Block-LSM: An Ether-aware Block-ordered LSM-tree based Key-Value Storage Engine

Zehao Chen*, Bingzhe Li†, Xiaojun Cai*, Zhiping Jia*, Zhaoyan Shen*, Yi Wang‡, Zili Shao§
*School of Computer Science and Technology, Shandong University, Qingdao, China
†School of Electrical and Computer Engineering, Oklahoma State University, Stillwater, OK, USA
‡College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China
§Department of Computer Science and Engineering, The Chinese University of Hong Kong, NT, Hong Kong

2021 IEEE 39th International Conference on Computer Design (ICCD) | 978-1-6654-3219-1/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICCD53106.2021.00017

***Abstract*—Ethereum as one of the largest blockchain systems plays an important role in the distributed ledger, database systems, etc. As more and more blocks are mined, the storage burden of Ethereum is significantly increased. The current Ethereum system uniformly transforms all its data into key-value (KV) items and stores them to the underlying Log-Structure Merged tree (LSM-tree) storage engine ignoring the software semantics. Consequently, it not only exacerbates the write amplification effect of the storage engine but also hurts the performance of Ethereum. In this paper, we proposed a new Ethereum-aware storage model called Block-LSM, which significantly improves the data synchronization of the Ethereum system. Specifically, we first design a shared prefix scheme to transform Ethereum data into ordered KV pairs to alleviate the key range overlaps of different levels in the underlying LSM-tree based storage engine. Moreover, we propose to maintain several semantic-orientated memory buffers to isolate different kinds of Ethereum data. To save space overhead, Block-LSM further aggregates multiple blocks into a group and assigns the same prefix to all KV items from the same block group. Finally, we implement Block-LSM in the real Ethereum environment and conduct a series of experiments. The evaluation results show that Block-LSM significantly reduces up to 3.7× storage write amplification and increases throughput by 3× compared with the original Ethereum design.**

***Index Terms*—Ethereum, Storage Engine, KV Store**

I. Introduction

Ethereum [1], as one of the most popular Blockchain systems, has been widely deployed on various application scenarios, including medicine [2], economics [3], Internet of Things [4], software engineering [5], and digital assets [6], etc. To guarantee the consistency of the Ethereum network, each full node must synchronize all the network transaction data to the local disk. However, as the network scales up, its data size is becoming pretty large (so far, the amount of data of Ethereum has reached about 800 GB [7]). Thus, the data storage engine of Ethereum must be carefully designed to optimize the system storage I/O consumption.

To mitigate the excessive data load of the Blockchain systems, some works [8], [9] adopt the idea of partial storage and propose to reduce the storage size by removing a certain part of the block. There are some other works [10], [11], which propos to change the smart contract structure or encoding approach to compress the data in blocks to reduce the storage overhead. However, although these methods can mitigate the storage pressure to a certain extent, they lead to information loss, making the query process more complicated. Besides, these methods only considered the Blockchain data structures and ignored the in-coordination of Blockchain semantics with its underlying storage engine.

Corresponding author: Xiaojun Cai, School of Computer Science and Technology, Shandong University, Qingdao, China. E-mail: xj_cai@sdu.edu.cn.

Currently, the underlying storage engine in Ethereum uses the write-friendly LSM-tree [12] based key-value (KV) storage engine to maintain all their data. During the synchronization process, the full nodes would obtain the latest blocks from the network, uniformly transform the data into KV items based on hash functions, and finally store them in the underlying LSM-based storage engine. However, this straightforward approach throws away some important Blockchain semantics, such as the block sequence, resulting in substantial meaningless I/O overheads in the underlying storage engine. In this paper, to the best of our knowledge, for the first time, we propose an Ether-aware LSM-tree based KV store by incorporating Ethereum structure semantics with underlying storage systems.

To enable the Ether-aware approach, three challenges must be addressed as follows:

- How to benefit from the coordination between Ethereum and underlying storage engine?
- Which part and how to transfer the Ethereum structure semantics to the underlying storage engine with minimum extra transformation overhead?
- How to promise the correctness and efficiency of the query process?

To address the above challenges, we first thoroughly analyze the performance of Ethereum and find that Ethereum suffers from the extremely large I/O amplification on the LSM-tree structure during the synchronization process. According to the observation, we propose a prefix-based (i.e., block number related) hashing approach to transform Ethereum data to KV items. By concatenating a prefix based on the current block number, the lexicographical order of all Ethereum data (i.e., KV items after hashing) is positively related to the time when the data is mined and written to the storage. Moreover, based on a comprehensive analysis of the average volume of data per block and the characteristics of underlying storage, we propose to reduce the overhead of prefix by grouping multiple blocks with the same prefix. Furthermore, to preserve the

978-1-6654-3219-1/21/$31.00 ©2021 IEEE
DOI 10.1109/ICCD53106.2021.00017

TABLE I
INSPECTING THE COMPOSITION OF ETHEREUM'S DATA.

