BC-Store: A Scalable Design for Blockchain Storage

I-Te Chou
Department of Computer Science & Information Engineering; Advanced Institute of Manufacturing with High-tech Innovations (AIM-HI)
National Chung Cheng University, Taiwan
yanxion@alum.ccu.edu.tw

Hung-Han Su
Department of Computer Science & Information Engineering; Advanced Institute of Manufacturing with High-tech Innovations (AIM-HI)
National Chung Cheng University, Taiwan
607410047@alum.ccu.edu.tw

Yu-Ling Hsueh
Department of Computer Science & Information Engineering; Advanced Institute of Manufacturing with High-tech Innovations (AIM-HI)
National Chung Cheng University, Taiwan
hsueh@cs.ccu.edu.tw

Chih-Wen Hsueh
Department of Computer Science & Information Engineering, National Taiwan University, Taiwan
cwhsueh@csie.ntu.edu.tw

ABSTRACT
The blockchain technology has obtained significant success in the past decades. However, a serious underlying problem still exists in the blockchain system – data bloating. In the blockchain system, each (full) node must store the full data set in blockchain history, incurring significant storage pressure in the initial synchronization process and the following maintenance of the blockchain system. Data bloating is a challenging problem to be confronted in the immediate future of blockchain. To address this problem, in this paper, we introduce the BC-Store framework that deploys a data accessing model on an IPFS-cluster system to classify the hot and cold blockchain data. The hot data are stored in the local cache, whereas the cold data are stored in the IPFS cluster, thereby substantially shortening the blockchain initial synchronization time and saving a considerable amount of data storage. Empirical experimentation shows that our framework can reduce the local storage size from over 265GB to 4GB with a hit ratio of 77% for Bitcoin without significant performance degradation with the whole data shared in an IPFS cluster.

CCS Concepts
• **Information systems → Information storage systems → Storage architectures→ Distributed storage**

Keywords
Blockchain; IPFS; Hot/cold identification; Data bloating

1. INTRODUCTION
The blockchain technology originated with the Bitcoin white paper [1] published by Nakamoto on October 31, 2008. It is different from the traditionally centralized mechanism for electronic cash. Bitcoin utilizes the cryptography to generate trusted blocks of transactions. Each added block contains a hash of the previous block. Numerous blocks are linked after the genesis block to form a chain of blocks. The Bitcoin stores all transaction history in the blockchain. The blockchain also represents a public ledger that records Bitcoin transactions. Bitcoin first utilizes the blockchain technology for nodes in the Bitcoin network to store data in a decentralized peer-to-peer (P2P) manner.

In the blockchain storage, each node stores the same ledger which increases in size over time of mature blocks, which are not the latest 100 blocks and are very likely to be immutable. Taking Bitcoin as an example, since 2009, the accumulated blockchain size has grown over 265GB in 2020. It is estimated that the size will reach 500GB by 2025. As a result, under the current mechanism, each node needs to store almost all the same huge data. As blockchain adopted more popularly in the future, these transaction data will grow even more dramatically. Therefore, the problem of data bloating in the blockchain is a primary problem to solve now. In this paper, we aim to reduce the amount of transaction data stored in each node so that the blockchain initial synchronization time can be reduced as well. Nevertheless, the storage scalability of the whole blockchain network can be improved.

To solve the data bloating problem, Zheng et al. [2] proposed combining two existing technologies: the blockchain framework and the interplanetary file system (IPFS) technologies. The proposed framework uploads blockchain transaction data to the IPFS system, which responds to the IPFS hash after receiving the data. If other nodes need these transaction data, they can use this IPFS hash value to request a transaction datum from the IPFS system.

However, in the framework proposed by Zheng et al. [2], all transaction data are uploaded to the IPFS system. This means it is only possible to download any transaction data from the IPFS system. As transaction data continue to increase, the bandwidth demands and the network traffic also increases. Hence, the proposed framework is not scalable, even the IPFS is considered scalable [3, 4].

