Received August 2, 2021, accepted August 14, 2021, date of publication August 17, 2021, date of current version August 24, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3105428

Optimizing Key-Value Stores for Flash-Based SSDs via Key Reshaping

SUNGGON KIM[1] AND YONGSEOK SON[2]
[1]Department of Computer Science and Engineering, Seoul National University, Seoul 08826, South Korea
[2]Department of Computer Science and Engineering, Chung-Ang University, Seoul 06974, South Korea

Corresponding author: Yongseok Son (sysganda@cau.ac.kr)

This work was supported by the National Research Foundation of Korea grant (No.2021R1C1C1010861) and was supported through BK21 FOUR (Fostering Outstanding Universities for Research) Program funded by Ministry of Education of Korea (No.I20SS7609062)

ABSTRACT Key-Value store (KV store) is becoming widely popular in both academia and industry due to its fast performance and simplicity in data management. To improve the performance of KV stores, recent Serial Advanced Technology Attachment (SATA) and Non-Volatile Memory express (NVMe) Solid-State Drives (SSDs) have been widely adopted. In contrast to the existing Hard-Disk Drives (HDDs), SSDs have unique characteristics which must be carefully considered to exploit the full performance. For example, due to the erase before write constraint, the access pattern of workloads impacts the performance and endurance of SSDs. Thus, the performance of SSD with the sequential workload is higher than that with the random workload. In this paper, we propose a key reshaping technique to improve the performance of KV stores with high performance storage devices. By reshaping keys, our scheme allows KV stores to process the random insert requests into sequential insert requests, improving request processing and Input/Output (I/O) performance. Our experimental results show that the proposed scheme can improve the performance of KV store by up to 106% and 281% compared with the existing scheme, in the case of SATA and NVMe SSDs, respectively.

INDEX TERMS Flash-based SSDs, key-value store, non-volatile memory, database.

I. INTRODUCTION

As recent applications produce a large amount of data, it is crucial to efficiently store and access the data. To do this, relational databases [1], [2] are widely used to manage the data. However, as relational databases support rich relationships, it can be difficult to manage large amounts of data with efficiency and high performance. To improve efficiency, KV stores are widely used in both industry and academia due to simpler data management and higher performance compared with relational databases.

To improve the performance of KV stores further, emerging high performance storage devices such as SATA and NVMe SSDs are widely adapted as storage devices. Compared with the existing HDDs, SSDs offer low latency and high bandwidth. However, SSDs have unique characteristics that should be carefully considered. For example, in SSDs, the performance of random write workload is far less than the performance of sequential write workloads [3], [4]. This is due to the erase-before-write constraint of NAND flash which constitutes an SSD. Thus, to fully exploit the performance of SSD, it is critical to consider the access pattern of workload.

There have been many studies that improve the performance of KV databases with SSDs. Wisckey [5] is SSD optimized KV store that separates keys and values and manages keys in the LSM tree. By only managing keys in the LSM tree, it reduces the size of LSM tree and improves the performance of general tree operation. KVSSD and NVMKV [6], [7] propose new architectures for SSD that integrate KV store with SSD. By redesigning the flash translation layer (FTL) in SSD, these schemes provide tight integration between KV store and emerging high performance SSDs. Our study is in line with these studies in terms of optimizing KV stores for emerging high performance storage devices. In contrast to these studies, our study focuses on reshaping key to improve the performance of KV stores.

In this paper, we propose a key reshaping scheme for KV stores with emerging high performance storage devices.

The associate editor coordinating the review of this manuscript and approving it for publication was Cristian Zambelli.

Our main scheme reshapes random insert requests issued by clients to sequential insert requests by allocating the sequential reshaped key to the incoming requests. This allows KV stores to insert the key value pairs efficiently and reduce the number of memory operations. In addition, KV stores with our proposed scheme perform sequential I/O operations even when clients are issuing random insert operations, improving the data locality inside SSD. We apply our scheme to WiredTiger which is a widely used KV store and a default storage engine of MongoDB [8]. For evaluation, we use SATA and NVMe SSDs. The evaluation results show that our scheme can improve the performance by up to 106% and 281% compared with the existing scheme using SATA and NVMe SSDs, respectively.