Types	Category	Details	KV Composition
Block	Headers	Block header	HeaderPrefix + num + block hash ->header
	Bodies	Block body	BodyPrefix + num + block hash ->body
Metadata	Receipts	Record the transaction process	ReceiptPrefix + num + block hash ->block receipts
	Difficulties	A metric desctibed mine	HeaderPrefix + num + block hash + TDSuffix ->td
	Block number	Record block hash	HeaderPrefix + num + hashSuffix ->block hash
	Block hash	Record block number	NumberPrefix + block hash ->num
	TxLookupEntry	Assist to query transaction	TxlookupPrefix + transaction hash ->transaction block number
	Trie preimages	Security	PreimagePrefix + state hash ->preimage
State data	Trie nodes	Record account information	state hash ->state data

relationship between the Ethereum transactions and blocks, we maintain attribute-oriented memory buffers for the correctness and efficiency of the query process.

In summary, this paper aims to mitigate the system I/O resource requirements during the Ethereum synchronization process. Specifically, we propose an Ether-aware LSM-tree based KV store, named Block-LSM, to transfer the Ethereum data semantics to the underlying KV storage engine with minimal I/O overhead and correctness guarantee. We implement a fully functional prototype of Block-LSM based on Ethereum v1.92 [13]. Various experiments have been performed with both real and synthesized workloads to demonstrate the effectiveness of the proposed Block-LSM design. The evaluation results show that compared with the original Ethereum, Block-LSM significantly increases system throughput by $3\times$ and reduces the write amplification by $3.7\times$ during the data synchronization process. We have released the open-source code of Block-LSM at https://github.com/czh-rot/Block-LSM.

The main contributions of this paper are as follows:

- We analyze the performance of Ethereum by performing some preliminary experiments and find that the synchronization process of Ethereum faces extremely large I/O overhead due to the incorporation of Ethereum semantics and the underlying KV storage engine.
- We design a new storage engine, Block-LSM, to bridge the semantic gap between the Ethereum data layer and its underlying storage layer by transforming Ethereum data to KV items with a prefix-based (i.e., block number related) hashing approach.
- We further propose a block grouping prefix design and attribute-oriented memory buffers to minimize the KV transformation overhead and promise to query correctly.
- We implement a prototype of Block-LSM based on Ethereum v1.92 and evaluate its effectiveness with various real and synthesized workloads.

The rest of this paper is organized as follows. Section II and Section III introduce the background and motivation of this work. Section IV explains the design of Block-LSM in detail. The implementation and evaluation results are given in Section V. Section VI presents the conclusion.

II. BACKGROUND

In this section, we briefly introduce the background of Ethereum basics and describe its underlying storage engine.

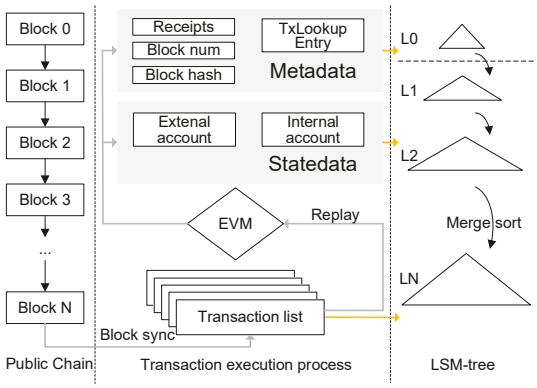


Fig. 1. Ethereum block synchronization process.

A. Ethereum Basics

Ethereum as a distributed ledger stores all committed transactions in a chain of blocks. New transactions are packed into blocks with a mining process and blocks are appended to the end of the chain. When a full node joins the Ethereum network, the full node needs to synchronize all transaction blocks from the main network and then can start to provide services, such as data transfers and verifiable queries.

To serve as a full node, after synchronizing a new block from the network, the full node first needs to replay all transactions in the blocks to generate the corresponding metadata (including Receipts, Difficulties, Block number, Block hash, TxLookupEntry, and Trie preimages) and state data (including external accounts and contract information). Metadata is used for efficient data retrieval and assists in transaction queries. State data is used to manage account information. Finally, all the data in the full node including blocks, metadata, and state data, are converted into 2-tuples (key, value) by several hash functions and written to the database, as shown in Figure 1. Table I shows the hash functions that are used to convert different data into KV items. For example, for the metadata "TxLookupEntry", when transferring it to KV items, the key is the TxlookupPrefix + $hash(transaction)$, and the value is the block number which the transaction is located in.

The Ethereum system contains two major types of lookups (i.e., transaction and account lookups).

Transaction Query. The hash value of a transaction is the unique identifier of the transaction, and users adopt the publicly available transaction hash to request the corresponding

key of the TxLookupEntry. With the TxLookupEntry, we can further get the corresponding transaction. The query process for a transaction follows three steps:

(1) Find the block number containing the target transaction based on the metadata TxLookupEntry mentioned above.

(2) Calculate the block hash value according to the block number.

(3) Read the block body from disk based on the block hash and find the required transaction.

