StreamNet: Enabling Large Scale DAG based Nakamoto Consensus through Streaming Graph Computing

Abstract—**To achieve high throughput in the POW based blockchain systems, a series of methods has been proposed, and DAG is one of the most active and promising field. We designed and implemented the StreamNet aiming to engineer a scalable and endurable DAG system. When attaching a new block in the DAG, only two tips are selected. One is the 'parent' tip whose definition is the same as in Conflux [21], another is using Markov Chain Monte Carlo (MCMC) technique by which the definition is the same as IOTA [27]. We infer a pivotal chain along the path of each epoch in the graph, and a total order of the graph could be calculated without a centralized authority. To scale up, we leveraged the graph streaming property, high transaction validation speed will be achieved even if the DAG is growing. To scale out, we designed the 'direct signal' gossip protocol to help disseminate block updates in the network, such that message can be passed in the network in a more efficient way. We implemented our system based on IOTA's reference code (IRI), and ran comprehensive experiments over different size of clusters of multiple network topologies.**

Keywords-**Block chain, DAG**

I. INTRODUCTION

Ever since bitcoin [26] has been proposed, blockchain technology has been widely studied for 10 years. Extensive adoptions of blockchain technologies was seen in real world applications such as financial services with potential regulation challenges [24], [32], supply chains [18], [33], [4], health cares [6], [35] and IOT devices [8]. The core of blockchain technology depends on the consensus algorithms applying to the open distributed computing world. Where computers can join and leave the network and these computers can cheat.

As the first protocol that can solve the so called Byzantine general's problem, bitcoin system suffers from the problem of low transaction rate with a transaction per second (TPS) of approximately 7, and long confirmation time (about an hour). As more and more machines joined the network, they are competing for the privileges to attach the block (miners) which result in huge waste of electric power. While sky rocketing fees are payed to make sure the transfers of money will be placed in the chain. On par, there are multiple proposals to solve the low transaction speed issue.

One method intends to solve the speed problem without changing the chain data structure, for instance, the bitcoin cash (BCH) fork of bitcoin (BTC) system tries to improve the throughput of the system by enlarging the data size of each block from 1 Mb to 4 Mb. To minimize the cost of POW, a proof of stake method POS [34] is proposed to make sure that only those who in the network have the privilege to attach the block only if they have a large amount of token shares. Another idea targeting at utilizing the power in POW to do useful and meaningful tasks such as training machine learning models are also proposed [23]. In addition, inspired by the PBFT algorithm [7] and a set of its related variations, so called hybrid (or consortium) chain was proposed. The general idea is to use two step algorithm, the first step is to elect a committee, the second step is collecting committee power to employ PBFT for consensus. Bitcoin-NG [13] is the early adopter of this idea, which splits the blocks of bitcoin into two groups, one is for master election and another for regular transaction blocks. Honey-badger [25] is the system that firstly introduced the consensus committee, it uses a predefined members to perform PBFT algorithm to reach consensus. The Byzcoin system [17] brought forth the idea of POW for the committee election, and uses a variation of PBFT called collective signing for speed purposes. The Algorand [14] utilizes a random function to anonymously elect committee and use this committee to commit blocks, and the member of the committee only have one chance to commit block. All these systems have one common feature, the split of layers of players in the network, which results in the complexity of the implementation of the system.

While aforementioned methods are trying to avoid side chains, another thread of effort is put on using direct acyclic graph DAG to merge side chains. The first ever idea comes with growing the blockchain with trees instead of chains [30], which results in the well known GHOST protocol [31]. If one block links to ≥ 2 previous blocks, then the data structure grows like a DAG instead of tree [28], [29], [20]. There is a improvement of the GHOST based DAG algorithm which can achieve 6000 of TPS in reality [21]. Another set of methods tried to avoid finality of constructing a linear total order by introducing the probability of confirmation in the network [27], [9]. However, suffering from engineering issues, mainly due to the lack of transaction frequency and the growing complexity due to the network expansion. These system in reality are rely on centralized methods to maintain their stability. Some of the side chain methods also borrows the idea of DAG, such as nano [19] and vite [22].

Social network analysis has widely adopted the method of streaming graph computing [12], [15], [11] which deals with how to quickly maintain information on a temporally or spatially changing graph without traversing the whole graph. We view the DAG based method as a streaming graph problem which is about how to compute the total order and achieve consensus without consuming more computing power. In distributed database systems, the problem of replicating data across machines is a well studied topic [10]. Due to the low efficiency of the bitcoin network, there are multiple ways to accelerate the message passing efficiency [16], however, they did not deal with the network complexity. We view the problem of scaling DAG system in the network of growing size and topological complexity as another challenging issue, and proposed our own gossip solution. The main contribution of this paper is how to utilize the streaming graph analysis methods and new gossip protocol to bring the DAG systems into real decentralized, and stabilized growing system.

II. Basic design

A. Data structure

The local state of a node in the StreamNet protocol is a direct acyclic graph (DAG) $G =< B, g, P, E >$. B is the set of blocks in G. $g \in G$ is the genesis block. For instance, vertex g in Figure 1 represents the Genesis block. P is a function that maps a block b to its parent block $P(b)$. Specially, $P(g) = \perp$. In Figure 1, parent relationships are denoted by solid edges. Note that there is always a parent edge from a block to its parent block (i.e., $\forall b \in B$, $b, P(b) >\in E$). E is the set of directly reference edges and parent edges in this graph. $e =< b, b' >\in E$ is an edge from the block b to the block b', which means that b' happens before b. For example in Figure 1, vertex 1 represents the first block, which is referenced by the subsequent block 2, 3 and 4. Vertex 5 has two edges, one is the parent edge pointing to 3, another is reference edge pointing to 4. When a new block is not referenced, it is called a tip. For example, in Figure 1, block 6 is a tip. All blocks in the StreamNet protocol share a predefined deterministic hash function Hash that maps each block in B to a unique integer id . It satisfies that $\forall b \neq b'$, Hash(b) $\neq$ Hash(b').

