ABACUS: Address-partitioned Bloom filter on Address Checking for UniquenesS in IoT Blockchain

ABSTRACT

DAG-based blockchain systems have been deployed to enable trustworthy peer-to-peer transactions for IoT devices. Unique address checking, as a key part of transaction generation for privacy and security protection in DAG-based blockchain systems, incurs big latency overhead and degrades system throughput.

In this paper, we propose a Bloom-filter-based approach, called ABACUS to optimize the unique address checking process. In ABACUS, we partition the large address space into multiple small subspaces, and apply one Bloom filter to perform uniqueness checking for all addresses in a subspace. Specifically, we propose a two-level address space mechanism so as to strike a balance between the checking efficiency and the memory/storage space overhead of the Bloom filter design. A bucket-based scalable Bloom filter design is proposed to address the growth of used addresses and provide the checking latency guarantee with efficient I/O access through storing all sub-Bloom-filters together in one bucket. To further reduce disk I/Os, ABACUS incorporates an in-memory write buffer and a read-only cache.

We have implemented ABACUS into IOTA, one of the most widely used DAG-based blockchain systems, and conducted a series of experiments on a private IOTA system. The experimental results show that ABACUS can significantly reduce the transaction generation time by up to four orders of magnitude while achieving up to 3X boost on the system throughput, compared with the original design.

KEYWORDS

DAG-based blockchain, Bloom filter, Performance boost

1 INTRODUCTION

Unique address checking is one key process to protect users' security and privacy when transactions are carried out in Directed-Acyclic-Graph (DAG) based blockchain systems that provides a promising direction to enable secure and trustworthy peer-to-peer transactions in the Internet of Things (IoT) environment [9, 13, 16, 20, 21]. It can guarantee that each transaction from each user account (wallet) is associated with a unique private/public key pair, in which the public and private keys will be utilized as the address and for the signature of the transaction, respectively. Specifically, for a transaction with a wallet, in a unique address checking process, each public key (i.e. the address of the transaction) generated by the seed and index of the wallet, needs to be checked to determine if it has been used (if yes, new public/private keys need to be generated until they are unique). With a unique address for each transaction, address forging can be easily detected and it is also extremely difficult to trace a user account based on the address information, thereby enhancing users' security and privacy. To guarantee address uniqueness, the address size must be large, and with the increase of the number of transactions, it becomes challenging to efficiently perform unique address checking.

Two approaches have been investigated for unique address checking. A database-based approach is commonly adopted in DAG-based blockchain systems. For instance, in IOTA, a representative DAG-based blockchain system, RockDB is adopted to store all used addresses [11]; unique address checking is accomplished through the time consuming database query process, by which one checking may take up to several seconds as shown in our experiments. Another approach is based on Bloom filters to perform efficient checking. For instance, in [23], a cuckoo-filter-based technique is proposed to accelerate the unique address checking process. The proposed technique, however, is based on an unrealistic assumption on which unique addresses are only required to be guaranteed in one snapshot period; that is, after a snapshot is performed once a year, all used addresses can be reused. Based on this assumption, the proposed technique simplifies the design of the Bloom filters but becomes impractical with various issues such as account forging and balance stealing [8].

To make the Bloom-filter-based approach practical, there are several challenges. First, to serve billions of IoT devices, a very large address space (e.g., several hundred bits) is required. It is a challenge for a Bloom-filter-based design to accommodate all possible addresses while efficiently performing uniqueness checking. Specifically, with such a gigantic address space, the size of a Bloom filter will become too big to be stored in memory. Therefore, an effective solution must be able to accomplish efficient checking even when the data of the Bloom filter is stored in storage devices. Second, as the address generation in an IoT blockchain is a dynamic growth process, it should waste too much memory/storage space if we utilize a static approach (such as in [14]) by pre-allocating a Bloom filter for the whole address space. To this end, a dynamic approach, which can adapt to the growth of used addresses, is more desirable. Furthermore, as used addresses increase, a fixed-sized Bloom filter may be overflowed, which needs to be taken into consideration as well.

To address these challenges, we propose an approach called ABACUS[1] (Address-partitioned Bloom filter on Address Checking for UniquenesS). In ABACUS, the core idea is to partition the large address space into multiple small subspaces, and for all addresses in a subspace, apply one Bloom filter to perform uniqueness checking. In such a way, by jointly optimizing the subspace size and Bloom filter design, one

[1]Our address-partitioned Bloom-filter-based approach divides the address space into two levels, which is analogous to an abacus that has multiple columns and is divided into two parts vertically.

Bloom filter can support all unique address checking of a subspace in the whole lifetime of a DAG-based blockchain system. To achieve this, the key is to minimize the checking latency with minimum memory/storage space, which is solved by several novel designs in ABACUS as follows.

First, we propose a two-level address space mechanism, by which the address space is first partitioned into multiple first-level subspaces, and each first-level subspace is further partitioned into multiple second-level subspaces. This two-level mechanism enables us to strike a balance between the checking efficiency and the memory/storage space overhead of the Bloom filter design. Specifically, for each second-level address subspace, one Bloom filter is adopted to perform uniqueness checking for all addresses under it; for each first-level address subspace, one read/write buffer is maintained and shared by all of its second-level subspaces for speeding up reads/writes of all Bloom filters under this first-level space. Therefore, we can effectively optimize the checking latency with minimum memory overhead.