Therefore, in this work, we propose a scalable framework for blockchain storage termed BC-Store, which is composed of a novel data block accessing model and an IPFS-cluster system. Figure 1 shows our system architecture, where Bitcoin is utilized as an underlying platform. The main objective is to identify the hot and

IECC 2020, July 8–10, 2020, Singapore, Singapore

ACM ISBN 978-1-4503-7770-6/20/07…$15.00
DOI: https://doi.org/10.1145/3409934.3409940

cold blockchain data. The hot data are the frequently used data in the blockchain, while the cold data are the infrequently used. In our proposed framework, the hot data stay in the local database, called LevelDB, whereas the cold data are uploaded to the IPFS system such that the initial synchronization time, and the storage size of the blockchain data can be significantly reduced. This solves the data bloating problem in blockchains.

The structure of the paper is as follows. In Section 2, we cover important related works and the existing IPFS implementation. In Section 3, we introduce our BC-Store framework, which is composed of the block accessing model and the IPFS-cluster system. In Section 4, we detail the block accessing model and the proposed algorithms. In Section 5, we describe the IPFS-cluster system within the BC-Store framework. In Section 6, we show the experimental results and the analysis of the model effectiveness. In Section 7, we conclude this paper.

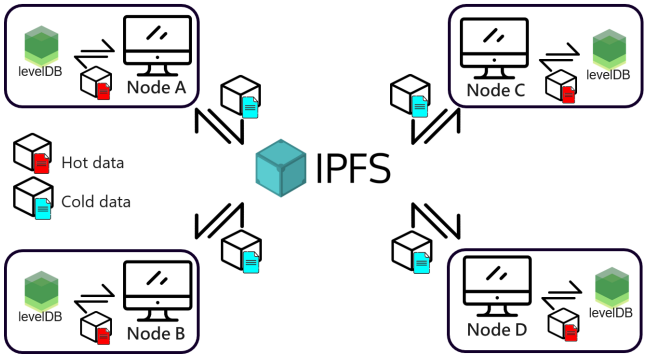

Figure 1. The block accessing model

2. RELATED WORK

In this section, we survey the related studies [5,6] that address the blockchain bloating problem and the implementation of IPFS. To continue the research on the blockchain bloating problem, we investigated related studies in the hot/cold data identification and the field of IPFS implementation.

To solve the problem of cold and hot data recognition, Kim et al. [7] proposed a compression ratio and a sector size of the requested data as the identification criteria. A compression ratio was used to classify hot/cold data, while the sector size was considered for hot/cold data prediction. The proposed scheme makes a considerable contribution to improve the performance of the NAND flash memory. Hsu et al. [8] proposed an approach which learns from the history data access behavior, including the patterns of read, write, modify and execute operations on the file access. A prediction engine is based on these features. The system is composed of deep neural networks (DNN), the support vector machine (SVM), and user ranking functions. By successful classification of hot/cold data, the system obtains significant computational cost reduction. Hsieh et al. [9] proposed a highly efficient method for online hot data identification. Multiple independent hash functions were adopted to reduce the chance of false identification of hot data. This provides excellent performance for hot data identification. In this approach, the method utilizes the decay period to process the data access count. The values of all counts are divided by 2 in terms of their bits. If the most significant bits of the hashed values contain a non-zero bit value, it is defined as hot data. This method increases the hot data identification performance. Due to the limitation of the traditional LRU and LFU methods, Afify et al. [10] proposed a novel algorithm called HC_Apriori to distinguish hot and cold data. However, their algorithm can only be applied to a certain application domain (i.e., frequent item set mining) such that it is not applicable to blockchain.

The blockchain technology can be utilized in different fields, such as agriculture, artificial intelligence (AI), and the Internet of Things (IoT). However, the storage issue still exists in the blockchain systems on these fields. For instance, Bitcoin takes a significant amount of disk space for storage resulting in data bloating. Therefore, the block data accumulated over 265GB currently reduces the synchronization performance of the blockchain system. To solve the data bloating problem, Zheng et al. [2] proposed a blockchain data storage model based on IPFS and Bitcoin. By utilizing the feature of content addressing in IPFS, every miner stores all the transaction data in the IPFS network after verifying all of the transactions, and preserves the hash key returned from IPFS. This change not only alleviates the pressure of blockchain storage, but also solves the synchronization problem. As a result, the consistency of the data in each node is achieved. In their experiments, the compression ratio reaches 0.0817, which means this model improves the storage efficiency of the Bitcoin system. To alleviate the storage pressure in Ethereum, Norvill et al. [11] proposed an IPFS-based Ethereum storage solution to reduce the size in the main chain. The smart contracts are compiled to bytecode and stored in the data field of the contract creation transaction (CCTX) in the main chain. The solution replaced the original data property of CCTX with IPFS hashes and set up the threshold to 33 bytes. According to the experiment result, this solution reduced the chain size by 93.86%. Although the above works achieved great improvement, the data allocation policy for IPFS was not addressed. Furthermore, it may incur notably time latency to the system because of storing all the transaction data without the allocation policy on IPFS.