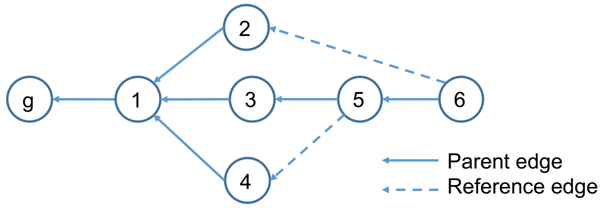


Figure 1. Example of the StreamNet data structure.

B. StreamNet Architecture

Figure 2 presents the architecture of StreamNet, it's consists of multiple StreamNet machines. Each StreamNet machine will grow its DAG locally, and will broadcast the changes using gossip protocol. Eventually, every machine will have a unified view of DAG. By calling total ordering algorithm, every machine can sort the DAG into a total order, and the data in each block can have a relative order regardless of their local upload time. Figure 3 shows the local architecture of StreamNet. In each StreamNet node, there will be a transaction pool accepting the transactions from the HTTP API. And there will be a block generator to pack a certain amount of transactions into a block, it firstly find a parent and reference block to attach the new block to, based on the hash information of these two blocks and the meta data of the block itself, it will then perform the proof of work (POW) to calculate the nonce for the received block. Algorithm 1 summarize the server logic for a StreamNet node. In the algorithm, the way to find parent reference node is by $Pivot(G, g)$. And the way to find reference node is by calling $MCMC(G, g)$ which is the Markov Chain Monte Carlo (MCMC) random walk algorithm [27]. The two algorithms will be described in the later section.

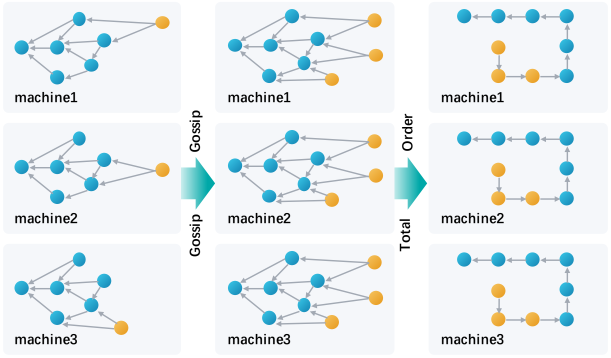


Figure 2. StreamNet architecture.

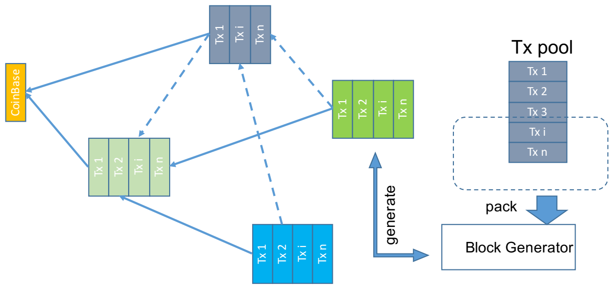


Figure 3. One node in StreamNet protocol.

C. Consensus protocol

Based on predefined data structure, to present the StreamNet consensus algorithm, we firstly define several utility functions and notations, which is a variation from the definition in the Conflux paper [21]. Chain() returns the

Algorithm 1: StreamNet node main loop.

```
Input: Graph $G =< B, g, P, E >$
while Node is running do
    if Received $G' =< B', g, P', E' >$ then
        $G'' \leftarrow< B \cup B', g, P \cup P', E \cup E' >$;
        if $G \neq G''$ then
            $G \leftarrow G''$ ;
            Broadcase updated G to neighbors ;


    if Generate block $b$ then
        $a \leftarrow Pivot(G, g)$ ;
        $r \leftarrow MCMC(G, g)$ ;
        $G \leftarrow< B \cup b, g, P \cup < b, a >, E \cup < b, a > \cup < b, r >>$ ;
        Broadcase updated G to neighbors ;

end
```

Algorithm 2: MCMC(G, b).

```
Input: The local state $G = < B, g, P, E >$ and a starting block $b \in B$
Output: A random tip $t$
$t \leftarrow b$
do
    for $b' \in Child(G, t)$ do
        $P_{bb'} = \frac{e^{\alpha Score(G,b')}}{\Sigma_{z:z \to b} e^{\alpha Score(G,z)}}$
    end
    $t \leftarrow$ choose $b''$ by $P_{bb''}$
while Score(G,t) != 0;
return $t$ ;
```

chain from the genesis block to a given block following only parent edges. Child() returns the set of child blocks of a given block. Sibling() returns the set of siblings of a given block. Subtree() returns the sub-tree of a given block in the parental tree. Before() returns the set of blocks that are immediately generated before a given block. Past() returns the set of blocks that are generated before a given block (but including the block itself). After() returns the set of blocks that are immediately generated after a given block. Later() returns the set of blocks that are generated after a given block (but including the block itself). SubGraph() returns the sub graph by removing blocks and edges except the initial set of blocks. ParentScore() presents the weight of blocks, each block have a score when referenced as parent. Score() presents the weight of blocks, each block achieves a score when attaching to the graph. TotalOrder() returns the 'flatten' order inferred from the consensus algorithm. Figure 4 represents the definition of these utility functions.