Second, we propose a bucket-based scalable Bloom filter design. Our focus is on a dynamic scheme that can adapt to the growth of used addresses and provide the checking latency guarantee with efficient memory/storage space. Specifically, for each second-level address space, a scalable Bloom filter (S-BF), consisting of a series of one or more sub-Bloom-filters [3] (Sub-BFs), is adopted for unique address checking. Particularly, in ABACUS, an SBF contains one in-memory working Sub-BF and a set of on-disk immutable Sub-BFs. Here, the in-memory working Sub-BF is used to record bit updates when a new address is inserted, and once it is full, it becomes an immutable Sub-BF and is stored on disks. The set of on-disk Sub-BFs are all stored together into one bucket so they can be read to memory with one IO access. Unique address checking is accomplished by querying the in-memory working Sub-BF and all on-disk immutable Sub-BFs. In ABACUS, as all on-disk immutable Sub-BFs can be read together with one I/O access, the worst-case checking time can be guaranteed.

We have implemented ABACUS into the IOTA blockchain system (one of the most widely used DAG-based blockchain systems) and conducted a series of experiments on a private IOTA system [2]. The experimental results show that ABACUS can significantly enhance the latency of the unique address checking (up to four orders of magnitude) and the system throughput (up to 3X) compared with the IOTA with RocksDB for unique address checking.

Our main contributions are summarized as follows:

1) We propose an address-partitioned Bloom-filter approach for unique address checking in DAG-based IoT blockchain systems.
2) Two novel designs, a two-level address space mechanism and a bucket-based scalable Bloom filter, are provided to strike a balance between the checking latency and the memory/storage space.
3) The proposed technique has been implemented into the IOTA blockchain system, and its effectiveness has been demonstrated with a private IOTA system.

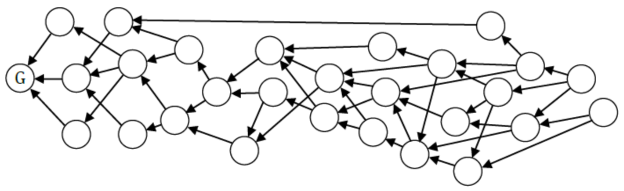


Figure 1: Basic structure of a DAG-based blockchain

The rest of this paper is organized as follows. Section 2 introduces the background and our motivation. Section 3 presents the design details of ABACUS. The experiments are presented in Section 4. Section 5 concludes this paper.

2 BACKGROUND AND MOTIVATION

2.1 DAG-based Blockchain

DAG-based blockchains such as IOTA [21], Byteball [9], NXT [20], OruMesh [16] and Nano [13], which utilize Directed-Acyclic-Graph rather than blocks as the basic data structure to manage transactions, are emerging and provides a promising direction to enable secure device-to-device transactions in an IoT environment [5, 10, 12, 18, 19, 22, 24, 26]. Figure 1 shows the basic architecture of a DAG-based blockchain. The root node of the DAG (G in Figure 1) acts as the genesis node. It defines the initial status of the whole blockchain. When a transaction comes, it will connect to one or more nodes in the DAG based on a consensus process. A consensus can be reached with various mechanisms such as voting [13] and connectivity calculation [21].

With such a structure, DAG-based block-chains can organize all transactions in a DAG rather than packing transactions into blocks in traditional block-based blockchain systems such as Bitcoin [15] and Etherunum [25]. Without utilizing block-based structures, DAG-based blockchains are not limited by block generation (e.g. block generation time and block size), so they have great potential to provide higher throughput and lower latency than block-based blockchains [11]. Furthermore, in DAG-based blockchains, transactions are validated and added with consensus mechanisms different from these in block-based blockchains where a computation-heavy and high-energy-consuming mining process is required. Without costly mining processes, transaction fees in DAG-based blockchains can be very cheap. Therefore, DAG-based blockchains are desirable for supporting device-to-device transaction in an IOT environment [6, 17].

2.2 Unique Address Checking

Unique address checking is used to guarantee that each transaction from each user account (wallet) is associated with a unique private/public key pair that can be generated by one-time signature schemes such as Winternitz one time signature [8]. Specifically, within the key pair, the public key will be used as the address of the transaction and the private key will be utilized for generating the signature of the transaction. As transactions are not directly associated with user accounts, this can effectively protect users' privacy. At the

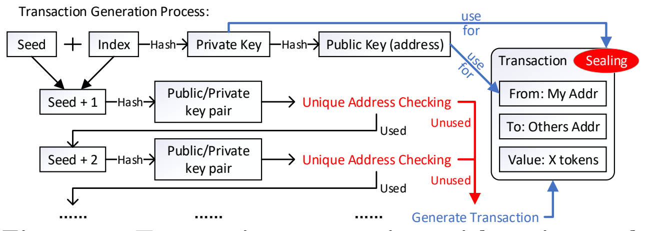


Figure 2: Transaction generation with unique address checking.

same time, transactions can be identified by unique addresses and verified based on private keys.

Figure 2 shows a typical procedure for transaction generation with unique address checking. For each transaction, Seed and Index are utilized to generate private/public key pairs, in which Seed is the ID of a wallet (a user account) and Index is an integer. A unique address checking process is performed to check if the public key, generated by Seed and Index, has been used; if yes, Index will be incremented and new public/private keys will be generated again. Only when public/private keys are unique, they can be used with the transaction.

2.3 Bloom Filter

A Bloom filter [7] is an array of m bits, which are initialized to 0. A set of k hash functions $H_1(x), H_2(x), ..., H_k(x)$ (e.g., using *murmur hash* [4] with k distinct seeds) are used to determine the corresponding k bits in the bit array. Specifically, when x is inserted, each hash function, i.e. $H_i(x)(1 \leqslant i \leqslant k)$, is used to calculate one index and the bits in the k indexes $H_1(x), H_2(x), ..., H_k(x)$ are set in the bit array. Upon a query for a key, based on the hash functions, if all the k corresponding bits are found to be set in the bit array, it indicates that the key is present in the filter.

Using Bloom filters in unique address checking can help reduce the latency [14]. It only requires several hash calculations (according to the number of hash functions) to determine whether an address has been used (i.e. if the address is present in a Bloom filter).