Our contributions are as follows:

- We propose a key reshaping technique that transforms random insert workloads into sequential insert workloads.
- We propose a remapping table to support correct read, update and delete requests.
- We evaluate our reshaping scheme using a widely used KV store with SATA and NVMe SSDs.

The rest of the article is as follows: Section II describes the background. Section III shows the overall design of our proposed scheme. Section IV presents the experimental result using our proposed scheme. Section V discusses the related work and Section VI.

II. BACKGROUND AND MOTIVATION

A. KEY-VALUE STORE

KV stores are designed to efficiently handle increased data volume produced by recent applications and users. Previous studies stated that the traditional relational database model is unnecessary in many cases due to the simple request pattern of applications [9], [10]. To reduce the overhead of supporting complex relations and data types that are not used by applications, KV stores only support a single data type called a key value pair. A key value pair is constituted by a key which is an index to search the data and a value which is the data associated with the key. This relationship between a key and value is the only relationship supported by KV stores and key-value pairs do not have any relation to each other. This allows KV stores to provide higher performance and lower data management overhead compared with the traditional RDBs.

When managing key value pairs in KV store, there are two widely used data structures which are B+ tree and LSM tree [11]. B+ tree is a widely used data structure which is a variant of self-balancing trees. By balancing the tree when a new insert, update, and delete request comes, B+ tree can have a high balancing overhead but can have better read performance compared with LSM tree as the tree is balanced. In contrast, LSM tree buffers the data in memory and flushes the data to the disk periodically in a sequential manner. The flushed data is managed using Sorted String Tables (SSTables) which contain sequentially written key value pairs. As the data is buffered and written sequentially, LSM tree can have better insert performance compared with B+ tree. However, since multiple SSTables exist in a disk, multiple SSTables can be accessed to find data which can result in low read performance [12]. Due to these characteristics, B+ trees are often used in various areas including file systems and KV stores that require high read performance [13].

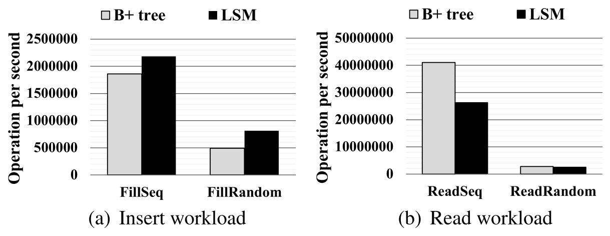


FIGURE 1. Performance of WiredTiger with B+ tree and LSM tree.

Figure 1 shows the motivational evaluation results that measure the operation per second using B+ tree and LSM tree as the data structure. For evaluation, we used a machine with an Intel i7 CPU with 4 cores and Samsung 840 Pro SSD. We used WiredTiger as KV store as it supports both B+ tree and LSM tree as the data structure. For a benchmark, we used dbbench benchmark from levelDB and configured the benchmark to insert 100 million key-value pairs with a value size of 100 bytes [14]. As shown in the figure, wiredtiger with B+ tree shows better read performance while wiredtiger with LSM tree shows better insert performance. In addition, the effect of the access pattern was greater when B+ tree was used. This is because B+ tree is a self-balancing tree and the access pattern of the workload affects the node access pattern within the tree. For example, when sequential insert workload is used, a certain subtree containing the target key is accessed which can be cached on memory. In contrast, random insert workloads can incur random node accesses which can result in multiple subtree accesses that can incur high memory footprint and disk cache access. This can result in high balancing overhead with lower performance compared with sequential insert workloads [15], [16]. This evaluation shows that the impact of access pattern is greater when KV store with B+ tree is used compared with KV store with LSM tree.

In the existing KV store as shown in Figure 2, random key-value pairs with values V_0, V_i, and V_n are being inserted by client 1 through n. To insert pairs in random key order, random index nodes and leaf nodes need to be accessed, resulting in random access patterns. As shown in the figure, to find the leaf node to insert the key-value pair with V_0, the KV store first searches head and finds the index node with a key close to K_0. After finding the leaf node to insert the pair with V_0, the pair is inserted to the leaf node and the pointer to the leaf node (K_0 and P_0) is created in both the head node and the index node. Thus, when inserting key value pairs with random keys, random index nodes and leaf nodes need to be accessed, resulting in random node access patterns.