1) Parent tip Selection by pivotal chain: The algorithm Algorithm 3 presents our pivot chain selection algorithm(i.e., the definition of $Pivot(G, b)$). Given a StreamNet state G, Pivot(G,g) returns the last block in the pivot chain starting from the genesis block g. The algorithm recursively advances to the child block whose corresponding sub-tree has the largest number of blocks. Which is calculated by $Score(G, b)$ When there are multiple child blocks with the same score, the algorithm selects the child block with the largest block hash. The algorithm terminates until it reaches a tip. Each block in the pivot chain defines a epoch in the DAG, the nodes in DAG that satisfy Past(G,b) - Past(G,p) will belong to the epoch of block b. For example, in Figure 5, the pivot chain is $< g, 1, 3, 5, 6 >$, and the epoch of block 5 contains two blocks 4 and 5.

$$\boxed{\text{G} = < B, g, P, E >}$$

$$Chain(G, b) = \begin{cases} g & \text{b = g} \\ Chain(G, P(b)) & \text{otherwise} \end{cases}$$

$$\overline{Chain(G, b)} = \{b' | b' \in B, b' \notin Chain(G, b)\}$$

$$Child(G, b) = \{b' | P(b') = b\}$$

$$Sibling(G, b) = Child(G, P(b))$$

$$SubTree(G, b) = (U_{i \in Child(G,b)} Substree(G, i)) \cup \{b\}$$

$$Before(G, b) = \{b' | b' \in B, < b, b' > \in E\}$$

$$Past(G, b) = (U_{i \in Before(G,b)} Past(G, i)) \cup \{b\}$$

$$After(G, b) = \{b' | b' \in B, < b', b > \in E\}$$

$$Later(G, b) = (U_{i \in After(G,b)} Later(G, i)) \cup \{b\}$$

$$SubGraph(G, B') =< B', P', E' > |$$

$$\forall < b, b' > \in E', b \subset B' \& b' \subset B'$$

$$ParentScore(G, b) = |SubTree(G, b)|$$

$$Score(G, b) = |Later(G, b)|$$

$$TotalOrder(G) = StreamNetOrder(G, Pivot(G, g))$$

Figure 4. The Definitions of Chain(), Child(), Sibling(), Subtree(), Before(), Past(), After(), Later(), SubGraph(), ParentScore(), Score(), and TotalOrder().

2) Reference tip selection by MCMC: The tip selection method by using Monte Carlo Random Walk (MCMC) is as Algorithm 2 shows. Starting from the genesis, each random walk step will choose a child to jump to, and the probability of jumping from one block to the next block will be calculated using the formula in the algorithm. α in the formula is an constant that is used to scale the randomness of the MCMC function, the smaller it is, the more randomness will be in the MCMC function. The algorithm returns until it finds a tip.

3) Total Order: The algorithm Algorithm 4 defines StreamNetOrder(), which corresponds to our block ordering algorithm. Given the local state G and a block a in the pivot chain, StreamNetOrder(G,a) returns the ordered list of all blocks that appear in or before the epoch of a. Using StreamNetOrder(), the total order of a local state G is defined as TotalOrder(G) in Figure 5. The algorithm in Figure 5

Algorithm 3: PIVOT(G, b).

```
Input: The local state $G = <B, g, P, E>$ and a
       starting block $b \in B$
Output: The last block in the pivot chain for the subtree
        of $b$ in $G$
do
    $b' \leftarrow$ Child$(G, b)$ ;
    $tmpMaxScore \leftarrow$ -1 ;
    $tmpBlock \leftarrow \perp$ ;
    do
        $score \leftarrow$ Score$(G, b')$ ;
        if $score > tmpMaxScore$ || $(score = tmpMaxScore$
        and Hash($b'$ ) < Hash($tmpBlock$)) then
            $tmpMaxScore \leftarrow score$ ;
            $tmpBlock \leftarrow b'$ ;
        end
    while $b' \neq 0$;
    $b \leftarrow tmpBlock$ ;
while Child(G,b) != 0;
return $tmpBlock$ ;
```

first recursively orders all blocks in previous epochs(i.e., the epoch of $P(a)$ and before). It then computes all blocks in the epoch of a as $B_{\triangle}$. It topologically sorts all blocks in $B_{\triangle}$ and appends it into the result list. The algorithm uses the unique hash to break ties.

Algorithm 4: STREAMNETORDER(G, b).

```
Input: The local state $G = <B, g, P, E>$ and a tip
       block $b \in B$
Output: The block list of total top order starting from
        Genesis block to the giving block $b$ in $G$
$L = \perp$
do
    $p \leftarrow$ Parent$(G, b)$ ;
    $B_{\triangle} \leftarrow$ Past($G$,$b$) - Past($G$,$p$) ;
    do
        $G' \leftarrow SubGraph(B_{\triangle})$ ;
        $B'_{\triangle} \leftarrow$ {x || Before($G'$,$x$) = 0} ;
        Sort all blocks in $B'_{\triangle}$ in order as $b'_1, b'_2, ..., b'_k$
        such that $\forall 1 \leq i \leq j \leq k$, Hash($b'_i$) $\leq$ Hash($b'_j$) ;
        $L \leftarrow L + b'_1 + b'_2 + ... + b'_k$ ;
        $B_{\triangle} \leftarrow B_{\triangle}$ - $B'_{\triangle}$ ;
    while $B_{\triangle} \neq 0$;
    $b = p$ ;
while $b$ != $g$;
return $L$ ;
```

D. The UTXO model

In StreamNet, the transactions utilizes the unspent transaction out (UTXO) model, which is exactly the same as in Bitcoin. In the confirmation process, the user will call $TotalOrder$ to get the relative order of different blocks, and the conflict content of the block will be eliminated if the order of the block is later than the one conflicting with it in the total order. Figure 6 shows the example of storage of UTXO in StreamNet and how conflict is resolved. Two blocks both referenced the same block with Alice having 5 tokens, and construct the new transaction out which representing the transfer of token to Bob and Jack respectively. However, after calling $totalOrder()$, the Bob transfer block precedes the Jack transfer block, thus the later block will be discarded.

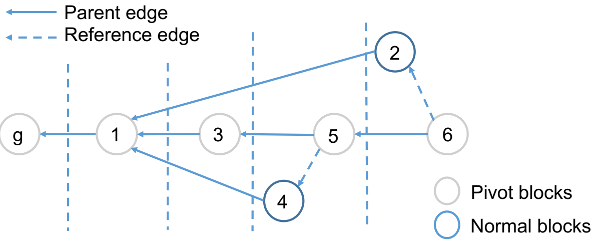


Figure 5. An example of total order calculation.