Note that a Bloom filter may have false positive but never has false negative. This means that if a Bloom filter answers "No" for a given address, this address must not be used; on the other hand, if it answers "Yes", the address may or may not be used. Therefore, Bloom filters are suitable for unique address checking.

With a "false positive" case, an address will be wasted. To reduce address wastage, we can control the false positive rate of a Bloom filter. Given an m-bit Bloom flter, with an expected filling ratio f (the number of 1's in the m bits), suppose that at most n unique keys (a.k.a the capacity of the Bloom filter) are recorded, the false postive rate P can be calcuated as follows:

$$\ln(P) = -\frac{m}{n} \ln f \times \ln(1-f) \tag{1}$$

2.4 Motivation

Unique address checking can be accomplished utilizing databases. For instance, in IOTA, RocksDB is adopted to store all used addresses and unique address checking is performed through a query process [1]. Our preliminary experimental results show that the database query process for unique address checking introduces big overhead in transaction generation. As shown in Figure 3, based on the private IOTA system we have built (the detailed configuration can be found in Section 4), by measuring and analyzing transaction latencies with various benchmarks, we found that among all transactions, on average, the database query process occupies more than 80% of the total time of a transaction, thus severely degrading the system performance.

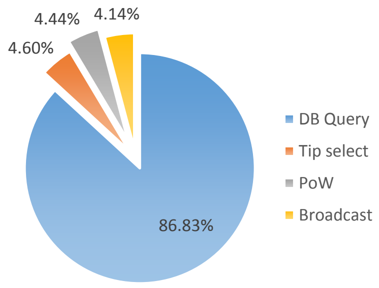


Figure 3: The time composition of attaching a transaction to DAG-based blockchain.

In this paper, we aim to optimize the unique address checking process and improve the system throughput. Utilizing Bloom filters on unique address checking provides a promising direction to solve these issues. However, the previous Bloom-filter-based solution [23] is based on an unrealistic assumption and cannot be used in practice. Thus, we propose a new Bloom-filter-based approach.

3 ABACUS

In this section, we present the design details of ABACUS. We will first introduce the overall architecture and then describe several important designs including address partition (in Section 3.1), bucket-based scalable Bloom filters (in Section 3.2), and read/write buffer design (in Section 3.3). Finally, we provide an example to illustrate how to partition the address space in ABACUS.

Table 1: The parameters for Bloom filter calculation

Param	Description
P	The final false positive rate
P_i	False positive rate of each sub-Bloom-filter
p	The expected fill ratio of one sub-Bloom-filter
r	The error reduction factor
l	The number of sub-Bloom-filters
N	The total number of keys in scalable Bloom filter
n_i	The number of keys in each sub-Bloom-filter
k	The number of hash functions
M	The total size (bits) of scalable Bloom filter
m_i	The size (bits) of each sub-Bloom-filter

The architecture of ABACUS is shown in Figure 4. It mainly includes two data structures (i.e., SBF and Sub-BF)

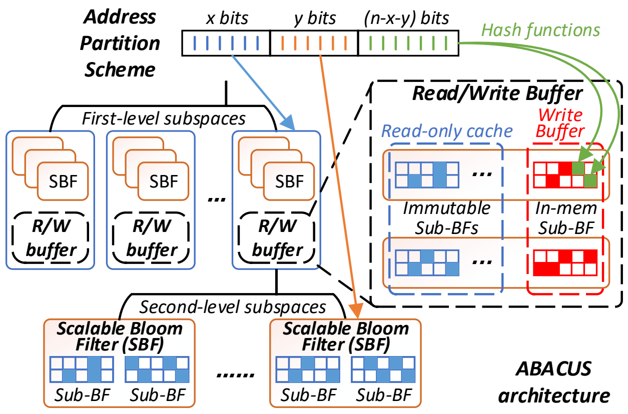


Figure 4: System architecture of ABACUS.

and three function modules (i.e., an address partition module, a bucket-based SBF module, and a read/write buffer module). The address partition module separates an incoming address into three parts. The higher two parts are utilized to partition the address space into two levels. For each first-level address subspace, one read/write buffer is maintained and shared by all of its second-level subspaces for speeding up reads/writes of all Bloom filters under this first-level space. The lowest part of an address is hashed onto an SBF with several pre-defined hash functions for unique address checking. An SBF is composed of one in-memory Sub-BF and multiple on-disk immutable Sub-BFs. All on-disk immutable Sub-BFs of an SBF are stored together into one bucket that can be read to the memory with one I/O access. The meanings of notations used in this section are summarized in Table 1.

3.1 Address Partition Scheme

Since the address length of the DAG-based blockchain system can be of several hundred bits, it consumes too much memory to maintain a Bloom filter to serve for all the possible addresses. Thus, the address partition scheme divides an address into three parts and organize the Bloom filters with a two-level address space as shown in Figure 5.

Suppose the address is of N bits, the first part contains x bits, the second part contains y bits, and the last part contains $N-x-y$ bits. Based on the first-level prefix (x bits), the address space is partitioned into multiple first-level subspaces; based on the second-level prefix (y bits), each first-level subspace is further partitioned into multiple second-level subspaces. For each first-level address subspace, one read/write buffer is maintained and shared by all of its second-level subspaces for speeding up reads/writes of all Bloom filters under this first-level space. For each second-level address subspace, one Bloom filter is adopted to perform uniqueness checking for all addresses (denoted by $N-x-y$ bits) under it.