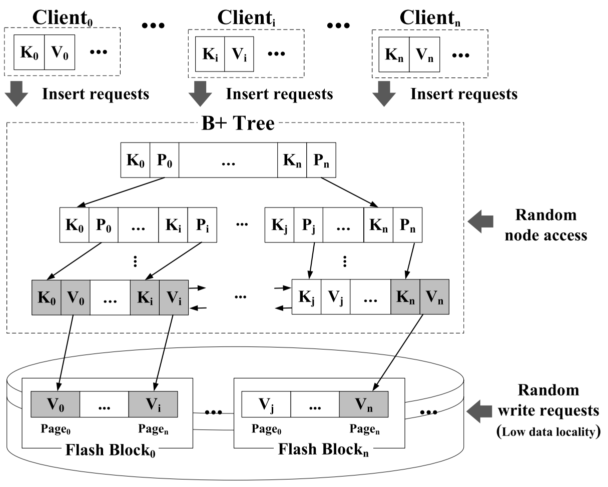


FIGURE 2. Overall architecture of existing KV store. (K: key, V: value, P: pointer).

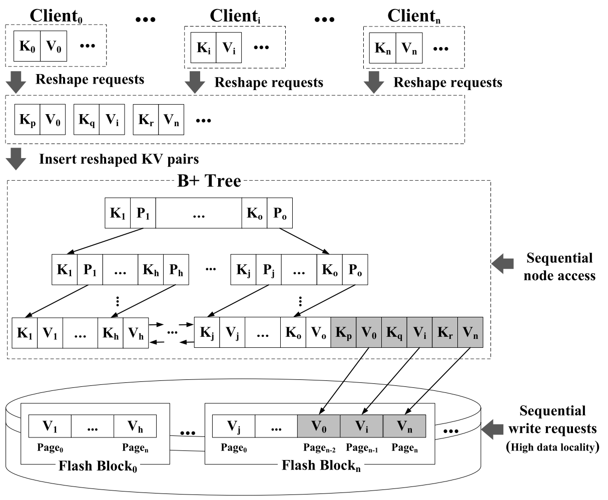


FIGURE 3. Overall architecture of the proposed scheme.

In addition to the random node access, random insert workload can result in random write requests. As shown in the figure, newly inserted values (V_0, V_i, and V_n) are stored in pages that are located in different flash blocks. As values within the block must be ordered, the blocks must be updated. As they are in different flash blocks, when a read-modify-write operation is needed, multiple flash blocks must be updated as the updates are scattered to multiple blocks. The KV store is issuing random I/O requests which increase the number of I/O requests, reducing the overall SSD performance. Also, when the data pages need to be reclaimed, two GC operations should be performed for two flash blocks instead of one block, reducing GC operation efficiency. Thus, random insert workloads can create random node access and random write requests, degrading the performance of KV store compared with the sequential insert workloads.

B. SOLID STATE DRIVES

SSDs are becoming widely adopted in both industry and academia due to their fast performance and low latency compared with traditional HDDs. Compared with HDDs, SSDs are often constituted by NAND flash blocks and controllers which do not include any moving parts. This allows SSDs to access multiple flash blocks in parallel, providing faster performance compared with HDDs.

However, as SSDs use flash blocks to store the data, the characteristics of flash blocks must be carefully considered to exploit the performance of SSDs. For example, once data is written in a flash block, the data cannot be updated in an identical block due to the characteristics of NAND flash. To update saved data, the original block must be first read, and the modified data must be written to another block. Then, the original block with old and unmodified data must be cleared for future usage which is called garbage collection (GC) operation. This update procedure is called the out-of-place update procedure.

Since the entire block must be updated regardless of the size of updated data, it is more efficient to update an entire block rather than multiple blocks. Even if the amount of data that must be updated is identical, the performance can be different based on the data locality and the access pattern of workloads [3], [17]. Thus, it is important to consider the data locality and the access pattern of workloads to fully exploit the performance of SSDs.

III. DESIGN AND IMPLEMENTATION

In this section, we present the design and implementation of our proposed key reshaping scheme for KV stores with SSDs.