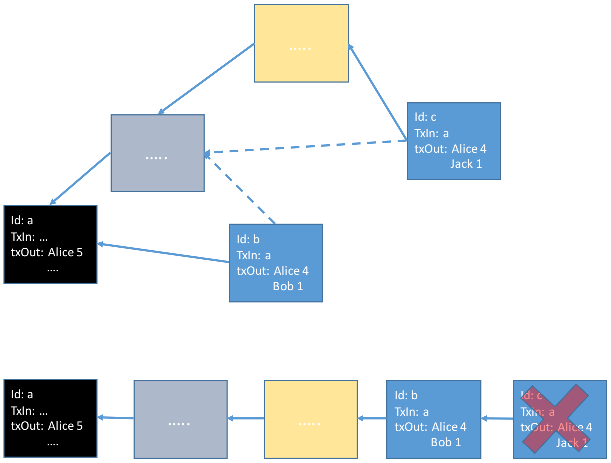


Figure 6. An example of UTXO.

E. Gossip Network

In the bitcoin and IOTA network, the block information is disseminated in a direct mail way [10]. Suppose there are N nodes and L links in the network, for a block of size B, to spread the information of it, the direct mail algorithm will have a total complexity of $O(LB)$. And the average complexity for a node will be $O(\frac{LB}{N})$ In the chain based system, this is fine, because the design of the system already assume that the transaction rate will be low. However, in the DAG based system, this type of gossip manner will result in low scalability due to high throughput of the block generation rate and will result in network flooding. What's worse, consider the heterogeneously and long diameters of network topology, the convergence of DAG will take long time which will cause the delay of confirmation time of blocks.

F. Differences with other DAG protocols

Here, we mainly compare the difference of our protocol with two mainstream DAG based protocols, one is IOTA, another is Conflux.

1) IOTA: The major difference with IOTA are in three points:

- Firstly, the IOTA tip selection algorithm's two tips are all randomly chosen, and ours is one deterministic which is for the total ordering purposes and one by random which is for maintaining the DAG property;
- Secondly, the IOTA consensus algorithm is not purely decentralized, it relies on a central coordinator to issue milestones for multiple purposes, and our algorithm does not dependent on such facility.
- Lastly, in IOTA, there is no concept of total order, and there are 3 ways to judge if a transaction is confirmed:
  - The first way is that the common nodes covered by all the tips are considered to be fully confirmed;
  - All transactions referenced by the milestone tip are confirmed.
  - The third way is to use MCMC. Call N times to select a tip using the tip selection algorithm. If a block is referenced by this tip, its credibility is increased by 1. After N selections have been cited M times, then the credibility is M/N.

2) Conflux: The major difference with Conflux are in two points:

- Firstly, Conflux will approve all tips in the DAG along with parent, which is much more complicated than our MCMC based two tip method. And when the width of DAG is high, there will be much more space needed to maintain such data structure.
- Secondly, Conflux total ordering algorithm advances from genesis block to the end while StreamNet advances in the reverse direction. This method is one of the major contribution to our streaming graph based optimizations, which will be discussed in the next chapter. In Conflux paper, there is no description of how to deal with the complexity paired with the growing graph.

III. Optimization Methods

One of the biggest challenges to maintain the stability of DAG system is that, as the local data structure growing, the graph algorithms ($Pivot()$, $MCMC()$, $StreamNetOrder()$), relies on some of the graph properties that needs to be recalculated every time these functions are called, which are very expensive. Table I list all the expensive graph properties that are called. Suppose the set of blocks in the DAG is represented by B, and the depth of the pivot chain is d. Then we give the analysis of complexity in the following way. $Score()$ relies on the calculation of $SubTree()$ which is dependent on the breath first search (BFS), and the average BFS complexity would be $O(|B|)$, consider the scenario of MCMC() and Pivot() which are advanced through the main chain, the complexity would be in total $O(|B|*d)$ in both of these two cases. The calculation of $Past()$ also relies on the BFS operator, in the StreamNetOrder() algorithm, the complexity would be accrued to $O(|B|*d)$. TopOrder() is used in sub-order ranking the blocks in the same epoch. It's the classical topological sorting problem, and the complexity in the StreamNetOrder() would be $O(|B|)$.

Table I
Analysis of Graph properties calculation

Graph Property	Algorithm used	Complexity	Tot
Score(G, b)	MCMC()	$O(\|B\|)$	$O(\|B\|*d)$
Score(G, b)	Pivot()	$O(\|B\|)$	$O(\|B\|*d)$
Past(G,b) - Past(G,p)	StreamNetOrder()	$O(\|B\|)$	$O(\|B\|*d)$
TopOrder(G, b)	StreamNetOrder()	$O(\|B\|)$	$O(\|B\|)$

Considering new blocks are generated and merged into the local data structure in a streaming way. The expensive graph properties could be maintained dynamically as the DAG grows. Such that the complexity of calculating these properties would be amortized to each time a new block is generated or merged. In the following sections, we will discuss how to design streaming algorithms to achieve this goal.

A. Optimization of Score()

In the optimized version, the DAG will have a map that keeps the score of each block. Once there is a new generated/merged block, it will trigger the BFS based UpdateScore() algorithm to update the scores of the block in the map that are referenced by the new block. The skeleton of the UpdateScore() algorithm is as Algorithm 5 shows.

