction initiated by all users is placed in an unconfirmed pool. When the miner packages it into a block, the transaction is written and confirmed at the same time. When the block chain continues to grow, the transaction eventually reaches the preset confirmation confidence level.

There are two problems in this asynchronous model:

- Transactions are not persisted to ledgers in an unconfirmed state. Unrecognized transactions are unstable, and there is no consensus involved, it can't prevent sending of transactions repeatedly.

- There is no asynchronous mechanism for writing and confirming of transactions. Transactions are only written when confirmed, and the speed of writing is restricted by the confirmation speed.

The Vite protocol establishes a more improved asynchronous model: first, the transaction is split into a transaction pair based on a "request - response" model, whether it is a transfer or a contract call, and the transaction is successfully launched when a request transaction is written to the ledger. In addition, the written and confirming of a transaction is asynchronous as well. Transactions can be written into the DAG account of Vite firstly and will not be blocked by the confirmation process. Transaction confirmation is done through snapshot chain, and snapshot action is asynchronous too.

This is a typical producer - consumer model. In the life cycle of the transaction, no matter how production rate changes in the upstream, the downstream can deal with the transaction at a constant rate, so as to fully utilized the platform resources and improve the system's throughput.

5 Virtual Machine

5.1 EVM compatibility

At present, there are many developers in the Ethereum field, and many smart contracts are applied based on Solidity and EVM. Therefore, we decided to provide EVM compatibility on the Vite virtual machine, and the original semantics in most of the EVM instruction sets is kept in Vite. Because Vite's account structure and transaction definition is different from Ethereum, the semantics of some EVM instructions need to be redefined, for example, a set of instructions to get block information. The detailed semantic differences can be referred to appendix A..

Among them, the biggest difference is the semantics of message calls. Next we will discuss in detail.

5.2 Event Driven

In the protocol of Ethereum, a transaction or message may affect the status of multiple accounts. For example, a contract invocation transaction may cause the status of multiple contract accounts to change at the same time through message calls. These changes occur either at the same time, or none at all. Therefore, the transaction in the Ethereum is actually a kind of rigid transaction that satisfies the characteristics of ACID (Atomicity, Consistency, Isolation, Durability) [24], which is also an important reason for the lack of expansibility in the Ethereum.

Based on considerations of scalability and performance, Vite adopted a final consistency scheme satisfying BASE (Basically Available, Soft state, Eventual consistency) [25] semantics. Specifically, we design Vite as an Event-Driven Architecture (EDA) [26]. Each smart contract is considered to be an independent service, and messages can be communicated between contracts, but no state is shared.

Therefore, in the EVM of Vite, we need to cancel the semantics of synchronous function calls across contracts, and only allow message communication between contracts. The EVM instructions affected are mainly **CALL** and **STATICCALL**.In Vite EVM, these two instructions can't be executed immediately, nor can they return the result of the call. They only generate a request transaction to write to the ledger. Therefore in Vite, the semantics of function calls will not be included in this instruction, but rather sends messages to an account.

5.3 Smart Contract Language

Ethereum provides a Turing complete programming language Solidity for developing smart contracts. To support asynchronous semantics, we extended Solidity and defined

a set of syntax for message communication. The extended Solidity is called Solidity++.

Most of the syntax of Solidity are supported by Solidity++, but not including the function calls outside the contract. The developer can define messages through the keyword *message* and define the message processor (MessageHandler) through the keyword *on* to implement the cross - contract communication function.

For example, the contract A needs to call the add () method in contract B to update its state based on the return value. In Solidity, it can be implemented by function call. The code is as follows:

```
pragma solidity ^0.4.0;

contract B {
    function add(uint a, uint b) returns
    (uint ret) {
        return a + b;
    }
}

contract A {
    uint total;

    function invoker(address addr, uint a,
    uint b) {
        // message call to A.add()
        uint sum = B(addr).add(a, b);
        // use the return value
        if (sum > 10) {
            total += sum;
        }
    }
}
```

In Solidity++, the function call code`uint sum = B(addr).add(a, b);` is no longer valid; instead of that, contract A and contract B communicate asynchronously by sending messages to each other. The code is as follows:

```
pragma solidity++ ^0.1.0;

contract B {
    message Add(uint a, uint b);
    message Sum(uint sum);

    Add.on {
        // read message
        uint a = msg.data.a;
        uint b = msg.data.b;
        address sender = msg.sender;
        // do things
        uint sum = a + b;
        // send message to return result
        send(sender, Sum(sum));
    }
}

contract A {
    uint total;

    function invoker(address addr, uint a,
    uint b) {
        // message call to B
        send(addr, Add(a, b))
        // you can do anything after sending
        // a message other than using the
        // return value
    }
    Sum.on {
        // get return data from message
        uint sum = msg.data.sum;
        // use the return data
        if (sum > 10) {
            total += sum;
        }
    }
}
```

In the first line ,code `pragma solidity++ 0.1.0;` indicates that the source code is written in Solidity++ but will not be compiled directly with the Solidity compiler to avoid that the compiled EVM code does not conform to the expected semantics. Vite will provide a specialized compiler for compiling Solidity++. This compiler is partially forward compatible: if there is no Solidity code that conflict with the Vite semantics, it can be compiled directly, otherwise the error will be reported. For example, the syntax of local function calls, transfers to other accounts will remain compatible; obtaining the return value of the cross contract function call, as well as the monetary unit `ether`, will not be compiled.