A. BENEFITS OF THE PROPOSED SCHEME

Figure 3 shows the overall architecture of the proposed key reshaping scheme. As shown in the figure, our proposed scheme reshapes the incoming keys into sequential keys, creating sequential request patterns. Through this, our scheme can transform the access pattern within the B+ tree and improve the data locality for the underlying SSD.

1) SEQUENTIAL ACCESS PATTERN IN B+ TREE

In our proposed scheme, the random insert requests from the clients are transformed into sequential requests through key reshaping operations. After the reshaping operation, the key of the new request is reshaped to have a higher key than the existing keys. Thus, when inserting the reshaped key, the leaf node of the key is placed at the rightmost of the B+ tree which reduces random access patterns to find the location of a new request. For example, in the existing KV store as shown in Figure 2, to insert K_0, K_i, and K_n, index nodes (K_0-P_0, K_i-P_i, and K_n-P_n) must be accessed to find the location of leaf nodes (K_0-V_0, K_i-V_i, and K_n-V_n). However, in the proposed KV store as shown in Figure 3, only a single index node (K_o-P_o) is needed to be accessed. This is because the key of the new request (K_p, K_q, and K_r) is always higher than the highest keys ($K_1 \ldots K_o$) that already exists in B+ tree. Thus, the proposed scheme can improve the overall performance of KV store by creating sequential access pattern in B+ tree.

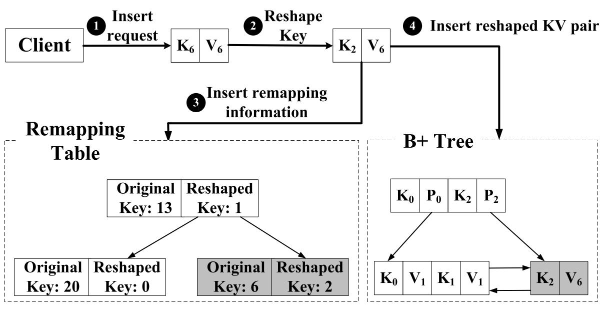

Client
❶ Insert request
K6 V6
❷ Reshape Key
K2 V6
❹ Insert reshaped KV pair
❸ Insert remapping information
Remapping Table
Original Key: 13 | Reshaped Key: 1
Original Key: 20 | Reshaped Key: 0
Original Key: 6 | Reshaped Key: 2
B+ Tree
K0 P0 K2 P2
K0 V1 K1 V1
K2 V6

FIGURE 4. Insert Operation.

2) IMPROVING DATA LOCALITY IN SSDs

Transforming random requests into sequential requests is also beneficial to the underlying storage devices, especially for flash-based SSDs. As mentioned in Section II, SSDs can benefit from high data locality due to the out-of-place characteristics of NAND flash blocks.

In contrast, in the proposed KV store as shown in Figure 3, the newly inserted values (V_0, V_i, and V_n) can be issued by a single large sequential write request. This reduces the number of I/O requests and improves the I/O performance. In addition, as the data pages are stored in a single flash block, if the pages need to be reclaimed, only a single GC operation can reclaim all the data pages, improving the efficiency of the GC operation. Thus, our scheme can improve the overall performance of KV store by creating a sequential access pattern in B+ tree and improving data locality in the underlying SSD.

B. DATABASE OPERATION

We will explain the overall procedure of insert, search, update, and delete operations of the proposed scheme.

1) INSERT OPERATION

When a client issues an insert request, the existing KV store receives a key-value pair. The value of the key is stored in the storage and the pair of key and pointer to the stored value is inserted into B+ tree. To do this, the existing KV store stores the value data, allocates a node with key and pointer to the value data, finds the location for the pair, and inserts the new KV pair. Since the B+ tree in KV store is a self-balancing tree when insert requests of random keys are issued by the client, KV store experiences performance degradation.

To improve the performance of random insert operation, our proposed scheme transforms random insert requests into sequential insert ones by reshaping the keys of requests. To do this, we first find the last key generated by the last insert operation. Then, we increment the found last key and replace the newly inserted key with the incremented key. For future requests (i.e., search, update, and delete), we store the remapping information of original and reshaped keys to remap from the original key issued by the client again to the reshaped key. To store the information of the original and reshaped key, we created a remapping table which is a red-black tree. After the information is inserted into the remapping table, we perform the insert operation with the reshaped key to B+ tree in KV store. Since the reshaped key always increments, the KV store can perform the sequential inserts, improving the insert operation performance.