Algorithm 5: UpdateScore(G, b).

```
Input: Graph $G$, Block $b$, Score map $S$
Output: Updated score map $S$
$Q = [b]$ ;
$visited = \{\}$ ;
while $Q != \emptyset$ do
    $b'$ = $Q.pop()$ ;
    for $b'' \in Before(G, b')$ do
        if $b'' \notin visited \wedge b'' != \perp$ then
            $Q.append(b'')$ ;
            $visited.add(b'')$ ;

    end
    $S[b'] + +$ ;
end
return $S$ ;
```

B. Optimization of Past(G,b) - Past(G,p)

We abbreviate the Past(G,b) - Past(G,p) as GetDiffSet(G,b,C) which is shown in the Algorithm 6. This algorithm is in essence a dual direction BFS algorithm. Starting from the block b, it will traverse its referenced blocks. Every time a new reference block b' is discovered, it will perform a backward BFS to 'look back' to see if itself is already covered by the b's parent block of p. If yes, b' would not be added to the forward BFS queue. To avoid the complexity of the backward BFS, the previous calculated diff set will be added to the covered set C, which will be passed to GetDiffSet() as a parameter. To be more specific, when a backward BFS is performed, the blocks in C will not be added to the search queue. This backward search algorithm is denoted as IsCovered() and described in detail in Algorithm 7.

Figure 7 shows the example of the GetDiffSet() method for block 5. It first perform forward BFS to find block 4 which does not have children, then it will be added to the diff set. 4 then move forward to 1, which have three children, if it detect 3 which is the parent of 5, it will stop searching promptly. If it continue searching on 2 or 4, these two blocks would not be added to the search queue, because they are already in the covered set.

```
Algorithm 6: GetDiffSet($G$, $b$, $C$).
Input: Graph $G$, Block $b$, covered block set $C$
Output: diff set $D \leftarrow Past(G,b) - Past(G,p)$
$D = \emptyset$ ;
$Q \leftarrow [b]$ ;
$visited = \{b\}$ ;
$p$ = $Parent(G,b)$ ;
while $Q! = \emptyset$ do
    $b'$ = $Q.pop()$ ;
    for $b'' \in Before(G,b')$ do
        if $IsCovered(G,p,b'',C) \wedge b''! = \perp$ then
            $Q.append(b'')$ ;
            $visited.add(b'')$ ;

    end
    $D.add(b')$ ;
    $C.add(b')$ ;
end
return $D$ ;
```

C. Optimization of TopOrder()

The topological order is used in sorting the blocks in the same epoch. To get the topological order, every time, there needs a top sort of the whole DAG from the scratch. However we can easily update the topological order when a new block is added or merged. The update rule is, when a new block is added, it's topological position will be $TopScore(G,b) \leftarrow min(TopScore(G,Parent(b)), TopScore(G,Reference(b)))$. This step can be done in $O(1)$

```
Algorithm 7: IsCovered($G$, $p$, $b'$, $C$).
Input: Graph $G$, Block $b'$, parent $p$, covered block set $C$
Output: true if covered by parent, else false
$Q \leftarrow [b']$ ;
$visited = \{b\}$ ;
while $Q! = \emptyset$ do
    $b''$ = $Q.pop()$ ;
    for $t \in Child(G,b'')$ do
        if $t = p$ then
            return true ;
        else if $t \notin visited \wedge t \notin C$ then
            $Q.add(t)$ ;
            $visited.add(t)$ ;

    end
end
return false ;
```

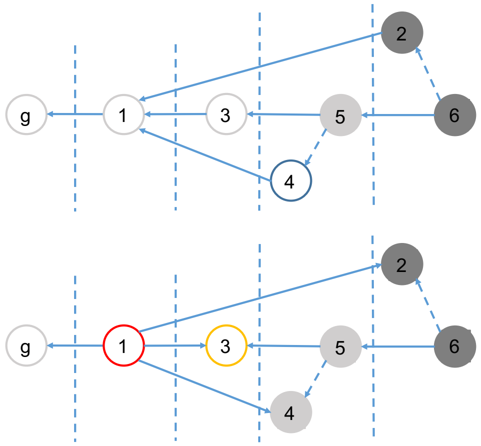


Figure 7. Example of the streaming get diff set method.

To summarize, the optimized streaming operators can achieve the performance improvement as Table II shows.

D. Genesis Forwarding

The above algorithm solved the problem of how to dynamically maintaining the information needed for total order computation. However, it still need to update the information until genesis block. With the size of the graph growing, the updating process still becoming harder to compute. Consider that with the DAG size growing, the old historical confirmed

Table II
ANALYSIS OF GRAPH PROPERTIES CALCULATION

Graph Property	Algorithm used	Complexity	Tot
Score(G, b)	MCMC()	$O(\|B\|)$	$O(d)$
Score(G, b)	Pivot()	$O(\|B\|)$	$O(d)$
Past(G,b) - Past(G,p)	StreamNetOrder()	$O(\|B\|)$	$O(\|B\|)$
TopOrder(G, b)	StreamNetOrder()	$O(\|1\|)$	$O(\|1\|)$

block are confirming by more and more blocks, and it will be very hard to mutate it. Hence, we can design a strategy to forward the genesis periodically and fix the historical blocks into a total ordered chain. The criteria to forward the genesis are based on the threshold of ParentScore. Suppose we define this threshold as t, then we only forward the genesis if:

$$\begin{aligned} &\exists b' | b' \in Chain(G, g), for \\ &\forall b'' | b'' \in \overline{Chain(G, b)}, suchthat \\ &ParentScore(b') > ParentScore(b'') + t \end{aligned} \tag{1}$$

In addition, after the new genesis has been chosen, we will induce a new DAG in memory from this genesis and the vertices in UTXO graph that belongs to the fixed blocks will be eliminated from the memory as well. The algorithm is as Algorithm 8 shows.