In contract A, when the `invoker` function is called, the `Add` message will be sent to the contract B, which is asynchronous and the result will not be returned immediately.Therefore, it is necessary to define a message processor in A by using the keyword `on` to receive returned result and update the state.

In contract B, the message `Add` is monitored. After processing, a `Sum` message is sent to the sender of the message `Add` to return the result.

Messages in Solidity++ will be compiled into **CALL** instructions and a request transaction will be added to the ledger. In Vite, ledgers serve as message middleware for asynchronous communication between contracts. It ensures reliable storage of messages and prevents duplication. Multiple messages sent to a contract by the same contract can guarantee FIFO (First In First Out); messages sent by different contracts to the same contract do not guarantee FIFO.

It should be noted that the events in Solidity (Event) and the messages in Solidity++ are not the same concept. Events are sent indirectly to front through the EVM log.

5.4 Standard Library

Developers who develop smart contracts on Ethereum are often plagued by the lack of standard libraries in Solidity. For example, loop verification in the Loopring protocol must be performed outside the chain, one of the important reasons is that floating-point computing function is not provided in Solidity, especially the n square root [1][1] for the floating numbers.

In EVM, a pre deployed contract can be called by **DELEGATECALL** command to realize the function of library function. Ethereum also provides several Precompiled Contract, which is mainly a few Hash operations. But these functions are too simple to meet the complex application needs.

Therefore, we will provide a series of standard libraries in Solidity++, such as string processing, floating point operations, basic mathematical operations, containers, sorting, and so on.

Based on performance considerations, these standard libraries will be implemented in a local extension (Native Extension) way, and most of the operations are built into the Vite local code, and the function is called only through the **DELEGATECALL** instruction in the EVM code.

The standard library can be extended as needed, but because the state machine model of the whole system is deterministic, it can not provide functions like random numbers. Similar to Ethereum, we can simulate pseudo random numbers through the hash of snapshot chains.

5.5 Gas

There are two main functions for Gas in the Ethereum , the first one is to quantify the computing resources and storage resources consumed by EVM code execution, and the second is to ensure that the EVM code is halted. According to the computability theory, the Halting Problem on Turing machines is an incomputable problem [27]. That means, it is impossible to determine whether a smart contract can be stopped after limited execution by analyzing the EVM code.

Therefore, the gas calculation in EVM is also retained in Vite. However, there is no Gas Price concept In Vite. Users do not buy the gas for an exchange by paying the fees, but through a quota based model to obtain computing resources. The calculation of quotas will be discussed in detail later in the chapter "economic model".

6 Economic Model

6.1 Native Token

In order to quantify platform computing and storage resources and encourage nodes to run, Vite has built a native token ViteToken. The basic unit of token is *vite*, the smallest unit is *attov*,,1 $vite = 10^{18}$ *attov*.

The snapshot chain is the key to the security and performance of the Vite platform. In order to incite node to participate in the transaction verification, the Vite protocol sets up the forging reward for the production of the snapshot block.

On the contrary, when users issue new tokens, deploy contracts, register VNS domain names [1] and obtain resource quotas, they need to consume or mortgage ViteToken.

Under the combined action of these two factors, it is conducive to optimizing the allocation of system resources.

6.2 Resource Allocation

Since Vite is a common dApp platform, the capabilities of smart contracts deployed on them vary, and each different smart contract has different requirements for throughput and delay. Even for the same smart contract, performance requirements at different stages are different.

In the design of the Ethereum, each transaction needs to be assigned a gas price when launching, so as to compete with other transactions to write accounts. This is a typical bidding model, which can effectively control the balance between supply and demand in principle. However, user is difficult to quantify the current supply and demand situation, and can not predict the price of other competitors, therefore market failure occurs easily. Moreover, the resources competing for each bid are directed against one transaction, and there is no agreement on the rational allocation of resources according to the account dimension.

6.2.1 Quota Calculation

We have adopted a quota based resource allocation protocol in Vite, which allows users to obtain higher resource quotas in three ways:

- A PoW is calculated when the transaction is initiated;
- Mortgage a certain amount of *vite* in the account;
- To destroy a small amount of *vite* in one time.

The specific quotas can be calculated through the following formula:

$$Q = Q_m \cdot \left(\frac{2}{1 + exp\left(-\rho \times \xi^{\top}\right)} - 1 \right) \tag{6}$$

Among them, Q_m is a constant, representing the upper limit of a single account quota, which is related to the total throughput of the system and the total number of accounts.$\xi = (\xi_d, \xi_s, \xi_f)$ is a vector that represents the cost of a user for obtaining a resource: ξ_d is the PoW difficulty that the user calculates when generating a transaction, ξ_s is the *vite* balance of the mortgage in the account, and ξ_f is the one-time cost that the user is willing to pay for the increase of the quota. It should be noted that ,ξ_f is different from the handling fee. These *vite* will be destroyed directly instead of paid to the miners.

[1] refer to *7.2 naming service*

In the formula, the vector $\rho = (\rho_d, \rho_s, \rho_f)$ represents the weight of the three way of obtaining the quota, that is, the quota obtained by the destruction of 1 *vite* is equivalent to the mortgaged ρ_s/ρ_f *vite*.

It can be seen from this formula that if the user neither mortgages *vite* nor pays the one-time cost, it is necessary to calculate a PoW, otherwise there will be no quotas to initiate a transaction, which can effectively prevent dust attacks and protect the system resources from being abused. At the same time, this formula is a Logistic function. It is relatively easy for users to get lower quotas, thereby reducing the threshold of low frequency users; and high frequency users need to invest a lot of resources in order to obtain higher quotas. The extra costs they pay will increase the benefits of all users.