Account Query. The account data is stored in the state data of the full node, which are queried through MPT (Merkle Patricia Trie) in Ethereum. An account lookup obtains data information by retrieving all nodes in a certain MPT path from root to leaf.

B. LSM-tree

Ethereum uses LSM-tree as its storage engine, which is one of the most popular design choices for persistent KV stores [14]–[17]. The LSM-tree based KV store consists of two components:

•*Memory component:* The memory component is used to accommodate the small and random KV pairs into sequential batched write operations. It uses two in-memory sorted skiplists (*memTable* and *immuTable memTable*) to store KV pairs with a sorted order.

•*Disk component:* The disk component is divided into multiple levels. The capacity of each level increases exponentially as the level goes deeper, and each level consists of a number of Sort String Table (SST) files.

The KV pairs from Ethereum data synchronization are firstly buffered in *memTable*. Once the *memTable* is full, it will be converted into an *immuTable*, formatted into an SST file, and written into the disk component. This procedure is called *flush*. In the disk component, When L_N (i.e., the level N of the disk component) reaches its size limit, the KV store would sort and merge its keys of all SST files that overlap with the files in the L_{N+1}. This process is called *compaction*, which ensures that all files in a particular level, except L_0, do not have key range overlap and all keys in each file is sorted.

III. Motivation

In this section, based on the performance analysis for the storage engine of Ethereum, we concluded the bottlenecks of Ethereum, which motivates to optimize the performance of Ethereum systems.

First, we performed some preliminary experiments to explore the efficiency of the LSM-tree based KV storage engine for Ethereum. For the experiments, we investigated the performance of different parts in Ethereum in terms of execution time and the number of operations. As shown in Figure 2, we separately synchronize about 1.6M, 2.3M, 3.4M, and 4.6M transaction blocks from the current Ethereum public network, and the sizes of which are about 10GB, 20GB, 40GB, and 80GB, respectively. We collected the compaction operations triggered during the synchronization process and

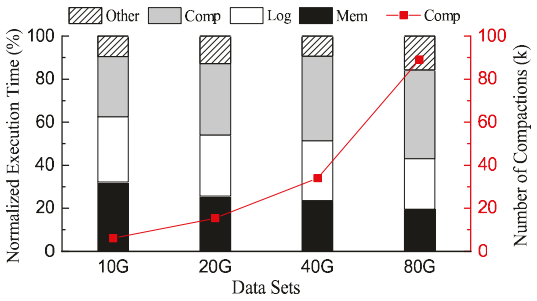


Fig. 2. Normalized execution time for different operations and the number of compaction operations with varying data sets during data synchronization.

also investigated breakdown execution times for different parts of underlying storage including the memory part (shown as "Mem"), the logging part (used for crash consistency, shown as "Log"), compaction part (shown as "Comp"), and other part (shown as "other").

As shown in the figure, we normalize the total execution time and collect the number of compaction operations during the synchronization process for different data sets. We can find that as the amount of synchronized data increases, the number of compaction operations during data synchronization increases sharply. Correspondingly, the ratio of execution time from the compaction part also increases a lot. According to the results, we can conclude that although Ethereum blockchain system writes blocks in an appending manner (i.e., sequentially write data into the underlying storage engine), the underlying storage engine during the synchronization process suffers the performance degradation from the compaction operations, which consumes a large amount of bandwidth on the disk and impedes the data write.

In LSM-tree based KV storage engine, the compaction mechanism mainly has two functions:

- Remove the invalid data (i.e., state KV items) during the compaction process.
- For query efficiency, the compaction mechanism is used to guarantee the strict ordering principle of LSM-tree.

Since Ethereum is an append-only chain, thus, there should not have update operations in the underlying storage engine. In Ethereum, once a transaction block is verified and appended on the chain, it will never be changed. With new transactions are verified and executed, the relevant account balance (i.e., state data) would be updated. However, when transferring the updated state data to KV items, the hash function will generate a new KV pair for the account update, thus, it would also not incur any KV update operation in the storage engine.

According to our analysis, the large number of compaction operations in the LSM-tree structure is caused by the key range overlaps of the inserting KV items. During the Ethereum synchronization process, although the transaction blocks are downloaded and replayed sequentially when transferring the data to KV items, the hash functions (as shown in Table I) abundant the sequence information in the keys. Thus, in the underlying LSM-tree based KV storage engine, the KV items

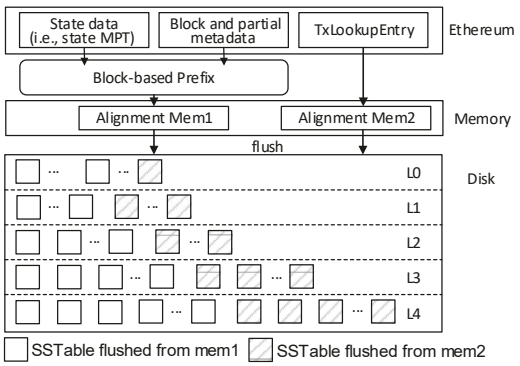


Fig. 3. Architecture overview of Block-LSM.

have to be sorted according to their hashed keys through both time and resource-consuming compaction process to meet the strict ordering principle.