In practice, however, even the last $N-x-y$ bits of an address are too many to build a Bloom filter for all its possible values in advance. Hence, we adopt the scalable Bloom Filter (SBF) design. In SBF, we will first create an m bits Sub-BF according to the length $N-x-y$ with a pre-defined false

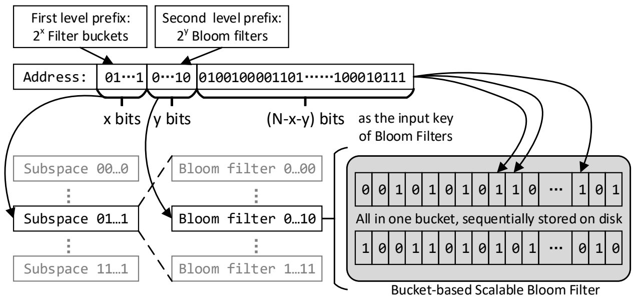

First level prefix: 2^x Filter buckets
Second level prefix: 2^y Bloom filters
Address: 01⋯1 0⋯10 0100100001101⋯⋯100010111
x bits
y bits
(N-x-y) bits
as the input key of Bloom Filters
Subspace 00…0
Subspace 01…1
Subspace 11…1
Bloom filter 0…00
Bloom filter 0…10
Bloom filter 1…11
0 0 1 0 1 0 1 0 1 1 0 ⋯ 1 0 1
All in one bucket, sequentially stored on disk
1 0 0 1 1 0 1 0 1 0 1 ⋯ 0 1 0
Bucket-based Scalable Bloom Filter

Figure 5: Address Partition Scheme.

positive rate (FPR). One in-memory Sub-BF will be updated based on the hash value of the allocated addresses. Once an in-memory Sub-BF achieves its capacity limitation, it becomes immutable and is stored into the bucket as an on-disk Sub-BF, and a new in-memory Sub-BF will be created to serve the following update operations. During a unique address checking, once an SBF is located with the higher two parts, all the in-memory and on-disk Sub-BFs in the SBF need to be checked to determine the existence status of the address. Hence, the in-memory Sub-BF is the basic update unit while all Sub-BFs of an SBF is the basic checking unit.

3.2 Bucket-based Scalable Bloom Filter

The SBF function module is used to fully utilize the allocated Sub-BF space under the FPR constraint. The generation, update, and checking process of the Sub-BFs in one SBF is presented as follows.

For a given SBF, its FPR is the product of all its Sub-BFs' FPR. Thus, we need carefully configure the space of each Sub-BF to achieve a lower FPR and make the final SBF's FPR converged. Suppose P_0 is the FPR of the first Sub-BF. Each newly added Sub-BF will have a reduced FPR compared with the previous one, which can be represented as $P_i = P_0 \times r^i$. The final FPR of the whole scalable Bloom filter can be calculated as:

$$P = 1 - \prod_{i=0}^{l-1} \left(1 - P_0 r^i\right). \tag{2}$$

The upper bound of the final FPR can be estimated by Equation 3 as follows:

$$P \leqslant \lim_{l \to \infty} \sum_{i=0}^{l-1} P_0 r^i = P_0 \frac{1}{1-r} \tag{3}$$

For example, if $P_0 = 0.05\%$ and the reduction rate $r = 0.95$, the final FPR will not exceed 1%.

According to Equation 1, the size of a Sub-BF is determined by the number of inserted keys, the FPR, and the expected filling ratio p:

$$m_i \approx n_i \frac{-\ln P_i}{\ln p \ln(1-p)}. \tag{4}$$

To minimize the storage space m_i, the expected filling ratio of each Sub-BF is $\frac{1}{2}$. It means when 50% of the current

Bloom filter is filled, we need to generate a new one and makes the old one as read-only.

For a newly allocated address, the SBF will update the corresponding bits of its in-memory Sub-BF. The update process is as shown in Algorithm 1. The k hash functions will set the corresponding k bits as "1" according to the lower $N-x-y$ bits of the address. After that, we will check if the filling ratio of the Sub-BF has reached 50%. If yes, we will generate a new Sub-BF with the same size as previous ones with $FPR = P_0 \times r^n$.

Algorithm 1 Address insertion in ABACUS

```
Input:
    The number of sub-Bloom-filters, $n$;
    Sub-Bloom-filters, $SubBF_i, i \in (1, n)$;
    The fill ratio of current Bloom filter, $p$;
    The number of hash functions, $k$;
    Hash functions, $h_i, i \in (1, k)$;
    The address needs to be inserted, $addr$.
Algorihthm:
1: Select $SubBF_n$ for key insertion;
2: Insert $addr$ into $SubBF_n$ using $h_1(addr), ..., h_k(addr)$;
3: Update $p = \frac{the\ number\ of\ 1\ in\ SubBF_n}{the\ size\ of\ SubBF_n}$
4: if $p \leqslant \frac{1}{2}$ then
5:     continue;
6: else
7:     Allocate a new sub-Bloom-filter $SubBF_{n+1}$;
8:     $n \Leftarrow n + 1$;
9: end if
```

Algorithm 2 Address checking in ABACUS

```
Input:
    The number of sub-Bloom-filters, $n$;
    Sub-Bloom-filters, $SubBF_i, i \in (1, n)$;
    The fill ratio of current Bloom filter, $p$;
    The number of hash functions, $k$;
    Hash functions, $h_i, i \in (1, k)$;
    The address needs to be checked, $addr$.
Algorihthm:
1: $i \Leftarrow n$;
2: while $i \geqslant 1$ do
3:     Check $addr$ in $SubBF_i$ using $h_1(addr), ..., h_k(addr)$;
4:     if $found$ then
5:         return '-1'; //already used
6:     else
7:         continue;
8:     end if
9:     $i \Leftarrow i - 1$;
10: end while
11: return '1'; //address available
```

When performing unique address checking, we will go through all the Sub-BFs in one SBF to check if the target address exists. We will load all on-disk Sub-BFs into memory and check both in-memory and on-disk Sub-BFs in parallel. If all the Sub-BFs return "Non-exist", the address can be used safely. Otherwise, a new address need to be created.