Algorithm 8: Genesis Forward Algorithm.

```
Input: Graph $G =< B, g, P, E >$
while Node is running do
    if $\exists b'$ satisties (1) then
        $g' \leftarrow b'$ ;
        $G' \leftarrow induceGraph(G, g')$ ;
        UpdateScore(G', g');
        $G \leftarrow G'$ ;

end
```

E. The Direct Signal Gossip Protocol

To minimize the message passing in the gossip network, there are solutions in [10]. And in Hyperledger [5] they have adopted the PUSH and PULL model for the gossip message propagation. However, their system is aiming at small scale. Suppose the size of the hash of a block is H, we designed the direct signal algorithm. The algorithm is divided into two steps, once a node generate or receive a block, it firstly broadcast the hash of the block, this is the PUSH step. Once a node receive a hash or a set of a hash, it will pick one source of the hash for the block content, this is the PULL step. The direct signal algorithm's complexity will be $O(LH+NB)$ and for a node averaged to $O(\frac{LH}{N}+1)$ The algorithm is as Algorithm 9 shows.

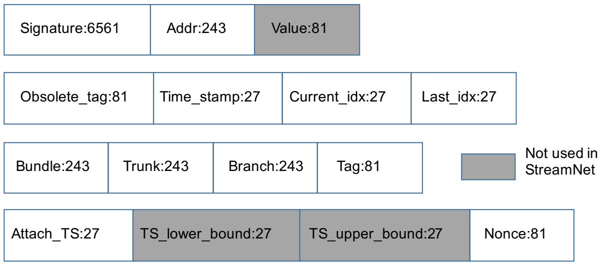
Algorithm 9: Gossip Algorithm.

```
Input: Graph $G =< B, g, P, E >$
while Node is running do
    if Received or Generate block $b$ then
        $h \leftarrow Hash(b)$ ;
        $cache[h] \leftarrow b$ ;
        Broadcast $h$ to neighbors ;

    if Received request of $h$ from neighbor $n$ then
        $b \leftarrow cache[h]$ ;
        Send $b$ to $n$ ;

    if Received hash $h$ from neighbor $n$ then
        $b \leftarrow cache[h]$ ;
        if $b = NULL$ then
            Send request $h$ to $n$ ;


end
```



Field	Field	Field	Field
Signature:6561	Addr:243	Value:81	
Obsolete_tag:81	Time_stamp:27	Current_idx:27	Last_idx:27
Bundle:243	Trunk:243	Branch:243	Tag:81
Attach_TS:27	TS_lower_bound:27	TS_upper_bound:27	Nonce:81

▨ Not used in StreamNet

Figure 8. Block header format, the main transaction information is stored in the signature part. Addr is sender's address, time stamp is the time the block has been created, current/last index and the bundle are used for storing the bundle information, trunk and branch are the hash address to store the parent and reference location, tag is used for store some tagging information, addtach_TS is when the block is attached to the StreamNet, nonce is used in POW calculation.

IV. Experimental Results

A. Implementation

We have implemented the StreamNet based on the IOTA JAVA reference code (IRI) v1.5.5 [1]. We forked the code and made our own implementation, the code is freely available at [3]. In this paper we use version v0.1.4-streamnet in v0.1-streamnet beta branch.

- The features we have adopted from the IRI are:
  - The block header format, as shown in Figure 8. Some of the data segments are not used in StreamNet which are marked grey.
  - Gossip network, the network is a bi-directional network in which every node will send and receive data from its peers;
  - Transaction bundle, because of the existence of the bundle hash feature, StreamNet can support both the single transaction for a block and batched transactions as a bundle.
  - Sponge hash functions which is claimed to be quantum immune, in our experiment, the POW hardness is set to 8 which is the same as the testnet for IOTA.
- The features we have abandoned from the IRI are:
  - The iota's transaction logic including the ledger validation part;
  - The milestone issued by coordinators, which is a centralized set up.

- The features we have modified based on the IRI is:
  - The tip selection method based on MCMC, since the tip selection on IRI has to find a milestone to start searching, we replace this with a block in the pivotal chain instead.
- The features we have added into the StreamNet are:
  - The consensus algorithms, and we have applied the streaming method directly in the algorithms;
  - The UTXO logic which is stored in the signature part of the block header, we used the graph data structure to store UTXO as well.
  - In IOTA's implementation, the blocks are stored in the RocksDB [2] as the persistence layer, which makes it inefficient to infer the relationships between blocks and calculate graph features. In our implementation, we introduced an in-memory layer to store the relationships between blocks, such that the tip selection and total ordering algorithm will be accelerated.

B. Environment Set Up

We have used the Tencent cloud services with 7 virtual machines, and we have splitted the virtual machines into two different set up with 3 strong nodes and 4 weak nodes.

- For a strong node, it includes a four core Intel(R) Xeon(R) CPU E5-26xx v4, with 15 Gb of memory size and 197Gb of disk size.
- For a weak node, it includes a two core Intel(R) Xeon(R) CPU E5-26xx v4, with 3 Gb of memory size and 118Gb of disk size.

The JAVA version is 1.8, we have deployed our service using docker and the docker version is 18.02.0-ce.

We have 7 topologies set up of nodes, which are shown in Figure 9, these configurations are aiming to test:

- The performance when the cluster connectivity is high (congestion of communications, like 3-clique, 4-clique, 7-clique and 7-star);
- The performance when the cluster diameter is high (long hops to pass message, like 4-circle, 7-circle, 7-bridge);

As for the data, we have created 1,000 accounts, with the genesis account having 100,000,000,000 tokens in the coinbase block. We divided the accounts into two groups (each group will have 500 accounts), the first group will participate in the ramp up step, which means the genesis account will distribute the tokens to these accounts. And for comparison we have issued four set of different size transactions (5000, 10000, 15000 and 20000) respectively. In the execution step, the first group of accounts will issue transactions to the second group of accounts, which constructs a bipartite spending graph. Since there are more transactions than the number of accounts, there will be double spend manners in this step. The number of threads in this procedure is equal to the number of nodes for each configuration. Jmeter is utilized as the driver to issue the transactions and Nginx is used to evenly and randomly distribute the requests to different nodes.