3. SYSTEM ARCHITECTURE

The objective of this paper is to design a scalable framework for solving the data bloating problem in blockchains. We propose the hot/cold data identification methods and integrate them into the block accessing model to access the hot block data from the LevelDB and the cold block data from the IPFS-cluster system. Figure 2 shows the reading procedure, and the writing procedure is illustrated in Figure 3.

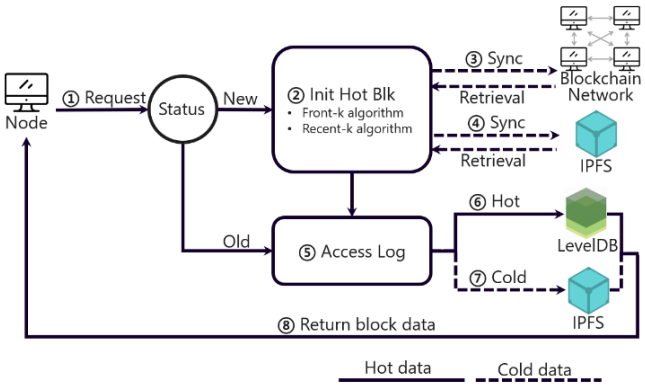

Figure 2. The block accessing model: reading procedure

For the reading procedure as illustrated in Figure 2, the block accessing model firstly examines the node status (either new for being accessed or old as not), when there is a request to the block accessing model from the requiring node (see Step (1)). If the status is new, the block accessing model executes the *Init* algorithm (see Step (2)) and broadcasts the request for synchronizing the hot block

data files from the neighboring nodes in blockchains (see Step (3)). If there is no available hot data in the blockchain network, the block accessing model requests the IPFS-cluster system to download the block data file via the *Front-k* or the *Recent-k* algorithms. (see Step (4)). We detail the *Front-k* algorithm and *Recent-k* algorithm in the following sections. After the *Init* algorithm is completed, the model records the access count of the block data file into the access log (see Step (5)). The access log is then leveraged for the *MRU* algorithm to track the least used block data file. We detail the *MRU* algorithm in the following sections. Finally, after reading the hot data from LevelDB (see Step (6)), the block accessing model returns the block data to the requiring node (see Step (8)). Otherwise, if the node status is old, the block accessing model records the access count of the block data file into the access log (see Step (5)) and returns the block data from LevelDB when it is available (see Step (6)). Otherwise, the requested data are returned from IPFS-cluster system to the requiring node (see Step (7)).

For the writing procedure as illustrated in Figure 3, the block accessing model firstly examines the storage space by comparing with the initial configuration when the node requests the block accessing model (see Step (1)). If the storage space is full, the model determines the least used block data file by analyzing the access log file (see Step (2)) and use the most recently used *(MRU)* algorithm to remove the infrequently used block data files such that we can guarantee enough storage space for downloading the block data files when necessary. After removing the least used block data file, the block accessing model writes the data to LevelDB (see Step (3)). Otherwise, if the storage is not full, the block accessing model directly writes the data to LevelDB (see Step (4)).