By using SBF instead of a plain Bloom filter, the filter size can dynamically grow when the number of inserted addresses increases. Thus, we do not need to pre-define a large filter but only provide an upper bound of the scalable Bloom filter size. The configuration of this upper bound will be discussed in Section 3.4.

3.3 Read/Write Buffer Design

ABACUS is partially maintained in storage to save memory space. When performing unique address checking, the SBF with all Sub-BFs (in-memory and on-disk) is the basic read unit and the in-memory Sub-BF is the basic write unit. Thus, an address checking process may incur several time-consuming I/O operations. To further mitigate disk I/Os, we propose an Read/Write buffer. Specifically, for each first-level address subspace, one read/write buffer is maintained and shared by all of its second-level subspaces for speeding up reads/writes of all Bloom filters under this first-level space.

As shown in Figure 6, the read/write buffer includes an in-memory Sub-BF-based write buffer and an on-disk Sub-BF-based read-only cache. For each first-level subspace, a read/write buffer is maintained. The write buffer contains all in-memory Sub-BFs from each second-level subspace under a first-level subspace. On the contrary, the read-only cache only stores several hot SBFs under a first-level subspace. Specifically, a read-only cache has several slots and each slot is used to accommodate all the on-disk Sub-BFs of a recently accessed SBF. The size of the write buffer is of $2^x \times 2^y \times size(SubBF)$. Suppose each SBF contains m Sub-BFs at most, the size of the read-only cache is of $number\ of\ slots \times 2^x \times 2^y \times (m-1) \times size(SubBF)$.

With the write buffer, all update operations are performed in memory. Once an in-memory Sub-BF in the the write buffer is full (the filling ratio reaches $\frac{1}{2}$), it will be flushed to disks. If its located SBF is buffered in the read-only cache, the SBF slot will also be updated; correspondingly, a newly generated in-memory Sub-BF of this SBF will be added into the write buffer.

For a unique address checking process, once we get the SBF number with the first two parts of the address, ABACUS will first check its in-memory Sub-BF in the write buffer to see if the address exist. If yes, the address cannot be used. Otherwise, ABACUS will search the read-only cache. With cache hits, ABACUS will directly perform the checking process based on the corresponding slot in the read-only cache. Only with cache misses, the corresponding on-disk Sub-BFs of the SBF will be read out from the disk to perform the checking process.

For the write buffer, since it contains all the in-memory Sub-BFs from each SBF under a first-level subspace, there is no need to do replacement. For the read-only cache, since the actual number of Sub-BFs in each SBF is different (at most $m-1$), when performing cache replacement, we consider both the slot's LRU information and its size.

In a DAG-based blockchain system, multiple servers are coordinated to maintain the DAG structure with all transactions. ABACUS will be deployed with the blockchain system

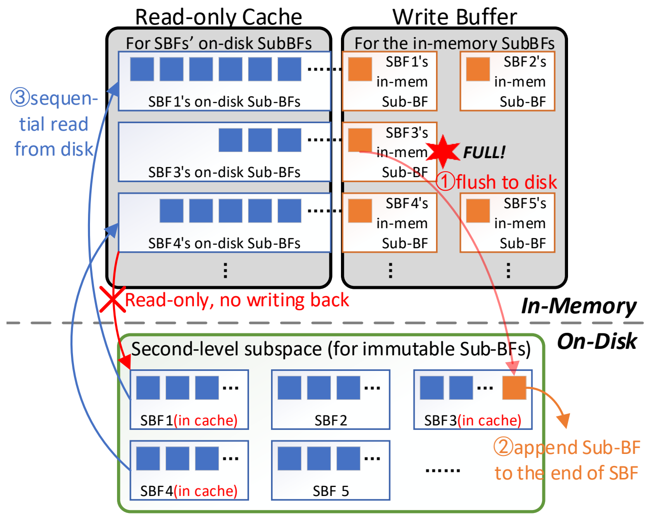


Figure 6: Read/Write Buffer Design.

on all of these servers. Once power failure happens for one server, ABACUS can be synchronized from other servers. Thus, the crash recovery of ABABUS can be managed together with other components of a DAG-based blockchain system.

3.4 Example for Address Partition

In this section, we provide an example to illustrate how to determine the parameters in ABACUS (i.e. the first- and second-level prefixes). ABACUS only requires the expected throughput and usage time to determine those prefixes.

Taking a DAG-based blockchain with 200 TPS as an example, it uses around 6,307,200,000 addresses per year. Suppose that we want to maintain the system for 20 years, then our Bloom filter should support the checking capacity of $20 \times 6,307,200,000 = 126,144,000,000$ addresses.

Considering the I/O efficiency, we fix the size of each in-memory Sub-BF as one memory page (4KB, 32768 bits) and the size of one SBF (contains all on-disk Sub-BFs) as the I/O prefetch size (128KB). According to Equation 4, we can get the number of keys that each sub-Bloom-filter can hold:

$$n_i \approx 32768 \times \frac{\ln\frac{1}{2} \times \ln\frac{1}{2}}{-\ln P_i} \approx \lfloor\frac{15743}{-\ln P_i}\rfloor = \lfloor\frac{15743}{-\ln P_0 r^i}\rfloor. \quad (5)$$

Let the starting P_0 be 0.05% and the reduction rate $r = 0.95$ in $P_i = P_0 \times r^i$ which will limit the final FPR at 1% according to Equation 3. We can get the approximate total number of keys in one SBF:

$$\begin{aligned} N = \sum_{i=0}^{\infty} n_i &\approx \sum_{i=0}^{\infty} \lfloor\frac{15743}{-(\ln P_0 + i \ln r)}\rfloor \\ &= \sum_{i=0}^{\infty} \lfloor\frac{15743}{-(\ln 0.0005 + i \ln 0.95)}\rfloor \end{aligned} \quad (6)$$

By combining Equation 5 and 6, we can get $N = \sum_{i=0}^{31} n_i \approx 60180$. It means when one SBF is full, it can handle 60180 keys in total. To serve the addresses in 20 years, we will need $126,144,000,000 \div 60180 \approx 2,096,112$ SBFs in total, which requires at least 21 bits (21 bits can represent $2^{21} = 2,097,152$ SBFs). Finally, ABACUS will occupy $2,097,152 \times 128KB \approx 256GB$ disk space and handle $2,097,152 \times 60180 = 126,206,607,360$ keys in total.