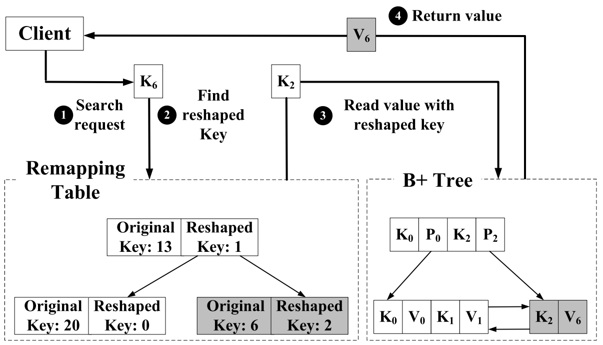

Client
V6
❹ Return value
K6
❶ Search request
❷ Find reshaped Key
K2
❸ Read value with reshaped key
Remapping Table
Original Key: 13 | Reshaped Key: 1
Original Key: 20 | Reshaped Key: 0
Original Key: 6 | Reshaped Key: 2
B+ Tree
K0 P0 K2 P2
K0 V0 K1 V1
K2 V6

FIGURE 5. Search Operation.

Figure 4 shows an example of insert operation in the proposed scheme. As shown in the figure, a client is issuing an insert request with the key of K_6 and the value of V_6 (❶). To reshape the key, we first find the latest key which is K_1 according to the current status of the remapping table. Thus, the original key of K_6 is reshaped to K_2 by incrementing K_1 (❷). Then, we insert the reshaped key information of the original key (K_6) and reshaped key (K_2) into the remapping table (❸). Finally, the reshaped key of K_2 and its value of V_6 is inserted into the existing B+ tree by updating P_2 and connecting K_1 and K_2, which will create a sequential access pattern.

2) SEARCH OPERATION

When a client issues a search request, the KV store receives a key and needs to return the value paired with the key. To do this, the KV store searches the key from the B+ tree, finds the value pointer that has the location of the value data, reads the data from the storage device and returns the value to the client.

In our proposed scheme, since our scheme inserts the reshaped key value pair into the B+ tree in KV store, the original key cannot be used to access the requested value. To access the requested value, the reshaped key must be found first before performing the read operation to the B+ tree. To do this, we first find the reshaped key associated with the original requested key. With the reshaped key, the read request is sent to the B+ tree. Then, we perform the read operation with the original value pointer that is paired with the reshaped key. Finally, the original value data is returned to the client, guaranteeing correct read operation.

Figure 5 shows an example of the search operation in the proposed scheme. As shown in the figure, a client is issuing a read request with the key of K_6 and the KV store needs to return the value that was paired with it (❷). However, since the K_6 is reshaped during the insert operation, the reshaped key index must be found first. Thus, we first find the reshaped key index by searching the remapping table with the original key (K_6) and find the reshaped key (K_2) (❸). Then, with the reshaped key of K_2, we perform the read request in B+

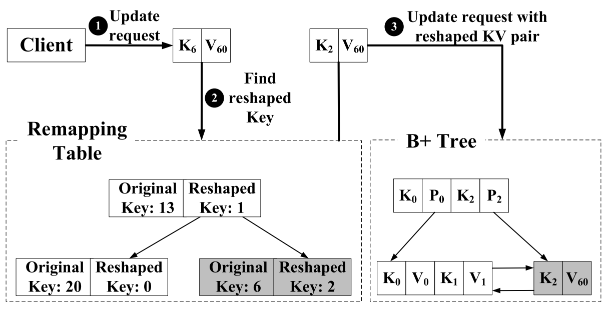

Client
❶ Update request
K6 V60
❷ Find reshaped Key
K2 V60
❸ Update request with reshaped KV pair
Remapping Table
B+ Tree
Original Key: 13 | Reshaped Key: 1
Original Key: 20 | Reshaped Key: 0
Original Key: 6 | Reshaped Key: 2
K0 P0 K2 P2
K0 V0 K1 V1
K2 V60

FIGURE 6. Update Operation.

tree (❹). Finally, since the V_6 is paired with the reshaped key of K_2, the KV store reads the V_6 and returns the value data to the client (❺). Thus, by referencing the remapping table first and finding a reshaped key, our proposed scheme can ensure the correct read operation of the reshaped keys.