Motivated by the above observations, in this paper, we propose to keep the sequence information of the original Ethereum data semantics through a prefix-based (i.e., block number related) hashing approach when transferring the Ethereum data to KV items, so as to eliminate the annoying compaction operations in the KV storage engine.

IV. Block-LSM

We propose a novel scheme called Block-LSM, to bridge the semantic gap between the Ethereum data layer and its underlying storage layer. Figure 3 shows the architecture overview of Block-LSM, mainly including three function models: the prefix-based hashing, the block group-based prefix, and the attribute-oriented memory buffers. The prefix-based hashing is used to transmit the Ethereum semantics (i.e., block number related information) to the underlying storage engine to make the insertion of KV pairs sequentially. The block group-based prefix scheme aims to minimize the transformation overhead during the semantics transferring process. To guarantee the query correctness, the KV transformation of some special metadata, such as the data used to record the mapping between transactions and blocks (i.e., TxLookupEntry), needs to be handled separately. The attribute-oriented memory buffers are used to isolate the KV items of this data from other KV items, so as to eliminate its damage to the KV insertion sequence (broken sequence would incur extra compaction operations).

A. Prefix-based Hashing

To transfer the Ethereum block order into the underlying transformed KV items, a straightforward method is to use the timestamp as a prefix for each KV entry written to the disk so that all KV entries are written sequentially. However, the timestamp for transaction blocks can not be known in advance before reading the content. Thus, the irregular characteristic of the timestamp would make the query process much difficult.

As described in Section II, during the synchronization process, when a full node downloads a transaction block from the public Ethereum network, it would replay all the transactions in the block and generate the corresponding metadata and state data. After that, all data including the transaction block, the corresponding metadata and the state data would be transferred into KV items according to the hash functions as shown in Table I. In Block-LSM design, we propose to use the transaction block number as the common prefix of the keys for the KV items related to the specified block. The construction method of the prefix is as follows:

$$Append(blocknumber, key) \rightarrow value$$

By using this transformation approach, all the KV items related to one transaction block would be aggregated into certain key ranges, and the KV items related to different transaction blocks are isolated to different key ranges according to their block number. During the synchronization process, since all the blocks are synchronized and replayed sequentially from the Ethereum public network, thus, the insertion sequence of KV items belonging to different transaction blocks in the underlying storage engine can also be reserved. As a result, the compaction operations can be reduced efficiently.

Note that when performing a transaction query, its block number is not known in advance. A user first needs to use the public transaction hash to locate the metadata TxLookupEntry. By checking the TxLookupEntry, we can further get the block number and read the block content. Thus, to guarantee the query efficiency, the transformation of the metadata TxLookupEntry cannot adopt the proposed prefix-based hashing method. In this work, we keep the original transformation method for TxLookupEntry to promise the correctness of query process in Ethereum as shown in Table I.

Moreover, during the insertion process of LSM-tree based KV store, even though the data from the same blocks of Ethereum use the same prefix, these data are possible to be inserted into two adjacent SSTable files. This is because the default size of SSTable file and memTable is fixed but the size of KV items from blockchain systems is varied from 10B to 50KB and cannot be aligned to the SSTable file size. As a result, two adjacent blocks will have overlapped key ranges and thus may cause severe compaction. To avoid this effect, we propose an **SSTable alignment** method to align the sizes of SSTable and KV items, which can flexibly perform the flush operations to ensure that all KV pairs related to the same transaction block are always stored in the same SSTable files.

The function SSTable alignment() in Algorithm 1 shows the detailed working process of our proposed prefix-based hashing approach. In Algorithm 1, Generate Prefix() and Add Prefix() show the details of how to generate and add prefix to the keys based on the transaction block number. In the beginning, all KV pairs with the prefix are sorted and inserted into memTable. When the capacity of memTable reaches a predefined threshold, Block-LSM would determine whether the current prefix is the same as the prefix of the last written KV pair, if not, it would perform the flush operation immediately (Line 12–14 of Algorithm 1). Otherwise, if the memTable still has sufficient capacity, it would write the current KV pair to the memTable (Line 17–18 of Algorithm 1).

Alogrithm 1 : The SSTable alignment Algorithm

```
1: function GENERATE PREFIX(Block)
2:     prefix := Int64ToBytes(Block.number)
3: end function
4:
5: function ADD PREFIX(K,V)
6:     if K.type is not TxLookupEntry then
7:         key := append(prefix, key)
8:     end if
9: end function
10:
11: function SSTABLE ALIGNMENT()
12:     if memTable.free <= threshold then
13:         if bytes.Compare(Last_prefix, Current_prefix) != 0) then
14:             db.flushMem()
15:         end if
16:     end if
17:     if Sizeof(Key,Value) < memTable.free then
18:         writeMem(Key,Value)
19:     else
20:         db.flushMem()
21:     end if
22: end function
```

B. Block Group-based Prefix

In the prefix-based hashing approach, by adding a block number related prefix to all the keys, Block-LSM would increase the bit length of the keys and increase the space overhead. In the LSM-tree based KV stores, since we only need to promise that the keys of the KV items of different SSTable files do not have overlap to avoid the compaction operations, we propose to separate transaction blocks into different groups and make the KV items from the same group share the same prefix (i.e., the group number) to alleviate the extra space overhead.