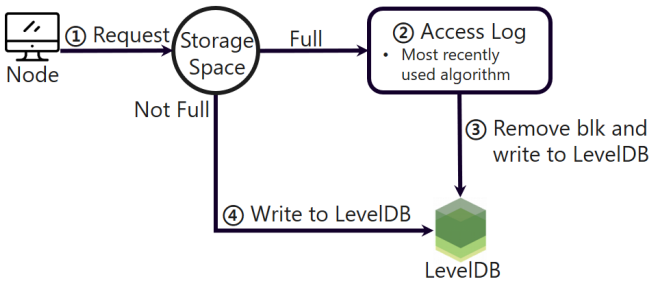

Figure 3. The block accessing model: writing procedure

4. BLOCK ACCESSING MODEL

In our block accessing model, the major components consist of the initial synchronization process of a new node (the *Init* algorithm in Section 4.1), the writing procedure (the *WriteBlock* algorithm in Section 4.2), and the reading procedure (the *ReadBlock* algorithm in Section 4.3). The details of each algorithm are described as follows, where "blk" is short for a block file.

4.1 The Init Algorithm

We propose the *Init* algorithm to allow a new node to synchronize the hot data with other nodes either from the blockchain network or IPFS-cluster. To design the hot/cold data identification mechanism, we analyze the Bitcoin transaction data for one week, and the access distribution of the block data files is shown in the Y axis of Figure 4, where #0 is the block data file that contains the genesis block and the #1800 block data file contains the latest data blocks. We can see that the most frequently accessed files are the recent block data files and those that are close to the genesis block during the week. Based on the abovementioned observation, we propose the front-k and recent-k algorithms together with the *Init* algorithm for providing the identification of hot data in a global manner.

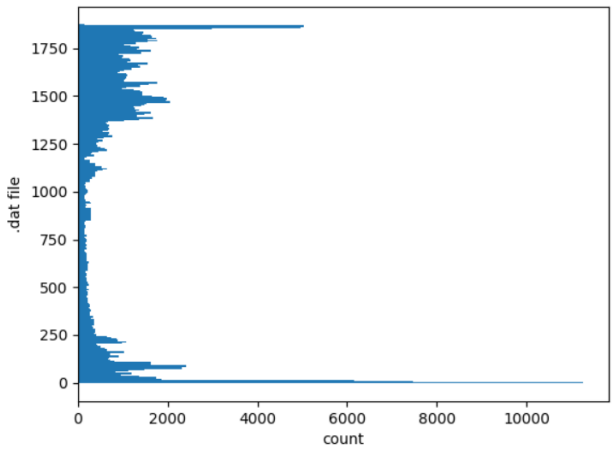

Figure 4. Block access distribution

As Algorithm 1 *Init* shows in Line 1, we initialize the *validity* as ***FALSE***. In Line 2, if the node status is new, then execute Line 3 to examine the hot data availability from the neighbor nodes. Otherwise, the *Init* algorithm is ended by returning ***FALSE***. Once the node status is checked as new, the function *neighborHasBlk(node)* in Line 3 examines whether the neighbor nodes store the hot block data. Next, the requiring node synchronizes the hot block data files when the neighbor node has hot block data and sets the *validity* as ***True*** to represent the success of synchronization in Line 4. Otherwise, in Line 6, the requiring node synchronizes the block data files from IPFS-cluster based on the front-k algorithm and the recent-k algorithms and sets the *validity* as ***True*** to represent the success of synchronization. Finally, in Line 9, the *Init* algorithm returns the value of *validity*.

Algorithm 1: Init($node$, $storage_{Limit}$)

input : *node*: a node that issues a request to read block data files. $storage_{Limit}$: node storage limit.
output: *validity*: a Boolean value to indicate whether the block data files are successfully retrieved.

```
1 Initialize validity to be FALSE;
2 if node.status == "new" then
3     if neighborHasBlk(node) == TRUE then
4         validity = syncBlkFromNeighbor(node, storage_Limit);
5     else
6         validity = syncHotBlkFromIPFS(storage_Limit);
7     end
8 end
9 return validity
```

4.1.1 The Front-k Algorithm

$$F_k = \left(Blk_{total} \times \lambda\, Threshold_{front} \right) \quad (1)$$

$$blk(i) = \begin{cases} Hot & 0 \leqq blkNum(i) < F_k \\ Cold & otherwise \end{cases} \quad (2)$$

In Eq. (1), the formula of the front-k algorithm is based on a threshold F_k to calculate the data range for identifying hot data blocks. The Blk_{total} represents the total number of block data files. $\lambda\, Threshold_{front}$ is a metric that indicates the percentage of a set of block data files containing hot data. The F_k can be determined by Blk_{total} times $\lambda\, Threshold_{front}$. Consequently, in Eq. (2), the front F_k data blocks, which range from #0 up to #F_k, are classified as hot data.