Because we maintain all the on-disk Sub-BFs (4KB each) of one SBF in one bucket (128KB), we can guarantee that using one I/O access can read out all those on-disk Sub-BFs. Taking commercial SSD for the latency analysis, one 4KB page-aligned read operation will take around $70-80ns$. However, due to the prefetch mechanism, the following sequential 4KB read operations within one prefetch zone (128KB) will only take around $10-15ns$ according to our experiment. Thus, the worse case of reading out all the on-disk Sub-BFs can be guaranteed at $80 + 31 \times 15 = 545ns$.

For the usage after 20 years, when a bucket is full, we can allocate more buckets to store the on-disk Sub-BFs of an SBF. Although this may lead to more storage space usage, more IO access overhead and higher false positive rate, ABACUS can still support effective unique address checking.

4 EVALUATION

We have implemented ABACUS with IOTA and conducted a series of experiments on a private IOTA system. In this section, we present and analyze the experimental results. We first introduce the IOTA implementation and experimental setup in Sections 4.1 and 4.2, respectively, and then present the experimental results in terms of the unique address checking latency (in Section 4.3), transaction generation time (in Section 4.4), system throughput (in Section 4.5) and storage space (in Section 4.6). Finally, we analyze various overheads in Section 4.7.

4.1 Implementation in IOTA

IOTA uses trytes[2] instead of bytes. Thus, for our address partition scheme, we will divide our address space according to trytes. In order to compare with the current RocksDB-based solution (used by IOTA), we set the parameters of ABACUS based on 6 times of the IOTA main network throughput (7 TPS), so ABACUS can support up to 42 TPS. Similar to the configurations in Section 3, the system is expected to operate for 20 years and the SBF and Sub-BF sizes are set as 128 KB and 4 KB, respectively. The false positive rate is set at 1%. Based on these configurations, three trytes and one trytes in an IOTA address are used as the first-level and second-level prefixes, respectively.

4.2 Experiment setup

The experiments have been conducted on a private IOTA blockchain network with a series benchmark tools [2]. The private IOTA network consists of 35 nodes, and each node is equipped with i5-6260u CPU and 4GB DDR4 memory. All nodes are separated into three roles. One node is used as a coordinator to send milestones to the private IOTA network, there are up to three full nodes, and the others are

[2]IOTA is based on trinary instead of binary, a tryte consists 3 trits and a trit has values $-1, 0, 1$. So the maximum value of a tryte is $3^3 = 27$, represented by a character in the tryte alphabet [A−Z, 9].

used as clients. Each node is used to simulate several clients and all the clients can issue transactions to the whole IOTA network in a configured frequency, while all the full nodes are equipped with IRI (IOTA Reference Implementation) to receive and handle those transactions. The load can be adjusted by increasing or decreasing the number of clients. In the evaluation, we call the original IOTA implementation with RocksDB as "Baseline" and call our proposed approach as "ABACUS".

We run the evaluation with different incoming transaction speeds. For each incoming transaction speed, we begin with an empty ABACUS (or an empty RocksDB with "Baseline") and run for 3 hours. Each experiment has been run for 5 times, and then the average results are obtained and utilized for comparison.

4.3 Unique address checking latency

We first evaluate the unique address checking time with various incoming transaction speeds. The evaluation results are shown in Figure 7(a), in which ABACUS significantly reduces the unique address checking process latency by 99% compared with Baseline. For instance, as the TPS reaches to 10, the latencies of Baseline increase from around 100 ms to several hundreds milliseconds, while the latencies of ABACUS are maintained at around 0.1 ms.

It can be observed that ABACUS and Baseline have different latency trends as the incoming request speed increases. As shown in the Figure 7(a), the address checking latency of "Baseline" increases with the increment of the incoming requests. The reason is that with more transactions, the database becomes larger, thus, making the query latency longer.

For ABACUS, as the incoming transaction speed increases, the latency keeps decreasing. This is because in ABACUS, each SBF is organized and stored into one bucket, and one bucket with all Sub-BFs of an SBF are stored together on disks. Hence, each bucket needs to allocate enough disk space (i.e. 128KB) for their SBFs when the first address is inserted into an SBF. The allocation time is included in the latency. Since the addresses (i.e. the public key generated from Seed) are evenly distributed, there will be a lot of allocation cost, thus increasing the latencies of ABACUS at the very beginning. After running a period of time, all the disk spaces are allocated, so the latency will drop back to its original value. With a higher incoming transaction speed, as disk spaces can be allocated faster, the allocation latency will have less impact on the average unique address checking latency.

4.4 Transaction generation time

We further evaluate the transaction generation time. During the transaction generation process, since a sender may need to collect tokens from multiple addresses, many unique address checking queries may be performed within one transaction generation. Thus, the transaction generation time may be much larger than the unique address checking latency.

We first configure the private blockchain system with one full node and perform the evaluation. Figure 7(b) shows the evaluation result. Compare to Baseline, ABACUS reduces the transaction generation time by up to four orders of magnitude. This is because the database-based solution needs to search one key in the whole database, which may trigger many time consuming I/O read operations. Furthermore, if one address has been proved to be unique, it needs to be inserted into the database with I/O write operations.