With the group-based prefix, although there will be a large number of KV pairs sharing the same prefix, these data will be firstly buffered in memTable. If we can properly flush them into one SSTable file, it will not incur any key range overlap between different SSTable files and compaction operations. Therefore, we only need to reasonably set the capacity and threshold of the memTable in the SSTable alignment algorithm to ensure that the KV pairs with the same prefix are flushed to one SSTable at the same time.

For the block-based prefix scheme, since each block corresponds to a prefix, the maximum value of the prefix should be equal to the number of blocks in Ethereum. So far, the entire Ethereum network has produced about 13 million blocks, and thus we need to use four bytes to represent block numbers. With block group-based prefix, since we have fewer groups, certain space can be saved. In practice, we set the capacity of a group to be 100 blocks, and we only need three bytes to represent the serial number of the group.

C. Attribute-oriented Memory Buffers

As discussed in Section IV-A, the metadata TxLookupEntry is used to look up the transaction block number during the query process. Thus, it can not adopt the block number related prefix hash policy and Block-LSM adopts the original transformation approach for TxLookupEntry. Therefore, the hash values of TxLookupEntry (i.e., the key of TxLookupEntry) is uniformly distributed. As a result, when inserting the KV items for TxLookupEntry and the KV items for other data, these keys would be mixed together in different SSTable files and make the key ranges of different SSTable files have overlap. Finally, Block-LSM would trigger many compaction operations to reorganize these SSTable files (the evaluation results shown in Figure 5 have demonstrated this scenario).

To avoid the disturbance, we propose to maintain different attribute-oriented memory buffers to separately handle special metadata KV items and other KV items (generated with the prefix-based hashing). With the attribute-oriented memory buffers, the KV items for TxLookupEntry and other data are maintained in different memTables and are flushed to their corresponding SSTable files as shown in Figure 3. Thus, in the underlying storage engine, the SSTable files for TxLookupEntry and other data have no key range overlap. Due to the hash randomness, the different SSTable files for TxLookupEntry must have key range overlap. However, the size of TxLookupEntry is much smaller than other data. So, Block-LSM will face a few compaction operations to compact the KV items for TxLookupEntry in different SSTale files and finally those compactions have slight influence on the overall synchronization performance.

V. EVALUATION

A. Experiment Setup

Environment. We develop a full functional prototype of Block-LSM based on Ethereum v1.92, which adopts LevelDB [18] as its underlying KV storage engine. The prototype includes 6300 lines of codes modification and is publicly accessible at https://github.com/czh-rot/Block-LSM. We deploy Block-LSM with a testing machine equipped with Inter(R) i7-10875 CPU @ 2.30 GHz with 8 cores, 16 GB Memory, and 1TB SN-750 NVMe SSD. The operating system is Ubuntu 18.04. For the KV storage engine settings, if not otherwise specified, the SST file is set to 2MB and the BlockCache is set to 200MB. We define the size of the group as 100 blocks to run all the tests.

Workloads. To evaluate the Ethereum synchronization process, we first use Geth to synchronize transaction blocks from the public main chain to local disk and then replay these transactions to the proposed Block-LSM storage engine. By doing this, we can exclude the network influence. For generality, we adopt four groups of real workloads with different numbers of blocks: 1.6M, 2.3M, 3.4M, and 4.6M from the public Ethereum network, and the sizes of which are about 10GB, 20GB, 40GB and 80GB, respectively.

For the read workloads, since tracing the read requests of the Ethereum main chain is a time lasting work (may need months

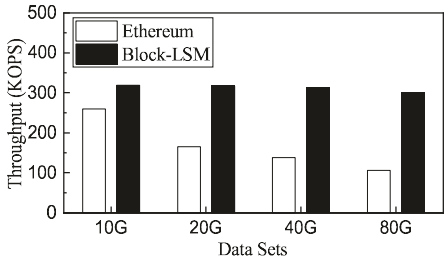


Fig. 4. The throughput performance during data synchronization.

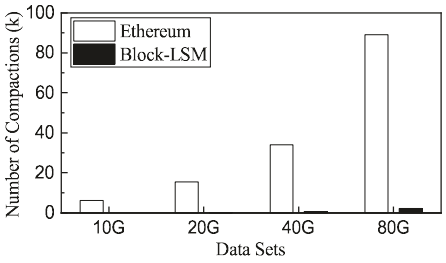


Fig. 5. The number of compaction operations during data synchronization.