4.1.2 The Recent-k Algorithm

$$R_k = \left(Blk_{total} \times \lambda\, Threshold_{recent} \right) \quad (3)$$

$$blk(i) = \begin{cases} Hot \\ Cold \end{cases} \quad \begin{matrix} Blk_{total} - R_k < blkNum(i) \leqq Blk_{total} \\ otherwise \end{matrix} \qquad (4)$$

In Eq. (3), the recent-k algorithm formulates the recent k blocks as hot data blocks based on R_k. The Blk_{total} represents the total number of block data files. $\lambda\, Threshold_{recent}$ is a metric that indicates the percentage of a set of block data files containing hot data. The R_k can be determined by Blk_{total} times $\lambda\, Threshold_{recent}$. Therefore, in Eq. (4), the hot data blocks are the most recent R_k consecutive blocks.

4.2 The WriteBlock Algorithm

We propose the WriteBlock algorithm to process the data writing procedure as listed in Algorithm 2. In Line 1, the WriteBlock algorithm firstly checks the storage space by comparing with the initial storage size ($Storage_{limit}$). If the storage space is full, the WriteBlock algorithm executes the MRU algorithm to retrieve the infrequently used block data file in Line 2, and removes the least used block data file by executing the function $removeBlkFromLevelDB()$ in Line 3. We detail the MRU algorithm in the 4.2.1 section. $accessLog$ is used to record the access count of each corresponding block data file. Finally, in Line 5, the algorithm writes the data to LevelDB. Through the above operations in the algorithm, we can guarantee enough storage space for the local block data files.

Algorithm 2: WriteBlock(*node*, $storage_{Limit}$)

input : *node*: a node that issues a request to read a block.
$block_{Height}$: the height of a rquested block.
output: *blockData*: the data block with the specified $block_{Height}$.

```
1 if checkStorageSize() >= Storage_Limit then /* Storage is full */
2 |  replaceBlk = MRU(accessLog);
3 |  removeBlkFromLevelDB(replaceBlk);
4 end
5 writeToDisk(block_Height);
```

4.2.1 The MRU Algorithm

We use the most recently used (*MRU*) algorithm to select the block data files that have not been accessed recently to guarantee that there is enough hot data storage space for storing the block data files. The pseudo code of the *MRU* algorithm is listed in Algorithm 3. In Algorithm 3, $hotBlk$ represents the block data files that are stored in the LevelDB. In Lines 4-10, the *MRU* algorithm retrieves the minimum block data access count and records the corresponding $blkNum$, which represents the number in the block data file name. Finally, in Line 11, the algorithm returns the *minblkNum* of the minimum access count.

Algorithm 3: MRU(*accessLog*)

input : *accessLog*: the log file that records the access count of each corresponding block data file.
output: *minBlkNum*: the file number of the blk file to be replaced.

```
1  Let hotBlk be the block data files stored at the LevelDB ;
2  Initialize minBlkNum to be -1;
3  Initialize minCount to be 0;
4  foreach blk ∈ hotBlk do
5  |  blkNum = getblkNum(blk);
6  |  if minCount > accessLog[blkNum] then
7  |  |  minBlkNum = blkNum;
8  |  |  minCount = accessLog[blkNum];
9  |  end
10 end
11 return minBlkNum;
```

4.3 The ReadBlock Algorithm

We propose the ReadBlock algorithm to process the data reading procedure. As Algorithm 4 shows in Line 1, $Get\_blk(Block_{height})$ returns the file number of the block data by giving a specific block height. In Line 2, $access$Log[ ] is used to record the access count of each corresponding block data file. If the $LevelDBhasBlk()$ function returns true in Line 3, it represents that the LevelDB contains the hot data. Therefore, in Line 4, the system reads the block data from LevelDB by performing the function $readFromLevelDB()$ that returns $blockData$, which is considered as hot data. Otherwise, in Line 6, the system reads the block data from IPFS by performing the function $readFromIPFS()$ to obtain $blockData$, which is considered as the cold data. Finally, in Line 8, the algorithm returns the block data to the requiring node.