3) UPDATE OPERATION

When the client issues an update request, the KV store receives a key value pair and the value that was originally paired with the key must be updated to the new value. To do this, similar to the read operation, the KV store searches the key from the B+ tree, finds the value pointer that has the location of the value data, and updates the data. Thus, the update operation is similar to the read operation but after the requested key is found in B+ tree, the value is updated rather than returned to the client.

Similar to the read operation, in the proposed scheme, the key is reshaped to have a sequential access pattern before being inserted into the B+ tree. Thus, the reshaped key must be found first to update the value to the new value. To do this, we first find the reshaped key from the remapping table. With the found reshaped key, we send the update request with the reshaped key and the new value from the client. Then, the original value data which is paired with the reshaped key is updated with a new value from the client.

Figure 6 shows an example of the update operation in the proposed scheme. As shown in the figure, a client is issuing an update request with a key of K_6 and a new value of V_{60} (❷). The KV store needs to update the value that was originally paired with K_6 to V_{60}. To find the original value, we first find the reshaped key of the original key by finding the remapping table (❸). Since the reshaped key of the original key (K_6) is K_2, we send the update request to B+ tree with a key of K_2 and the new value V_{60} (❹). Finally, the new value is written to the storage and the value pointer of K_2 gets updated to V_{60} in the B+ tree. Thus, our proposed scheme can accurately update the value of the existing key by finding a reshaped key through the remapping table.

4) DELETE OPERATION

When a client issues a delete request, the KV store receives a key and both value data associated key and the KV pair in B+ tree must be deleted. To do this, the KV store, similar to the read and update operation, KV store searches the key

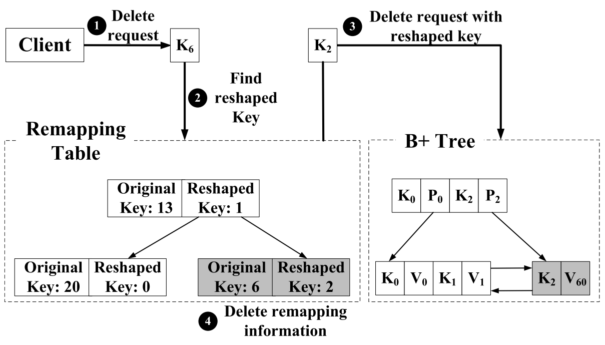

Client
❶ Delete request
K6
❷ Find reshaped Key
K2
❸ Delete request with reshaped key
Remapping Table
B+ Tree
Original Key: 13 | Reshaped Key: 1
Original Key: 20 | Reshaped Key: 0
Original Key: 6 | Reshaped Key: 2
K0 P0 K2 P2
K0 V0 K1 V1
K2 V60
❹ Delete remapping information

FIGURE 7. Delete Operation.

from the B+ tree, finds the value pointer that has the location of the value data, and removes the data and KV pair from the disk and B+ tree.

Similar to the read and update operation, in the proposed scheme, the reshaped key must be found first to locate the value data associated with the original key. To do this, we find the reshaped key associated with the original key from the delete request. Then, a delete request with the reshaped key is sent to B+ tree, deleting both the value data and KV node with reshaped key and value pointer. Finally, the remapping information with the original and reshaped key is deleted from the remapping table.

Figure 7 shows an example of delete operation in the proposed scheme. As shown in the figure, a client is issuing a delete request with K_6 (❷). To delete the value paired with K_6, we first find the reshaped key from the remapping table (❸). Then, with the found reshaped key (K_2), we send a delete request with K_2 to B+ tree (❸). After the value data of V_{60} and KV pair with K_2 is deleted from B+ tree, we delete the remapping information by deleting a node with the original key of K_6 and reshaped key of K_2. Thus, our scheme can delete the existing KV pair and remapping information by searching reshaped key first and deleting KV pair associated with the reshaped key.