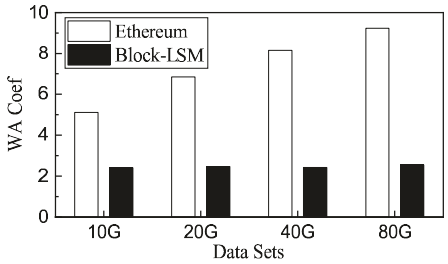


Fig. 6. The WA coef during data synchronization.

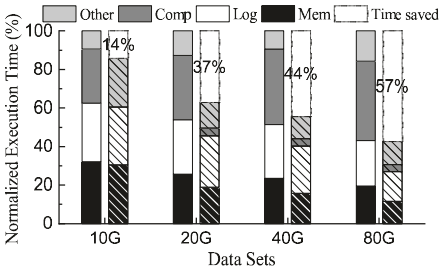


Fig. 7. Comparison of normalized execution time. The bar filled with lines represents Block-LSM, and another represents Ethereum.

or years) and there is no open-sourced testing data sets, thus, in this work, we syntheses multiple read workloads following three different distributions [19], [20] (i.e., Skewed Latest Zipfian, Scrambled Zipfian, Normal Random) depending on the rules of performing transactions and the characteristics of accounts. The read workloads are generated based on the synchronized 4.6M Ethereum blocks (the data size is 80GB) with different distributions, which contain nearly 100 million transactions and 10 million accounts in total, respectively.

B. Data Synchronization Performance

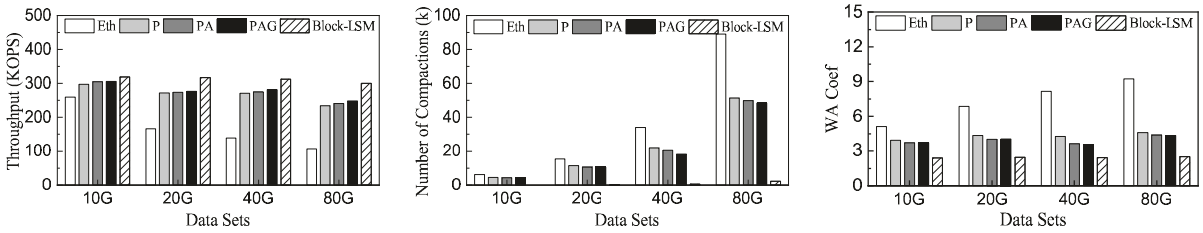
For the data synchronization process, we evaluate the performance of the original Ethereum and Block-LSM in terms of throughput, the number of compaction, and write amplification. The results are shown in Figure 4– 7.

•Throughput: Figure 4 shows the system throughput (Thousand operations per second, a.k.a, KOPS, note that an operation here means a KV insertion) of different Ethereum data sizes during the synchronization process. We can see that Block-LSM outperforms Ethereum across the board, and with the data sizes varying from 10GB to 80GB, the throughput improvement of Block-LSM over Ethereum ranges from 23.01% to 182.75%. More importantly, unlike Ethereum, whose data synchronization time slows down significantly as the data set gets larger, the throughput of Block-LSM stably keeps at a much higher level. The reason for the throughput improvement of Block-LSM is that by transferring the Ethereum block order to the underlying KV storage engine, the number of time-consuming compaction operations (which involves multiple SSTable copies, KV item sorting, and SSTable written back) is significantly reduced.

•Compaction: Figure 5 shows the number of compaction operations during the data synchronization process of Block-LSM and the original Ethereum. From the figure, we can see that Block-LSM efficiently eliminates most of the compaction operations in the storage engine. As the store sizes increase, the compaction operations slightly increase but at a much lower level. In contrast, the original Ethereum design suffers from a huge number of compaction operations. When the data size increases, the number of compaction operations increases significantly. With the data size of 80GB, the original Ethereum has up to 9×10^4 more compaction operations than the proposed Block-LSM. Note that Block-LSM does not completely remove all compaction operations, and the reason for this is that to promise the query correctness and efficiency, the data structure TxLookupEntry of Ethereum is still transferred to KV items by hashing. Thus, when flushing those KV items to disk, they are not ordered and may incur some compaction operations.

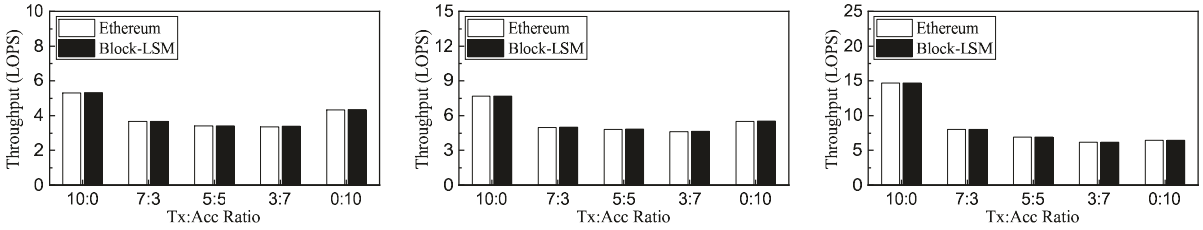
•Write Amplification: Figure 6 shows the write amplification coefficient (WA Coef) for Ethereum and Block-LSM. As shown in the figure, as the size of the data set increases, the WA Coef of Ethereum ranges from 5.12 to 9.24, which presents an increasing trend. However, benefiting from the elimination of compaction operations, the WA Coef of Block-LSM is about 2 and is relatively stable for the different data sets. The extra write amplification for Block-LSM is mainly caused by the Write-Ahead-Logging (WAL) mechanism of the underlying KV storage engine (each KV pair must first be written to a log file before being inserted into the memTable).