6.2.2 Resource Quantification

Because snapshot chain is equivalent to a global clock, we can use it to quantify the resource usage of an account accurately. In each transaction, the Hash of a snapshot block is quoted, the height of the snapshot block is took as the timestamp of the transaction. Therefore, according to the difference between the two transaction timestamps, we can judge whether the interval between the two transactions is long enough.

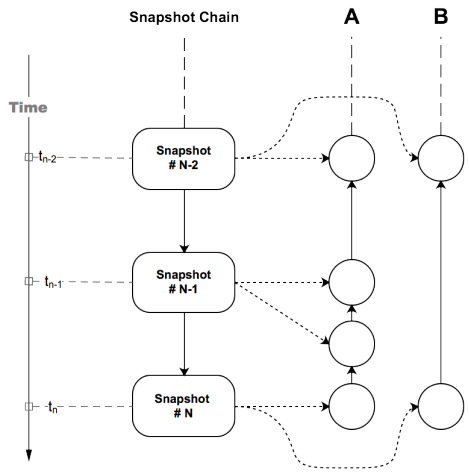

Figure 9: snapshot chain as a global clock

As shown above, account A generated 4 transactions in 2 time intervals, while account B generated only 2 transactions. Therefore, the average TPS of A in this period is 2 times that of B. If it's just a transfer transaction, the average TPS of the quantified account is enough. For smart contracts, each exchange has a different consumption of resources, so it is necessary to accumulate gas for each transaction to calculate the average resource consumption for a period of time. The average resource consumption of the recent k transactions in an account chain with a height of "n" is:

$$Cost_k(T_n) = \frac{k \cdot \sum_{i=n-k+1}^{n} gas_i}{timestamp_n - timestamp_{n-k+1} + 1} \quad (7)$$

Among them, for a transaction T_n, $timestamp_n$ is the timestamp of the transaction, that is, the height of the snapshot block it refers to; gas_n is the fuel consumed for the transaction.

When verifying a transaction, the node will determine whether the quota satisfies the condition: $Cost(T) \leq Q$, and if it is not satisfied, the transaction will be rejected. In this case, users need to repackage a transaction, increase quotas by paying a one-time fee, or wait for a period of time to quote a higher snapshot in the transaction.

6.2.3 Quota Lease

If a user holds abundant *vite* assets, but does not need to utilize so many resource quotas, he can choose to rent his quota to other users.

The Vite system supports a special type of transaction to transfer the right to use an account resource quota. In this transaction, the number of *vite* that can be mortgaged, the address of a transferee, and the duration of a lease can be specified. Once the transaction is confirmed, the resource quota corresponding to the amount of the token will be included in the assignee's account. Once the lease time is exceeded, the quota will be calculated into the transferor account. The unit of leasing time is second. The system will be converted into the height difference of the snapshot block, so there may be some deviation.

The leasing income can be obtained by the user. The Vite system only provides a quota transfer transaction, and the pricing and payment of the leasing can be achieved through a third party smart contract..

6.3 Asset Issurance

In addition to native token ViteToken, Vite also supports users to issue their tokens. The issue of tokens can be done through a special transaction, Mint Transaction. The target address of the mint transaction is 0. In the field *data* of the transaction, the parameters of the token are specified as follows:

```
Mint: {
    name: "MyToken",
    totalSupply: 999999999000000000000000000,
    decimals: 18,
    owner: "0xa3c1f4...fa",
    symbol: "MYT"
}
```

Once the request is accepted by the network, the *vite* included in the mint transaction will be deducted from the

initiator account as the mint transaction fee. The system records the information of the new token and assigns a *token_id* to it. All the balances of the newly generated tokens will be added to the *owner* address, that is to say, the owner account is the genesis account of the token.

6.4 Cross Chain Protocol

In order to support cross chain value transfer of digital assets and eliminate "value island", Vite designed a Vite Cross-chain Transfer Protocol (VCTP).

For every asset that needs cross-chain transmission on the target chain, a token that corresponds to it is needed in the Vite as the voucher of the target Token circulating within the Vite, which is called the ToT (Token of Token). For example, if you want to transfer the *ether* in the Ethereum account to Vite, you can issue a ToT with an identifier of *ETH* in Vite, the initial quantity of TOT should be equal to the total quantity of *ether*.

For each target chain, there is a Gateway Contract on Vite to maintain the mapping relationship between Vite transactions and target chain transactions. In the consensus group where the contract is located, the node responsible for generating blocks is called VCTP Relay. VCTP Relay needs to be the Vite node and the full node of the target chain at the same time, and listen transactions on both sides. On the target chain, we also need to deploy a Vite Gateway Contract.

Before VCTP Relay starts to work, the corresponding ToT in Vite should be transferred to the gateway contract. After that, the supply of ToT can only be controlled by the gateway contract, and no one can be added to ensure the 1: 1 exchange ratio between the ToT and the target asset. At the same time, the assets on the target chain are controlled by the Vite gateway contract, and no user can use it, so as to ensure that ToT has a full acceptance reserve.