For ABACUS, all the Bloom filter update operations are absorbed in our proposed SBF-oriented write buffer, while the read operations could also benefit from the read-only cache. In addition, since the size of an SBF is increased dynamically based on our scalable Bloom filter design, each cache miss will not always incur a 128KB I/O read, and instead, it only needs to read all of the current on-disk Sub-BFs, which can help reduce the checking latency as well.

We further configure the private blockchain system with three full nodes. As shown in Figure 7(c), with more nodes, the transaction generation requests will be distributed among them, which accelerates the transaction generation time of both the implementations. Nevertheless, the generation time of ABACUS is still much lower than that of Baseline.

4.5 System Throughput

We evaluate the system throughput by configuring the system with one and three full nodes. As shown in Figure 8(a), the throughput of ABACUS benefits a lot from the reduced transaction generation time. With one full node configuration, ABACUS achieves up to 5X throughput improvement compared with Baseline as the incoming transaction speed increases.

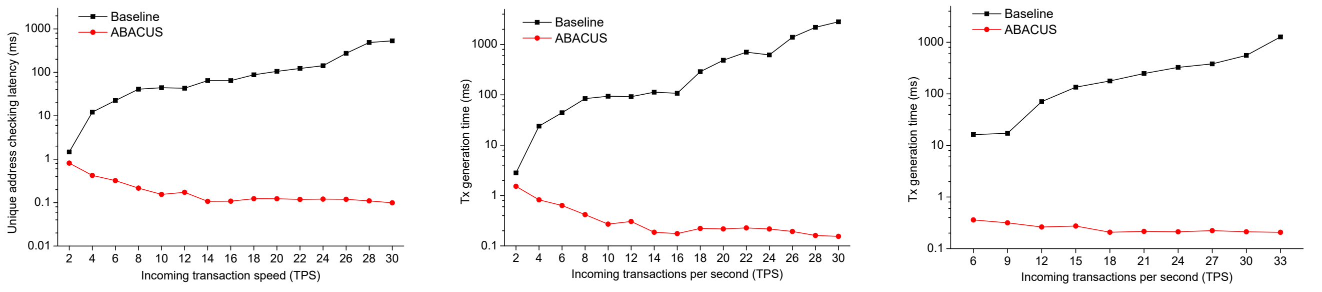
With three full nodes configuration, as shown in Figure 8(b), the throughput improvement is less than the configuration with one full node. This is mainly because the synchronization time between multiple full nodes will consume some computation resources, thus increasing the overall time of attaching a transaction to the DAG. In general, ABACUS achieves up to 3X throughput improvement compared with Baseline.

4.6 Storage space analysis

In our implementation, we suppose that ABACUS is designed for 20 years with 6 times of the current IOTA throughput (42 TPS). Based on this configuration, IOTA will consume around 1,324,512,000 addresses per year. Each address consists 81 trytes, which need $405bits = 51bytes$[3] for storage. Thus, for traditional IOTA implementation that adopts RocksDB as the database, at least $1,324,512,000 \times 51Bytes \approx 63GB$ storage space are consumed per year.

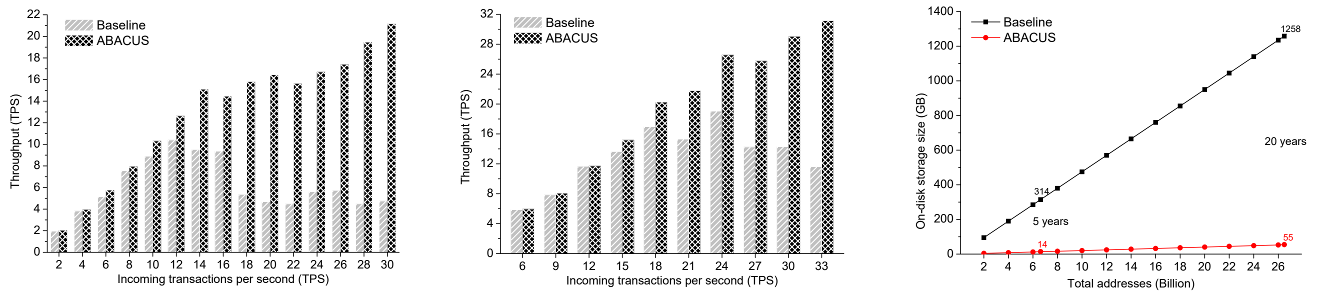
Compared with Baseline, theoretical analysis shows ABACUS can reduce the space usage by 95% (around 14GB vs. 314GB) for the first 5 years usage. Furthermore, with the system expanded, the difference between those two storage costs becomes even larger. To support 20-year usage, ABACUS will use around 55GB while Baseline needs around 1.2 TB, as shown in Figure 8(c).

[3]1 tryte has 27 values which need at least 5 bits ($2^5 = 31$) to present.



(a) Unique address checking latency (b) Tx generation time with one full node (c) Tx generation time with three full nodes

Figure 7: The time comparison between ABACUS and the RocksDB-based solution (baseline).



(a) Throughput with one full node (b) Throughput with three full nodes (c) Storage space usage

Figure 8: The performance and storage cost of ABACUS vs. the RocksDB-based solution (baseline).

4.7 Overhead analysis

Table 2: Overhead on CPU and memory

	Memory usage	CPU usage
ABACUS	20.2035%	57.8671%
Baseline	17.9330%	76.1317%

Memory overhead. The memory overhead of ABACUS mainly comes from the in-memory Sub-BF write buffer and the read-only SBF cache. As shown in Table 2, for the first 3 hours running with 21 incoming TPS, our memory usage is 20% while the Baseline uses 18%. However, with more addresses inserted into ABACUS, more SBFs will be allocated and their last Sub-BFs will be cached in memory. As the blockchain system expands, the in-memory Sub-BF write buffer will consume 2GB memory at the worst case.