Figure 7 shows the distribution of the time spent in different components during each run of the test between Ethereum and



(a) Write Throughput. (b) Number of Compactions (c) WA Coef.

Fig. 8. Breakdown analysis of data synchronization.



(a) Normal Random. (b) Scrambled Zipfian. (c) Skewed Zipfian.

Fig. 9. The system read performance.

Block-LSM. As shown in figure, compared to the Ethereum, compaction operations in Block-LSM account for only a small part of the total time, for the reasons explained in Figure 5. In addition, since the inserted KV pairs are initially ordered, which avoids the sort operation of the memTable, the execution time of memTable is also significantly reduced. In summary, compaction operations in Block-LSM are no longer a bottleneck that limits system performance and introduces write overhead.

C. Breakdown Analysis

In this section, to investigate influence of each proposed scheme on write performance, we superpose each scheme one by one until it becomes Block-LSM as described below:

- **Ethereum (Eth):** represents the original Ethereum.
- **Prefix (P):** stands for our design that only adopts the block number related prefix hashing approach.
- **Prefix+Align (PA):** stands for our design with both block number related prefix hashing and SSTable alignment.
- **Prefix+Align+Group (PAG):** stands for our design with block group related prefix hashing and the SSTable alignment scheme.
- **Block-LSM:** is the design that includes all the proposed policies.

Figure 8 shows the throughput of these implementations with varied Ethereum data sizes. As shown in the figure, Block-LSM achieves the highest throughput improvement over the original Ethereum among all these implementations, and Prefix has the lowest of that. By integrating attributed-oriented memory mechanism with block-based prefix, the write throughput also increases for all data sets.

By adopting the block number related prefix hashing, most of the transferred KV items are inserted into the KV storage engine orderly, and the compaction number has been reduced a lot (about 35%–74%) compared with the original Ethereum. However, there are still many compaction operations due to the alignment issue and the scatter hashed $TxLookupEntry$ KV items. By incorporating the alignment design, the compaction number is further reduced, but the percentage is very small (i.e., 3%–6% compared with Prefix). The block group-based prefix policy is used to mitigate the space overhead introduced by the extra key prefixes and it would not contribute to the compaction reduction. Through the evaluation results, we can observe that it also does not introduce any side effect to the compaction process. Note that compared with the $Prefix + Align$ implementation, Block-LSM further reduces the compaction operations significantly. This indicates that the scatter hashed $TxLookupEntry$ KV items severely influence the stability of the underlying KV storage LSM structure and our attributed-oriented memory buffer design can fundamentally solve this problem.

D. Read Performance

We investigate the read performance of Ethereum and Block-LSM under various workloads. The metric for the read performance is Lookup Operations Second (LOPS). Note that an operation here means an account or a transaction search rather than a KV pair query.

Figure 9 shows the read performance results for Ethereum and Block-LSM under the workloads with three distributions. As shown in figure, despite our block-based prefix scheme increasing the length of the KV pairs queried, there is no significant difference in performance between Ethereum and Block-LSM across all workloads, with the performance gap consistently within the 3%. The reason is as follows.

Ethereum's read workload has a characteristic: A request in Ethereum tends to query multiple KV entries with the same

TABLE II
CPU USAGE OF DIFFERENT WRITE WORKLOADS.

Data Sets	10G	20G	40G	80G
Ethereum	11.43%	11.15%	11.13%	11.82%
Block-LSM	9.18%	9.21%	8.90%	9.86%

type in succession and these entries are generated by a block. Therefore, data generated from the same block is clustered in adjacent locations on the disk due to the prefix, which has a positive effect on querying. The two factors balance each other, so the read performance of Block-LSM does not show significant degradation.

E. Overhead

In this section, we analyze the overhead of Block-LSM from two aspects: computation overhead and space overhead.

•Computation Overhead. In Block-LSM, when performing the prefix-based hashing and the SSTable alignment, extra computation resources would be needed. Therefore, to compare the computation resource consumption, we record the system CPU utilization during the data synchronization process of the original Ethereum and Block-LSM with various data sizes. The results are summarized in Table II.

As shown in the table, though Block-LSM introduces extra computation overhead, its average CPU utilization is slightly lower than the original Ethereum design. This is because Block-LSM significantly reduces the system compaction operations (which is both I/O and computation hungry) and thus the overall CPU utilization with Block-LSM is slightly decreased.