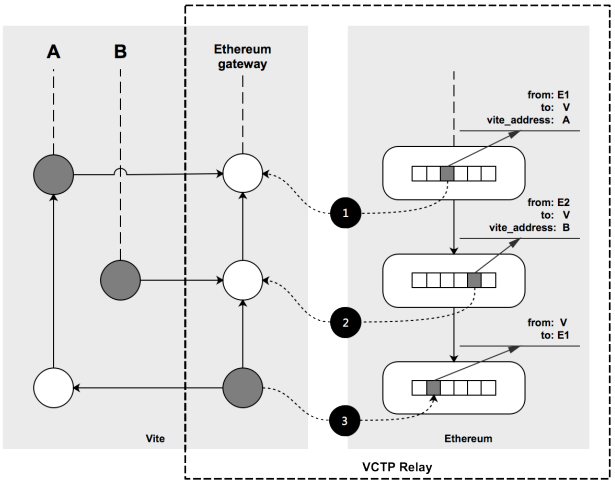

Figure 10: Cross Chain Protocol

The above picture is an example of the cross chain value transmission between the Vite and the Ethereum. When the Ethereum user $E1$ wants to transfer the token from the Ethereum to the Vite, it can send a transaction to the Vite gateway contract address V, while the user's address A on the Vite is placed in the parameter. The balance of the transfer will be locked in the gateway contract account and become part of the ToT reserve. After listening to the transaction, the VCTP relay node generates a corresponding account sending transaction of Vite, sending the same amount of ToT to the user's account A in the Vite. In the picture, ① and ② respectively indicate that $E1$ and $E2$ transfer to Vite account A and B. It should be noted that if the user does not specify the Vite address when transferring, the contract will reject the transaction.

The reverse flow is shown in ③, When the user A launches transfering from the Vite account to the Ethereum account, a transaction will be sent to the Vite gateway contract, transfers to a certain quantity of ToT, and specifies the reception address $E1$ of the Ethereum in the transaction. The VCTP relay node will generate the corresponding response block on the Ethereum Gateway contract, and package a transaction of the Ethereum to the Vite gateway contract on the Ethereum. In the Ethereum , the Vite gateway contract will verify whether this transaction is initiated by a trusted VCTP relay, and then the same amount of *ether* is transferred from the Vite gateway contract to the target account $E1$.

All cross chain relay nodes will monitor the target network, and they can verify whether each cross chain transaction is correct and reach consensus within the consensus group. But snapshot consensus group will not monitor the transaction of the target chain, nor will it verify whether the mapping between the two chains is correct. If the target network is rolled back or hard forked, the mapped transactions in the Vite system cannot be rolled back; similarly, if the cross chain transactions in the Vite are rolled back, the corresponding transaction of the target network can not be rolled back at the same time. Therefore, when doing cross - chain transactions, it is necessary to deal with transaction rollback in contract logic. At the same time, as described in the 4.4 part, we need to set a *delay* parameter for the cross chain Relay consensus group.

6.5 Loopring Protocol

Loopring protocol [1] is an open protocol to build a decentralized asset trading network. Compared to other DEX solutions, the Loopring protocol is based on the multiparty loop matching, which provides a dual authorization technology to prevent preemptive transactions and is fully open.

We build the Loopring protocol into Vite, which is conducive to promoting the circulation of digital assets in Vite, so that the whole value system can be circulated. In this value system, users can issue their own digital assets, transfer assets outside the chain through VCTP, and use

the Loopring protocol to achieve asset exchange. The whole process can be completed within the Vite system and is completely decentralized.

In Vite, Loopring Protocol Smart contract (LPSC) is a part of the Vite system. Asset transfer authorization and multi-party atomic protection are all supported in the Vite. The Loopring relay is still open to fully integrate with its own ecosystem.

Users can use *vite* to pay for asset exchange transactions, so the earned token by miners of Looping who perform loop matching in the Vite platform is still *vite*.

7 Other Designs

7.1 Scheduling

In the Ethereum, smart contracts are driven by transactions, and the execution of contracts can only be triggered by users initiating a transaction. In some applications, a timing scheduling function is needed to trigger the execution of a contract through a clock.

In Ethereum, this function is achieved through third party contracts.[1], performance and security are not guaranteed. In Vite, we add the timing scheduling function to the built in contract. The users can register their scheduling logic into the timed scheduling contract. The public consensus group will use the snapshot chain as a clock, and send the request transaction to the target contract according to the user defined scheduling logic.

There is a specialized `Timer` message in Solidity++. Users can set up their own scheduling logic in the contract code through `Timer.on`.

7.2 Name Service

In Ethereum, the contract will generate an address to identify a contract when it is deployed. There are two problems in identifying contracts with addresses:

- An address is an identifier with 20 bytes without meaning. It is unfriendly to users and inconvenient to use.
- Contracts and addresses are one-to-one. They cannot support contract redirection.

In order to solve these two problems, the developer of Ethereum has provided a third party contract ENS [2]. However, in the actual scenario, the use of naming services will be very frequent, and the use of third party contracts can not guarantee the global uniqueness of naming, so we will build a name service VNS (ViteName Service) in Vite.

Users can register a set of names which is easy to remember and resolve them to the actual address through VNS. Names are organized in the form of domain names, such as *vite.myname.mycontract*. The top-level domain name will be retained by the system for specific purposes. For example, *vite.xx* represents Vite address, and *eth.xx* represents an Ethereum address. The second level domain name is open to all users. Once the user owns the second level domain name, the subdomain can be expanded arbitrarily. The domain name owner can modify the address directed by the domain name at any time, so this function can be used for contract upgrading.