Algorithm 4: ReadBlock($block_{Height}$)

input : $block_{Height}$: the height of a rquested block.
output: *blockData*: the data block with the specified $block_{Height}$.

```
1 blkNum = getBlk(Block_height);
2 accessLog[blkNum] += 1;
3 if LevelDBhasBlk(block_Height) then
4 |  blockData = readFromLevelDB(block_Height) ;      /* hot data */
5 else
6 |  blockData = readFromIPFS(block_Height) ;        /* cold data */
7 end
8 return blockData;
```

5. IPFS-cluster SYSTEM

IPFS is deployed for handling the cold data in this work. To improve the efficiency of IPFS, we adopt a three-replication strategy by manipulating the IPFS-cluster system, where each file in the cluster has three replicas conserved by three different nodes. Therefore, the minimum number of nodes in the cluster should be greater than three nodes. The storage limit of each node is adjusted dynamically to prevent the failure of the data provider. Furthermore, to avoid frequent change to the storage limit, we define the stable interval to constraint the timing of modification. Eq. (5) defines the storage limit for the nodes in the IPFS-cluster system, where the storage limit of a cluster node is denoted by *S_limit*. *SizeOfBC* denotes the data size of the blockchain. *R* implies the replication factor, and *N* is the number of nodes in the cluster. If the S_limit_{new} *is* greater than the stable interval *SIV,* all the nodes in the system update their storage limit to the new S_limit_{new}. Otherwise, the current storage limit of the cluster remains unchanged.

$$S\_limit_{new} = \begin{cases} \frac{SizeOfBC \times R}{N} \\ S\_limit_{old} \end{cases} \quad \begin{matrix} if\ S\_limit_{new} > SIV \\ otherwise \end{matrix} \qquad (5)$$

In our system design, the users can choose to join the IPFS-cluster to conserve the block data for our system. As extra incentive, the cluster users are given coins according to the size of the block data they conserve. Figure 5 indicates the configuration of the IPFS-cluster system in our framework. The nodes in the red rectangle represents the cluster nodes. Due to their contribution on the storage space, they can obtain extra rewards besides the mining in the blockchain. The design of our IPFS-cluster system is to increase the incentive for the full nodes to join our framework so that our framework can achieve better stability and security.

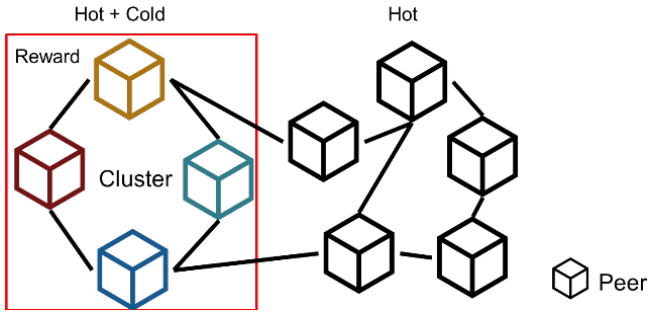

Figure 5. The IPFS-cluster system configuration

6. EXPERIMENTS

We use block data in Bitcoin MainNet as our testing data and analyze the storage size, hit ratio, and the effective access time for our system. The experimental results are presented in the following subsections.

6.1 Storage Size

We adopt 4GB as the storage limit for every node in our system. Figure 6 indicates the comparison of the Bitcoin storage size and our framework. The blue line represents the Bitcoin storage size, where the red line represents the storage size of our system. According to the experiment in Figure 6, we can discover that when the data size is less than the storage limit, both Bitcoin and our framework store all of the data in the local database. When the data size is greater than the storage limit, our framework stops increasing the storage size by utilizing the MRU algorithm, where Bitcoin still increases the storage capacity of the hard disk. Therefore, each node can greatly reduce the file storage capacity and save the hard disk storage space through our framework.