C. Results and Discussions

1) Block generation rate test: To test the block generation rate, we set each block in StreamNet to have only one transaction. And the performance on this configuration is as Figure 10 shows. To begin with, as the size of the cluster grows, the network as a whole will witness little performance loss on all of the data scales. This is because in IOTA implementation, when a node receives data from its peers, it will run a new POW to check the nonce and hash, by the time of this paper is written, we do not delve into the detail of whether this could be replaced with a constant time validation process, but the chance is high. As in the gossip network, a node can not distinguish if the block is generated by itself or if the block is already consumed, it will run redundant checks, this can explain the reason why as the complexity of the network topology grows, the TPS reduces (In figure we can see that TPS(4-circle)>TPS(4clique) and TPS(7-star)>TPS(4-bridge)>TPS(4-circle)>TPS(4-clique)). We can also observe the performance thrashing in the results, this is because for some times, the flood of the blocks will hit the memory limit of the weak nodes, which will cause the out of memory (OOM) error and let the node shut down. As the system scale and complexity of the network growing, we will see this performance thrashing issue more often.

In the experiment, we can also see that with the growth of the data, the reduction of the TPS ratio is less than the ratio of the growth of the data (1-2 times as opposed to 4 times). Considering the system is dealing a growing graph instead of a chain, and the complexity analysis in the previous section, our streaming algorithm shed the light on how to deal with the complexity of the growing DAG. And if we can find a way to snapshot the DAG in a decentralized way (for instance pushing the genesis ahead), the real stability can be achieved.

2) Bundle transaction test: By default, each block in StreamNet will support bundle transaction. Each block will contain 20 transactions. And the performance on this configuration is as Figure 11 shows. In this experiment, we can see weak scaling effect on every data set, as the performance will grow with the size of the clusters. This is because of the batching, the total number of the blocks that needs to be calculated is becoming less, and there will be less network message passing. In addition, the effect of network topology still holds, as the more complex the network structure, the less efficiently it will perform. One thing to note is that, as the data size grows, the performance degradation for the smaller cluster will be more than the larger cluster. This is because in the small cluster, small number of nodes will

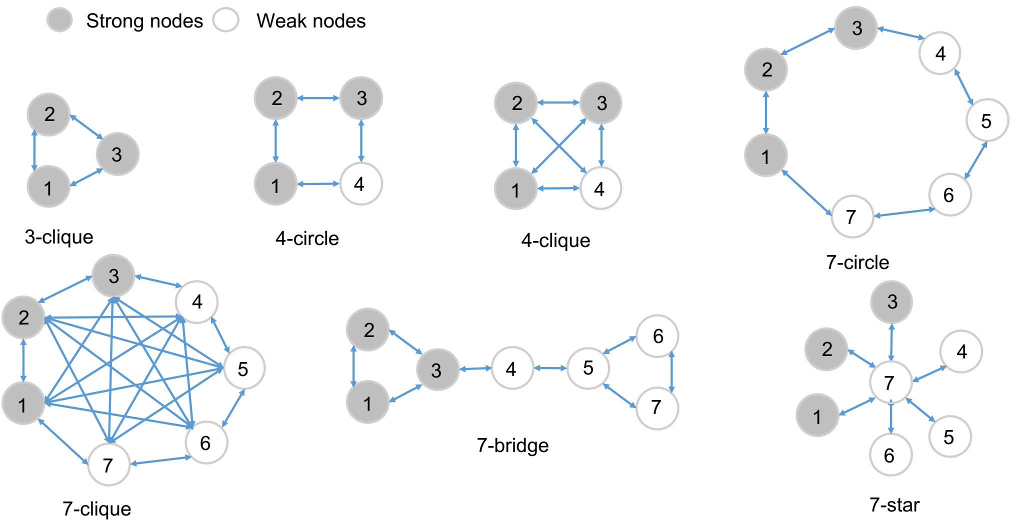


Figure 9. Cluster set up for different network topologies. For 3-cluster configuration only strong computing nodes are involved in the experiment, for 4-cluster and 4-circle, 3 strong and 1 weak computing node are involved in the experiment.

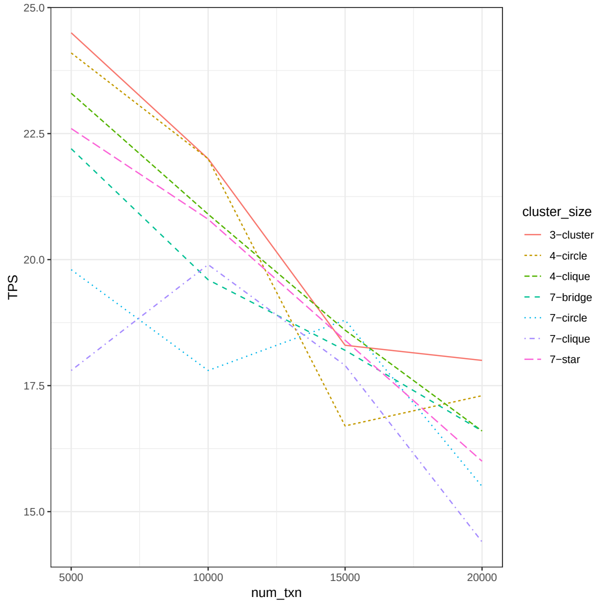


Figure 10. Experimental results for block generation rate.

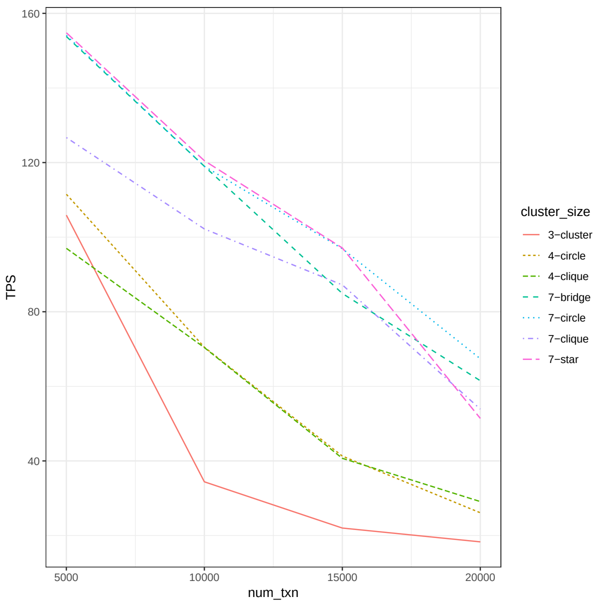


Figure 11. Experimental results for bundle transaction.

process large amount of data, which will soon flood the memories of the neighbor nodes, which will result in a sharp performance down turn. And in the bundle transaction test, we see much less performance thrashing because there are much less number of blocks passed in the network.