The length of the domain name is not restricted. In VNS, the hash of the domain name is actually stored. The target address can be a non Vite address of less than 256 bit, which can be used for cross chain interaction.

It should be noted that VNS is different from the smart contract Package specification EIP190[3] in Ethereum. VNS is a name resolution service, the name is established at runtime, and the resolution rules can be dynamically modified; and EIP190 is a package management specification, the namespace is static, and it is established at compile time.

7.3 Contract Update

The smart contract of Ethereum is immutable. Once deployed, it can not be modified. Even if there is a bug in the contract, it can not be updated. This is very unfriendly to developers and makes dApp's continuous iteration very difficult. Therefore, Vite needs to provide a scheme to support smart contract update.

In Vite, the process of contract updating includes:

A. Deploys a new version of the contract to inherit the status of the original contract.

B. Points the name of the contract to the new address in VNS.

C. Removes the old contract through the **SELFDESTRUCT** instruction

These three steps need to be completed at the same time, and the Vite protocol ensures the atomicity of the operation. Developers need to ensure that the old contract data are correctly processed in the new version contract.

It should be noted that the new contract will not inherit the address of the old contract. If quoted by the address, the transaction will still be sent to the old contract. This is because different versions of contracts are essentially two completely different contracts, whether they can be modified dynamically or not, depending on the semantics of contracts.

In Vite systems, smart contracts are actually divided into two types, the first one is the background of a dApp, and its business logic is described; and the second is a kind of contract that maps the real world. The previous one is equivalent to an application's background service, which

[1] *Ethereum Alarm Clock* is a third party contract used to schedule the execution of other contracts, refer to `http://www.ethereum-alarm-clock.com/`

[2] *Ethereum Name Service* is a third party contract used for name resoluton, refer to `https://ens.domains/`

[3] *EIP190* Ethereum Smart Contract Packaging Specification,refer to `https://github.com/ethereum/EIPs/issues/190`

needs to be continuously iterated through an upgrade; the latter is equivalent to a contract, and once it comes into effect, no modification can be made, otherwise it is a breach of contract. For such a contract that is not allowed to be modified, it can be decorated with keyword *static* in Solidity++, for example:

```
pragma solidity++ ^0.1.0;

static contract Pledge {
    // the contract that will never change
}
```

7.4 Block Pruning

In a ledger, any transaction is immutable, and users can only add new transactions to the ledger without altering or deleting historical transactions. Therefore, with the operation of the system, the ledgers will become bigger and bigger. If a new node who joining the network wants to restore the latest status, starting from the genesis block and redoing all the historical transactions. After running the system for a period of time, the space occupied by the account book and the time consumed for redoing transactions will become unacceptable. For the high throughput system of Vite, the rate of growth will be much higher than Bitcoin and Ethereum, so it is necessary to provide a technique for clipping the blocks in the ledgers.

Block clipping refers to the deletion of historical transactions that cannot be used in the ledgers, and does not affect the operation of the transactional state machine. So, which transactions can be safely deleted? It depends on which scenario the transaction will be used, including:

- **Recovery**.The primary role of a transaction is to recover status. Because in Vite, snapshot chain stores snapshot information of account status, nodes can recover state from a snapshot block. All transactions before *lastTransaction* in the snapshot block can be tailored to state recovery.
- **Verification of transactions**.To verify a new transaction, it needs to verify the exchange's previous transaction in the account chain, and if it is a response transaction, it also needs to verify the corresponding request transaction. Therefore, in the tailored accounting ledgers, at least one last transaction should be retained in each account chain. In addition, all open request transactions cannot be tailored because their hashes may be referenced by subsequent response transactions.
- **Calculate quotas.**Whether a transaction meets the quota is calculated by judging the sliding average of the last 10 transaction resources, so at least the last 9 transactions need to be saved on each account chain.
- **Inquire about history**. If the node needs to query the transaction history, the transaction involved in the query will not be tailored.

According to different usage scenarios, each node can choose several combinations from the above clipping strategy. It is important to note that clipping involves transactions in ledgers, while snapshot chains need to be kept intact. In addition, what is recorded in the snapshot chain is the hash of the contract state. When the account is clipped, the corresponding state of the snapshot needs to be kept intact.

In order to ensure the integrity of Vite data, we need to retain some "Full nodes" in the network to save all transaction data. Snapshot consensus group nodes are full nodes, and in addition, important users such as exchanges may also become full nodes.

8 Governance

For a decentralized application platform, an efficient governance system is essential for maintaining a healthy ecosystem. Efficiency and fairness should be considered when designing governance systems.

The governance system of Vite is divided into two parts: on-chain and off-chain. On-chain is a voting mechanism based on protocol, and off-chain is the iteration of the protocol itself.

On the voting mechanism, it is divided into two types: Global voting and local voting. The global voting is based on the *vite* held by the user to calculate the rights as the voting weight. The global voting is mainly used for the election of the snapshot consensus group proxy node. The local vote is aimed at a contract. When the contract is deployed, a token is designated as the basis for voting. It can be used to elect the agent nodes of the consensus group in which the contract is located.

Besides the verification of transactions, the agent node of snapshot consensus group has the right to choose whether to upgrade the Vite system Incompatibility. The delegated consensus group proxy node has the right to decide whether to allow the contract to be upgraded so as to avoid potential risks arising from the escalation of contracts. The agent node is used to upgrade decision-making power on behalf of users in order to improve the efficiency of decision-making and avoid the failure of decision-making due to insufficient participation in voting. These proxy nodes themselves are also restricted by consensus protocol. Only if most [1] agent nodes are passed, will the upgrade take effect. If these agents do not fulfill their decision-making power according to the user's expectations, users can also cancel their proxy qualification by voting.