The SBF read-only cache stores parts of the SBFs in memory and its size can be adjusted according to the system status and device memory limitation. For the meta-data part, as we only have $27^3 = 19683$ first-level subspaces and each one only occupies 4 bytes for its entry point, the total size of meta-data is only $19683 \times 4Byte \approx 77KB$, which is negligible.

CPU overhead. For CPU part, ABACUS will consume some computation resources for the Bloom filter's hash value calculation. However, with faster unique address checking, a higher incoming TPS can be processed smoothly by ABACUS. Table 2 also shows the CPU usage between ABACUS and Baseline. As some transactions are blocked due to the low throughput of Baseline, the CPU usage of ABACUS is much less than the Baseline.

Address wastage. Due to the false positive rate of a Bloom filter, an unused address may be regarded as "used" which may cause some address wastage. We manually fill up one SBF and monitor the false positive rate of this SBF. The results show the FPR finally reaches 2.6% which is double than the pre-fixed 1%. That is mainly because of the collision of the Bloom filter's hash functions, making the actual FPR higher than its theoretical value. More Sub-BFs will increase the false positive rate of an SBF as well. How to reduce FPR will be investigated in the future.

5 CONCLUSION

Unique address checking introduces big overheads when transactions are generated in DAG-based blockchain systems. Traditional database-based solutions slow down this critical process, making the blockchain system suffer from low performance. This paper proposes a novel Bloom-filter-based approach for unique address checking. The experimental results show that the proposed technique can significantly optimize the unique address checking latency and system throughput.

REFERENCES

[1] 2016. IOTA Reference Implementation. https://github.com/iotaledger/iri.

[2] 2019. IOTA-Benchmark-Tool. https://github.com/wty715/IOTA-Benchmark-Tool.

[3] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. 2007. Scalable bloom filters. *Inform. Process. Lett.* 101, 6 (2007), 255–261.

[4] Austin Appleby. 2008. MurmurHash. *URL https://sites. google. com/site/murmurhash* (2008).

[5] Vidal Attias and Quentin Bramas. 2018. Tangle analysis for IOTA cryptocurrency. (2018).

[6] Paulo C Bartolomeu, Emanuel Vieira, and Joaquim Ferreira. 2018. IOTA Feasibility and Perspectives for Enabling Vehicular Applications. In *2018 IEEE Globecom Workshops (GC Wkshps)*. IEEE, 1–7.

[7] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[8] Johannes Buchmann, Erik Dahmen, Sarah Ereth, Andreas Hülsing, and Markus Rückert. 2011. On the security of the Winternitz one-time signature scheme. In *International Conference on Cryptology in Africa*. Springer, 363–378.

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

[10] Zhongli Dong, Emma Zheng, Young Choon, and Albert Y Zomaya. 2019. DAGBENCH: A Performance Evaluation Framework for DAG Distributed Ledgers. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 264–271.

[11] B Kusmierz. 2017. The first glance at the simulation of the Tangle: discrete model. *http:/iota. org* (2017).

[12] Bartosz Kusmierz, Philip Staupe, and Alon Gal. 2018. *Extracting tangle properties in continuous time via large-scale simulations*. Technical Report. working paper.

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

[14] Lailong Luo, Deke Guo, Richard TB Ma, Ori Rottenstreich, and Xueshan Luo. 2018. Optimizing Bloom filter: Challenges, solutions, and comparisons. *IEEE Communications Surveys & Tutorials* 21, 2 (2018), 1912–1949.

[15] Satoshi Nakamoto et al. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).

[16] OruMesh. 2017. Whitepaper: OruMesh. *orumesh.com [online] https://orumesh.com/whitepaper2.0.pdf* (2017).

[17] Seongjoon Park and Hwangnam Kim. 2019. DAG-Based Distributed Ledger for Low-Latency Smart Grid Network. *Energies* 12, 18 (2019), 3570.

[18] Seongjoon Park, Seounghwan Oh, and Hwangnam Kim. 2019. Performance Analysis of DAG-Based Cryptocurrency. In *2019 IEEE International Conference on Communications Workshops (ICC Workshops)*. IEEE, 1–6.

[19] Huma Pervez, Muhammad Muneeb, Muhammad Usama Irfan, and Irfan Ul Haq. 2018. A comparative analysis of DAG-based blockchain architectures. In *2018 12th International Conference on Open Source Systems and Technologies (ICOSST)*. IEEE, 27–34.

[20] Serguei Popov. 2016. A probabilistic analysis of the nxt forging algorithm. *Ledger* 1 (2016), 69–83.

[21] Serguei Popov. 2016. The tangle. *cit. on* (2016), 131.

[22] W Qian, W Tianyu, et al. 2019. Re-Tangle: A ReRAM-based processing-in-memory architecture for transaction-based blockchain. In *IEEE ICCAD*. IEEE.

[23] Sehrish Shafeeq, Sherali Zeadally, Masoom Alam, and Abid Khan. 2019. Curbing Address Reuse in the IOTA Distributed Ledger: A Cuckoo-Filter-Based Approach. *IEEE Transactions on Engineering Management* (2019).

[24] Qian Wang, Zhiping Jia, Tianyu Wang, Zhaoyan Shen, Mengying Zhao, Renhai Chen, and Zili Shao. 2019. A Highly Parallelized PIM-based Accelerator for Transaction-based Blockchain in IOT Environment. *IEEE Internet of Things Journal* (2019).

[25] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151 (2014), 1–32.

[26] Manuel Zander, Tom Waite, and Dominik Harz. 2019. DAGsim: Simulation of DAG-based distributed ledger protocols. *ACM SIGMETRICS Performance Evaluation Review* 46, 3 (2019), 118–121.