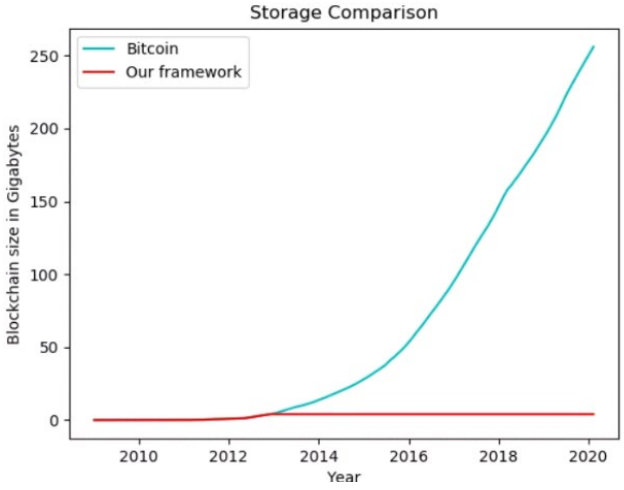

Figure 6. The comparison of the block storage before and after data processing (storage limit is 4GB)

6.2 Hit Ratio

As illustrated in Figure 7, we can discover that the hit ratio increases along with the storage limit. The hit ratio is approximately equal to 77% when the storage limit is 4GB. When the storage limit is 40GB, the hit ratio is approximately equal to 86%. The hit ratio is approximately equal to 94% when 100GB is initiated as the storage limit for our system. In this case, we select 4GB as the storage limit because the setting results in less initial synchronization time and still maintains a reasonable hit ratio.

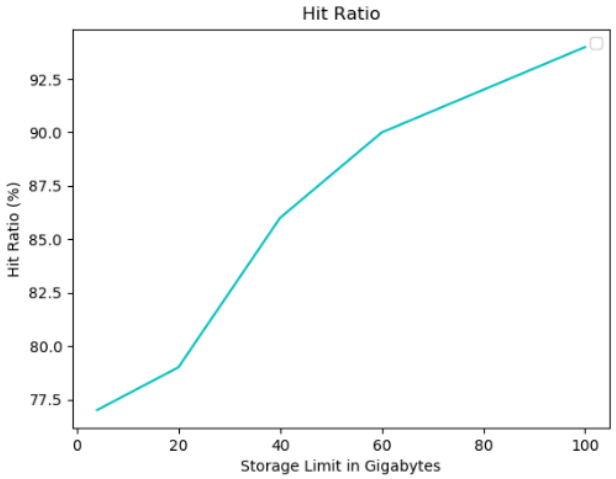

Figure 7. The hit ratio of the storage limit

6.3 Effective Access Time

The performance evaluation of our framework is analyzed by calculating the effective access time (EAT). In our experiment, we simulate the behavior that the users download the block data file from the IPFS cluster via the WriteBlock algorithm. We formulate the EAT formula as shown in Eq. (6).

$$EAT = hot\ hit\% \times local\ access\ time + cold\ hit\% \times (IPFS\ search\ time + IPFS\ download\ time) \quad (6)$$

$$Local\ access\ time = search\ time + loading\ time \quad (7)$$

We set the fixed hit ratio to 77%, which represents that the storage limit per node is 4GB. To measure the effective access time for our system, we firstly evaluate the access performance of the IPFS-cluster system as shown in Table 1. To consider the different network environments for the users, we measure the access performance of the IPFS cluster with the network speed 100Mbps, 10Mbps, and 5Mbps. Furthermore, we define the time of retrieving files in IPFS-cluster as the total access time which is composed of the search time and the download time. The search time is the time that searches the files in the IPFS-cluster nodes based on the Kademlia distributed hash tables, whereas the download time is the time that downloads the data from the provider in the IPFS-cluster system. Besides, the local access time of the Bitcoin node also needs to be evaluated in the EAT formula. Therefore, we define Eq. (7) to measure the local access time. The evaluation performance of Bitcoin local access time is shown in Table 2. After obtaining the elapsed time of the IPFS cluster and the local access time of the Bitcoin system, the EAT can be obtained via Eq. (6). Table 3 indicates the effective access time of our framework at three different network speeds. According to the experimental results, we can conclude that the EAT at the network speed of 100Mbps has the most effective performance, which is 2.946544821 seconds in our framework.