The governance of off-chain is realized by the community. Any Vite community participant can propose an improvement plan for the Vite protocol itself or related systems, which is called VEP (Vite Enhancement Proposal). VEP

[1] according to DPoS protocol, the valid majority is 2/3 of total agent nodes.

can be widely discussed in the community and whether to implement the solution is decided by Vite ecological participants. Whether the protocol will be upgraded for the implementation of a VEP will be ultimately decided by the agent node. Of course, when the differences are large, you can also start a round of voting on the chain to collect a wide range of user opinions, and the proxy node will decide whether to upgrade according to the result of the vote.

Although some Vite participants may not have enough *vite* tokens to vote for their opinions. But they can freely submit VEP and fully express their views. The users who have the right to vote must take full account of the health of the whole ecology for their own Vite rights, and therefore take the views of all the ecological participants seriously.

9 Tasks in future

Transaction verification on snapshot chains is a major performance bottleneck of the system. Because Vite adopts asynchronous design and DAG account structure, transaction validation can be executed in parallel. However, due to the dependence between the transactions of different accounts, the degree of parallelism will be greatly restricted. How to improve the parallelism of transaction verification or adopt a distributed verification strategy will be an important direction for future optimization.

Some shortcomings exist in the current HDPoS consensus algorithm as well. It is also an optimization direction to improve the consensus algorithm, or to be compatible with more consensus algorithms in the delegated consensus group.

In addition, the optimization of virtual machine is also very important for reducing system delay and improving system throughput. Because of the simple design of EVM and the simplification of the instruction set, it may be necessary to design a more powerful virtual machine in the future and define a smart contract programming language with more ability to describe and less security vulnerabilities.

Finally, besides the Vite core agreement, the construction of ancillary facilities supporting ecological development is also an important topic. In addition to SDK support for dApp developers, there is much work to do in dApp foreground ecosystem construction. For example, you can build a dApplet engine based on HTML5 in the mobile wallet application of Vite, allowing developers to develop and publish dApp at low cost.

10 Summary

Compared with other similar projects, the characteristics of Vite include:

- **High throughput**.Vite uses the DAG ledger structure, the orthogonal transaction can be written in parallel to the book; in addition, multiple conconsensus groups do not depend on each other in the HDPoS consensus algorithm, and can work in parallel; the most important thing is that the Vite's inter contract communication is based on the asynchronous model of the message. All these are helpful to improve the throughput of the system.
- **Low delay**.Vite uses the HDPoS consensus algorithm to collaborate to complete the rotation production block through the proxy node, without the need to calculate PoW, the block interval can be reduced to 1 second, which is beneficial to reduce the delay of transaction confirmation.
- **Scalability**.In order to meet the scalability requirements, Vite makes a single degree of freedom limit on the transaction, grouping the transactions in the account according to the account dimension, allowing the block production of different accounts to be completed by different nodes, and to remove the ACID semantics of the cross contract calls to BASE semantics based on the message. In this way, nodes no longer need to save all the state of the world, and the data are saved in the entire distributed network in sharding mode
- **Usability**.The improvements of Vite's usability include providing standard library support in Solidity++, dedicated to processing message syntax, timing scheduling of contract, VNS naming services, support of contract upgrading, and so on.
- **Value circulation**.Vite supports digital asset issuance, cross chain value transfer, token exchange based on Loopring protocol, and so on, forming a complete value system. From the user's point of view, Vite is a fully functional decentralized exchange.
- **Economy**.Because Vite adopts quota based resource allocation model, lightweight users who do not trade frequently do not have to pay high fees or gas charges. Users can choose a variety of ways to change the calculation. Extra quota can also be transferred to other users through quota leasing agreement to improve the efficiency of system resource utilization.

11 Thanks

Sincerely, we would like to thank our consultants for their guidance and assistance to this article. Especially We would like to appreciate the contribution of Loopring team and Loopring community to this project.

References

[1] Daniel Wang, Jay Zhou, Alex Wang, and Matthew Finestone. Loopring: A decentralized token exchange protocol. *URL https://github.com/Loopring/whitepaper/blob/master/en_whitepaper.pdf*.

[2] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[3] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151, 2014.

[4] Vitalik Buterin. Ethereum: a next generation smart contract and decentralized application platform (2013). *URL http://ethereum. org/ethereum.html*, 2017.

[5] Chris Dannen. *Introducing Ethereum and Solidity*. Springer, 2017.

[6] Jae Kwon and Ethan Buchman. Cosmos a network of distributed ledgers. *URL https://cosmos.network/whitepaper*.

[7] Anonymous. aelf - a multi-chain parallel computing blockchain framework. *URL https://grid.hoopox.com/aelf_whitepaper_en.pdf*, 2018.

[8] Anton Churyumov. Byteball: A decentralized system for storage and transfer of value. *URL https://byteball.org/Byteball.pdf*.

[9] Serguei Popov. The tangle. *URL https://iota.org/IOTA_Whitepaper.pdf*.

[10] Colin LeMahieu. Raiblocks: A feeless distributed cryptocurrency network. *URL https://raiblocks.net/media/RaiBlocks_Whitepaper___English.pdf*.