V. Conclusion

In this paper, we proposed a way to compute how to grow the blocks in the growing DAG based blockchain systems. And how to maintain the total order as the DAG structure is dynamically turning larger. We referred one of the earliest DAG implementation IRI to conduct our own experiments on clusters of different size and topology. Despite the network inefficiency in the IRI implementation, our method is proven to be able to tolerate the increasing complexity of the graph computation problems involved. This is due to the streaming graph computing techniques we have introduced in this paper.

References

[1] "Iota reference implementation," *https://github.com/iotaledger/iri*.

[2] "Rocksdb reference implementation," *http://rocksdb.org*.

[3] "Streamnet reference implementation," *https://github.com/triasteam/iri*.

[4] S. A. Abeyratne and R. P. Monfared, "Blockchain ready manufacturing supply chain using distributed ledger," 2016.

[5] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich *et al.*, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018, p. 30.

[6] A. Azaria, A. Ekblaw, T. Vieira, and A. Lippman, "Medrec: Using blockchain for medical data access and permission management," in *Open and Big Data (OBD), International Conference on*. IEEE, 2016, pp. 25–30.

[7] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, 1999, pp. 173–186.

[8] K. Christidis and M. Devetsikiotis, "Blockchains and smart contracts for the internet of things," *Ieee Access*, vol. 4, pp. 2292–2303, 2016.

[9] A. Churyumov, "Byteball: A decentralized system for storage and transfer of value," *URL https://byteball. org/Byteball. pdf*, 2016.

[10] A. Demers, D. Greene, C. Houser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," *ACM SIGOPS Operating Systems Review*, vol. 22, no. 1, pp. 8–32, 1988.

[11] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "Stinger: High performance data structure for streaming graphs," in *2012 IEEE Conference on High Performance Extreme Computing*. IEEE, 2012, pp. 1–5.

[12] D. Ediger, J. Riedy, D. A. Bader, and H. Meyerhenke, "Tracking structure of streaming social networks," in *2011 IEEE International Parallel & Distributed Processing Symposium Workshops and PhD Forum*. IEEE, 2011, pp. 1691–1699.

[13] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse, "Bitcoin-ng: A scalable blockchain protocol." in *NSDI*, 2016, pp. 45–59.

[14] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 51–68.

[15] O. Green, R. McColl, and D. Bader, "A fast algorithm for incremental betweenness centrality," in *Proceeding of SE/IEEE international conference on social computing (SocialCom)*, 2012, pp. 3–5.

[16] U. Klarman, S. Basu, A. Kuzmanovic, and E. G. Sirer, "bloxroute: A scalable trustless blockchain distribution network whitepaper."

[17] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, "Enhancing bitcoin security and performance with strong consistency via collective signing," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 279–296.

[18] K. Korpela, J. Hallikas, and T. Dahlberg, "Digital supply chain transformation toward blockchain integration," in *proceedings of the 50th Hawaii international conference on system sciences*, 2017.

[19] C. LeMahieu, "Nano: A feeless distributed cryptocurrency network," *Nano [Online resource]. URL: https://nano. org/en/whitepaper (date of access: 24.03. 2018)*, 2018.

[20] Y. Lewenberg, Y. Sompolinsky, and A. Zohar, "Inclusive block chain protocols," in *International Conference on Financial Cryptography and Data Security*. Springer, 2015, pp. 528–547.

[21] C. Li, P. Li, W. Xu, F. Long, and A. C.-c. Yao, "Scaling nakamoto consensus to thousands of transactions per second," *arXiv preprint arXiv:1805.03870*, 2018.

[22] C. Liu, D. Wang, and M. Wu, "Vite: A high performance asynchronous decentralized application platform."

[23] S. Matthew and E. T. Nuco, "Aion: Enabling the decentralized internet," *Aion project yellow paper*, vol. 151, pp. 1–22, 2017.

[24] J. MICHAEL, A. COHN, and J. R. BUTCHER, "Blockchain technology," *The Journal*, 2018.

[25] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of bft protocols," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 31–42.

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

[27] S. Popov, "The tangle," *cit. on*, p. 131, 2016.

[28] Y. Sompolinsky, Y. Lewenberg, and A. Zohar, "Spectre: Serialization of proof-of-work events: confirming transactions via recursive elections," 2016.

[29] Y. Sompolinsky and A. Zohar, "Phantom, ghostdag."

[30] ——, "Accelerating bitcoins transaction processing," *Fast Money Grows on Trees, Not Chains*, 2013.

[31] ——, "Secure high-rate transaction processing in bitcoin," in *International Conference on Financial Cryptography and Data Security*. Springer, 2015, pp. 507–527.

[32] A. Tapscott and D. Tapscott, "How blockchain is changing finance," *Harvard Business Review*, vol. 1, 2017.

[33] F. Tian, "An agri-food supply chain traceability system for china based on rfid & blockchain technology," in *Service Systems and Service Management (ICSSSM), 2016 13th International Conference on*. IEEE, 2016, pp. 1–6.

[34] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

[35] X. Yue, H. Wang, D. Jin, M. Li, and W. Jiang, “Healthcare data gateways: found healthcare intelligence on blockchain with novel privacy risk control,” *Journal of medical systems*, vol. 40, no. 10, p. 218, 2016.