Table 1. The access performance of the IPFS-cluster system

Network speed	Search time	Download time	Access time
100Mbps	0.256877955s	12.554122s	12.811s
10Mbps	5.132529405s	69.906470595s	75.039s
5Mbps	7.243331462s	183.033668538s	190.277s

Table 2. The Bitcoin access performance

Local search time	Local loading time	Local access time
0.0016846ms	0.0175638ms	0.0192483ms

Table 3. The effective access time of our framework

100Mbps	10 Mbps	5 Mbps
2.946544821s	17.2589848212s	43.7637248212s

7. CONCLUSION

In this paper, we design a BC-Store framework that contains the block accessing model and the IPFS-cluster storage system for blockchain. The IPFS-cluster system is used in our framework to conserve the block data since it achieves an excellent performance on distributed storage. We can discover from the result that our model substantially shortens the initial synchronization time in blockchains and saves a considerable amount of data storage space. Since the Bitcoin system frequently loads the hot data which are stored on the local LevelDB, the block data access performance remains competitive to that of the existing Bitcoin system. The cold data are stored on IPFS-cluster system, which triggers the Bitcoin system to download the transaction data from the IPFS-cluster system when necessary. Consequently, the data storage size can be reduced from over 265GB to a user specified storage capacity, thereby saving a considerable amount of data storage space. In our experiments, the BC-Store framework can reduce the storage space from 265GB to 4GB. The hit ratio of hot data achieves 77% for Bitcoin. The storage scalability is feasible and improved.

8. ACKNOWLEDGEMENT

This work was financially supported by the Advanced Institute of Manufacturing with High-tech Innovations (AIM-HI) from the Featured Areas Research Center Program within the framework of the Higher Education Sprout Project by the Ministry of Education (MOE) in Taiwan, and the National Science Council under the Grants MOST 108-2221-E-194-040-MY3.

9. REFERENCES

[1] Nakamoto, Satoshi, 2008, *Bitcoin: A Peer-to-Peer Electronic Cash System*.

[2] Q. Zheng, Y. Li, P. Chen, X. Dong, 2018, *An Innovative IPFS-Based Storage Model for Blockchain*, in Proceedings of the 2018 IEEE/WIC/ACM International Conference on Web Intelligence (WI)

[3] J. Benet, 2014, *IPFS-Content Addressed, Versioned, P2P File System (DRAFT 3)*.

[4] P. Maymounkov, D. Mazieres, 2002, *Kadmelia: A peer-to-peer information system based on the xor metric*, in Proceedings of the International Workshop on Peer-to-Peer Systems (pp. 53-65). Springer, Berlin, Heidelberg.

[5] M. Amelchenko and S. Dolev, 2017, "Blockchain abbreviation: Implemented by message passing and shared memory (Extended abstract)," in Proceedings of the 2017 IEEE 16th International Symposium on Network Computing and Applications (NCA).

[6] Shlomi Dolev and Yuval Poleg, 2017, Beyond Replications in Blockchain - On/Off-Blockchain IDA for Storage Efficiency and Confidentiality (Brief Announcement), in Proceedings of the third International Symposium on Cyber Security Cryptography and Machine Learning (CSCML).

[7] K. Kim, S. Jung, and Y. H. Song, 2011, *Compression ratio based hot/cold data identification for flash memory*, in Proceedings of the IEEE International Conference on Consumer Electronics (ICCE)

[8] Y.-F. Hsu, R. Irie, S. Murata, M Matsuoka, 2018, *A Novel Automated Cloud Storage Tiering System through Hot-Cold Data Classification*, in Proceedings of the IEEE 11th International Conference on Cloud Computing (CLOUD)

[9] J. W. Hsieh, L. P. Chang, T. W. Kuo, 2005, *Efficient On-line Identification of Hot Data for Flash-Memory Management*, in Proceedings of the 2005 ACM Symposium on Applied Computing

[10] G. M. Afify, A. E. Bastawissy, O. M. Hegazy, 2016, *Identifying Hot/Cold Data in Main-Memory Database using Frequent Item set Mining*, in Proceedings of the International Journal of Enhanced Research in Management & Computer Applicationsr

[11] R. Norvill, B. B. Pontiveros, R. State, & A. Cullen, 2018, *IPFS for reduction of chan size in Ethereum*, in Proceedings of the 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) (pp. 1121-1128). IEEE.