[11] Anonymous. Delegated proof-of-stake consensus, a robust and flexible consensus protocol. *URL https://bitshares.org/technology/delegated-proof-of-stake-consensus/*.

[12] Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. *URL https://eprint.iacr.org/2017/573.pdf*, 2017.

[13] Anonymous. Eos.io technical white paper v2. *URL https://github.com/EOSIO/Documentation/blob/master/TechnicalWhiteP*

[14] Dai Patrick, Neil Mahi, Jordan Earls, and Alex Norta. Smart-contract value-transfer protocols on a distributed mobile application platform. *URL https://qtum.org/uploads/files/cf6d69348ca50dd985b60425ccf282f3.pdf*, 2017.

[15] Ed Eykholt, Lucius Meredith, and Joseph Denman. Rchain platform architecture. *URL http://rchain-architecture.readthedocs.io/en/latest/*.

[16] Anonymous. Neo white paper a distributed network for the smart economy. *URL http://docs.neo.org/en-us/index.html*.

[17] Anonymous. Byzantine consensus algorithm. *URL https://github.com/tendermint/tendermint/wiki/Byzantine-Consensus-Algorithm*.

[18] Shapiro Marc, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. *URL https://hal.inria.fr/inria-00609399v1*, 2011.

[19] Deshpande and Jayant V. On continuity of a partial order. *Proc. Amer. Math. Soc. 19 (1968), 383-386*, 1968.

[20] Weisstein and Eric W. Hasse diagram. *URL http://mathworld.wolfram.com/HasseDiagram.html*.

[21] Chunming Liu. Snapshot chain: An improvement on block-lattice. *URL https://medium.com/@chunming.vite/snapshot-chain-an-improvement-on-block-lattice-561aaabd1a2b*.

[22] Anonymous. Problems. *URL https://github.com/ethereum/wiki/wiki/Problems*.

[23] Dantheman. Dpos consensus algorithm - the missing white paper. *URL https://steemit.com/dpos/@dantheman/dpos-consensus-algorithm-this-missing-white-paper*.

[24] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983.

[25] Dan Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, May 2008.

[26] Jeff Hanson. Event-driven services in soa. *URL https://www.javaworld.com/article/2072262/soa/event-driven-services-in-soa.html*.

[27] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing, second edition, 2006.

Appendices

Appendix A EVM Instruction set

A.0.1 0s: Stop and algebraic operation instruction set

No.	Words	POP	PUSH	Semantics in EVM	Semantics in Vite
0x00	STOP	0	0	Stop to Excute.	Sanme semantics
0x01	ADD	2	1	Add two operands.	Same semantics
0x02	MUL	2	1	Multiplying two operands.	Same semantics
0x03	SUB	2	1	Subtracting two operands.	Same semantics
0x04	DIV	2	1	Divide two operands If the divisor is 0 then returns 0	Same semantics
0x05	SDIV	2	1	Divided with symbol.	Same semantics
0x06	MOD	2	1	Modulus Operation.	Same semantics
0x07	SMOD	2	1	Modulus with symbol.	Same semantics
0x08	ADDMOD	3	1	Add the first two operands and module with 3rd	Same semantics
0x09	MULMOD	3	1	Mmultiply the first two operands and module with 3rd	Same semantics
0x0a	EXP	2	1	The square of two operands.	Same semantics
0x0b	SIGNEXTEND	2	1	Symbol extension.	Same semantics

A.0.2 10s: Comparison and bit operation instruction set

No.	Words	POP	PUSH	Semantics in EVM	Semantics in Vite
0x10	LT	2	1	less than.	Same semantics
0x11	GT	2	1	greater than.	Same semantics
0x12	SLT	2	1	less than with symbol.	Same semantics
0x13	SGT	2	1	greater than with symbol.	Same semantics
0x14	EQ	2	1	equal to.	Same semantics
0x15	ISZERO	1	1	if it is 0.	Same semantics
0x16	AND	2	1	And by bit.	Same semantics
0x17	OR	2	1	Or by bit.	Same semantics
0x18	XOR	2	1	Xor by bit.	Same semantics
0x19	NOT	1	1	Nor by bit.	Same semantics
0x1a	BYTE	2	1	Take one of byte from the second operands.	Same semantics

A.0.3 20s: SHA3 instruction set

No.	Words	PoP	PUSH	Semantics in EVM	Semantics in Vite
0x20	SHA3	2	1	Calculate Keccak-256 hash.	Same semantics

A.0.4 30s: Environmental information instruction set

No.	Words	POP	PUSH	Semantics in EVM	Semantics in Vite
0x30	ADDRESS	0	1	Obtain address . of current account	Same semantics
0x31	BALANCE	1	1	Obtain the balance of an account.	Same semantics. returned is the *vite* balance of account
0x32	ORIGIN	0	1	Obtain the sender addresss of original transaction	Different samantics return 0 forever Vite doesn't maintain the causal relationship between internal transaction and user transaction.
0x33	CALLER	0	1	Obtain the address of direct caller.	Same semantics.
0x34	CALLVALUE	0	1	Obtain the transferred amount in called transaction.	Same semantics
0x35	CALLDATALOAD	1	1	Obtain the parameter in this calling	Same semantics
0x36	CALLDATASIZE	0	1	Obtain size of parameter data in this calling.	Same semantics
0x37	CALLDATACOPY	3	0	Copy called parameter data into memory.	Same semantics
0x38	CODESIZE	0	1	Obtain the size of the running code in current environment.	Same semantics
0x39	CODECOPY	3	0	Copy the running code in current environment into memory.	Same semantics
0x3a	GASPRICE	0	1	Obtain the gas . price in current enviroment	Different samantics ,return 0 forever.
0x3b	EXTCODESIZE	1	1	Obtain the code size of an account.	Same semantics
0x3c	EXTCODECOPY	4	0	Copy the code of. an account into memory	Same semantics
0x3d	RETURNDATASIZE	0	1	Obtain data size of returned from previous calling.	Same semantics
0x3e	RETURNDATACOPY	3	0	Copy the returned data calling previously into memory into memory.	Same semantics