•Space Overhead. The block number related prefix hashing approach increases the length of the keys of transferred KV pairs, which makes Block-LSM require more disk space. To evaluate the space overhead, we collected the required storage space of the original Ethereum and Block-LSM. With the Ethereum data sizes vary from 10GB to 80GB, the space overhead of Block-LSM with block-based prefix ranges from 2.43% to 2.76%, and space overhead of the block group-based prefix ranges from 1.82% to 2.07%. Considering the performance improvement, especially for the data synchronization, we claim the space overhead is acceptable for Block-LSM.

VI. CONCLUSION

In this paper, we propose Block-LSM, an Ethereum-aware LSM-tree based KV store to optimize the system I/O resource requirements. Block-LSM achieves its design goal by introducing the block number related prefix hashing to make the insertion of KV pairs sequentially, a block group related prefix hashing to minimize the transformation overhead, an attribute-oriented memory mechanism to avoid the disturbance between various KV items. We implement a fully functional prototype of Block-LSM based on Ethereum v1.92 and various experiments have been performed. Our experimental results show that Block-LSM achieves significant improvement. The open-source code of Block-LSM is available at https://github.com/czh-rot/Block-LSM.

ACKNOWLEDGEMENT

This work was sopported by NSFC-Shandong Joint Fund (No.U1806203), direct Grant for Research, the Chinese University of Hong Kong (No.4055151), major scientific and technological innovation project in Shandong Province (No.2019JZZY010449), the National Science Foundation of Young Scientists of China under Grant (No.61902218), the National Natural Science Foundation of China under Grant (No.92064008).

REFERENCES

[1] Wood, Gavin, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, pp. 1-32, 2014.

[2] Shynu, P. G. , et al., "Blockchain-based Secure Healthcare Application for Diabetic-Cardio Disease Prediction in Fog Computing," *IEEE Access*, vol. 9, pp. 45706-45720, 2021.

[3] Huckle, S., et al., "Internet of Things, Blockchain and Shared Economy Applications," *The 7th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2016)*, vol. 98, pp. 461-466, 2016.

[4] Ali Dorri, et al., "Blockchain for IoT security and privacy: The case study of a smart home," *2017 IEEE International Conference on Pervasive Computing and Communications Workshops, (PerCom Workshops 2017)*, pp. 618-623, 2017.

[5] Usama Farooq, et al., "Blockchain-Based Software Process Improvement (BBSPI): An Approach for SMEs to Perform Process Improvement," *IEEE Access*, vol. 9, pp. 10426–10442, 2021.

[6] Haya R. Hasan and Khaled Salah, "Proof of Delivery of Digital Assets Using Blockchain and Smart Contracts," *IEEE Access*, vol.6, pp. 65439–65448, 2018.

[7] Etherscan, https://etherscan.io/.

[8] JD Bruce, "The mini-blockchain scheme," *White paper*, 2014.

[9] Junying Gao, Bo Li, and Zhihuai Li, "Blockchain Storage Analysis and Optimization of Bitcoin Miner Node," *Communications, Signal Processing, and Systems - Proceedings of the 2018 CSPS* Vol. 517, pp. 922–932, 2018.

[10] Matt Corallo, "Bip152: Compact block relay," See https://github.com/bitcoin/bips/blob/master/bip-0152, mediawiki, 2016.

[11] Donghui Ding, Xin Jiang et al., "Txilm: Lossy Block Compression with Salted Short Hashing," *IACR Cryptol. ePrint Arch*, vol. 2019, pp. 649, 2019.

[12] O'Neil, P., Cheng, E., Gawlick, D. et al., "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[13] Ethereum, WG, "A secure decentralised generalised transaction ledger", vol. 151, pp: 1-32, 2014.

[14] Chang, Fay , et al., "Bigtable: A Distributed Storage System for Structured Data," *Acm Transactions on Computer Systems*, vol. 26, no. 2, pp. 1-26, 2008.

[15] Lu, Lanyue, et al., "Wisckey: Separating keys from values in ssd-conscious storage." *14th USENIX Conference on File and Storage Technologies (FAST 2016)*, pp. 133-148, 2016.

[16] Vora, Mehul Nalin., "Hadoop-HBase for large-scale data." *Proceedings of 2011 International Conference on Computer Science and Network Technology*, vol. 1, pp. 601-605, 2011.

[17] Pandian Raju, et al., "PebblesDB: Building Key-Value Stores by using Fragmented Log-Structureed Merge Trees," *ACM Proceedings of the 26th Symposium on Operating Systems Principle*, pp. 497-514, 2017.

[18] 2017, GolevelDB, https://github.com/syndtr/goleveldb.

[19] Berk Atikoglu, et al., "Workload analysis of a large-scale key-value store," *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2012)*, pp. 53-64, 2012.

[20] Zhichao Cao, et al., "Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook," *In 18th USENIX Conference on File and Storage Technologies (FAST 2020)*, pp. 209-223, 2020.