A.0.5 40s: Block info instructions set

No.	Words	POP	PUSH	Semantics in EVM	Semantics in Vite
0x40	BLOCKHASH	1	1	Obtain hash of a block.	Different semantic. return Hash of corresponing snapshot block.
0x41	COINBASE	0	1	Obtain the address. of miner beneficiary in current block	Different semantic. return 0 forever.
0x42	TIMESTAMP	0	1	Return timestamp of current block.	Different semantic. return 0 forever.
0x43	NUMBER	0	1	Return the number or current block.	Different semantic. Return the number of responding transaction block in account chain
0x44	DIFFICULTY	0	1	Return the difficulty of the block.	Different semantic. return 0 forever.
0x45	GASLIMIT	0	1	Return the gas. limitation of the block	Different semantic. return 0 forever.

A.0.6 50s: Stach Memory Storege Control stream operation instruction set

No.	Words	POP	PUSH	Semantics in EVM	Semantics in Vite
0x50	POP	1	0	Pop one data from top of stack.	Same semantics
0x51	MLOAD	1	1	load a word from memory.	Same semantics
0x52	MSTORE	2	0	Save a word to memory	Same semantics
0x53	MSTORE8	2	0	Save a byte to memory.	Same semantics
0x54	SLOAD	1	1	Load a word from storage.	Same semantics
0x55	SSTORE	2	0	Save a word into storage.	Same semantics
0x56	JUMP	1	0	Jump instructions.	Same semantics
0x57	JUMPI	2	0	Jump instructions with condition.	Same semantics
0x58	PC	0	1	Obtain program counter's value.	Same semantics
0x59	MSIZE	0	1	Obtain size of memory.	Same semantics
0x5a	GAS	0	1	Obtain available gas .	Different semantic. return 0 forever.
0x5b	JUMPDEST	0	0	Mark a destination of jumping .	Same semantics

A.0.7 60s and 70s: Stack operation instructions

No.	Words	POP	PUSH	Semantics in EVM	Semantics in Vite
0x60	PUSH1	0	1	Push one byte object. into top of stack	Same semantics
0x61	PUSH2	0	1	Push two bytes object into top of stack.	Same semantics
⋮	⋮	⋮	⋮	⋮	
0x7f	PUSH32	0	1	Push 32 bytes object (whole word) into top of stack	Same semantics

A.0.8 80s: Duplication operation instructions

No.	Words	POP	PUSH	Semantics in EVM	Semantics in Vite
0x80	DUP1	1	2	Duplicate 1st object and push it into top of stack.	Same semantics
0x81	DUP2	2	3	Duplicate 2nd object . and push it into top of stack.	Same semantics
⋮	⋮	⋮	⋮	⋮	
0x8f	DUP16	16	17	Duplicate 16th object and push it into top of stack.	Same semantics

A.0.9 90s: Swap operation instructions

No.	Words	POP	PUSH	Semantics in EVM	Semantics in Vite
0x90	SWAP1	2	2	Swap 1st and 2nd object in stack.	Same semantics
0x91	SWAP2	3	3	Swap 1st and 3rd object in stack.	Same semantics
⋮	⋮	⋮	⋮	⋮	
0x9f	SWAP16	17	17	Swap 1st and 17th object in stack.	Same semantics

A.0.10 a0s: Log operation instructions

No.	Words	POP	PUSH	Semantics in EVM	Semantics in Vite
0xa0	LOG0	2	0	Extend log record, no scheme.	Same semantics
0xa1	LOG1	3	0	Extend log record,. 1 scheme	Same semantics
⋮	⋮	⋮	⋮	⋮	
0xa4	LOG4	6	0	Extend log record,. 4 schemes	Same semantics

A.0.11 f0s: System operation instructions

No.	Words	POP	PUSH	Semantics in EVM	Semantics in Vite
0xf0	CREATE	3	1	Create a new contract.	Same semantics
0xf1	CALL	7	1	Call another contract.	Different semantic. indicate sending a message to an account The returned vale is 0 forever.
0xf2	CALLCODE	7	1	Call the code of another contract Change the status of account.	Same semantics
0xf3	RETURN	2	0	Stop execution and return value.	Same semantics
0xf4	DELEGATECALL	6	1	Call the code of another contract, change contract, change current account status keep original transaction info.	Same semantics
0xfa	STATICCALL	6	1	Call another contract, not allow to change status.	Different semantic. represents to sending message to a contract, don't change status of target contract. return 0 forever.needed result Sending another message through target contract and return.
0xfd	REVERT	2	0	Stop execution and . recover status and return value	Same semantics no semantics of returning left gas.
0xfe	INVALID	∅	∅	invalid instructions.	Same semantics
0xff	SELFDESTRUCT	1	0	Stop execution, set the contract as waiting for deleting return all balance.	Same semantics