C. CLEANING OPERATION

In the existing log-structured systems [18]–[20], the random write requests are transformed into sequential write requests by writing data in an append only manner. Thus, when a data block that is already written is updated, the data block must be invalidated. To remove the invalid data blocks, a cleaning operation is needed. In our proposed scheme, we perform the cleaning operation whenever the update operation is performed. Thus, we invalidate and remove the value associated with removed keys immediately after performing the update operation. By doing so, our scheme supports this cleaning operation on the fly in contrast to existing log-structured systems.

When reshaping the key value, we use a global atomic int value and increase the value by 1 using atomic instruction to ensure the correct key value with multiple threads. However, since the size of the key is limited, the global int value can reach its maximum size as KV store operates. To overcome this issue, we perform a key cleaning operation when the global int value cannot be increased due to overflow. To do this, we create a new database with a new B+ tree. Then, we read the valid key from the old database and insert the valid key value pair into a new B+ tree. By doing so, the invalidated keys can be reclaimed and can be used for future requests.

D. CRASH CONSISTENCY

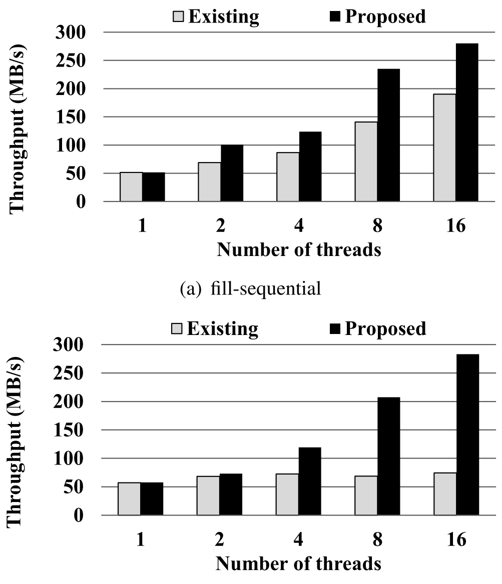
In the existing KV stores, crash consistency is provided using Write-ahead Logging (WAL). When a client starts a transaction, WAL first records transaction begin to denote that a transaction has begun. Then, as the client issues requests, WAL records the type of requests, and KV pairs. Finally, when the client commits a transaction, WAL records the transaction commit. By doing so, the KV store can guarantee consistency and recover a consistent state when a system failure occurs.

In our proposed scheme, it is important to guarantee the consistency of the remapping table. Because we utilize a remapping table to reshape the original keys to sequential keys, the remapping table must be restored to access the original key. To guarantee the consistency of the remapping table, we flush the remapping table to the persistent storage device when the transaction is committed. By flushing the remapping table before committing a transaction, our proposed scheme can guarantee crash consistency on the remapping key information of the committed transaction.

IV. EVALUATION

In this section, we present evaluation results with the existing and proposed scheme. For evaluation, we used a single machine equipped with Intel(R) CPU i7-4790 @ 3.6GHz processors which has 4 physical cores and 8 cores with hyper-threading. The machine is equipped with 8GB of DRAM memory. To evaluate our scheme with various storage devices, we used a Samsung 840 PRO SSD which is a widely used SATA SSD, and Samsung PM1725a which is a widely used enterprise-grade NVMe PCIe SSD. Thus, we evaluated our scheme in both consumer-grade and enterprise-grade SSDs.

For the operating system, we used 64-bit Linux Ubuntu 16.04. Also, we used ext4 file system [21] which is the default file system of Linux operating system. For the KV database, we used the WiredTiger KV store which is a default storage engine for the widely used MongoDB database system [8]. For benchmark, we used widely used dbbench benchmark [14] as a micro-benchmark, and Yahoo! Cloud Serving Benchmark (YCSB) [22] and Sysbench's OLTP benchmark [23] as macro benchmarks. To evaluate our proposed scheme, we modified the WiredTiger storage engine to support reshaping operations and manage the remapping table. The evaluation results with the existing WiredTiger KV store are labeled as *Existing* and the results using the proposed key value reshaping scheme are labeled as *Proposed*.



(a) fill-sequential

(b) fill-random

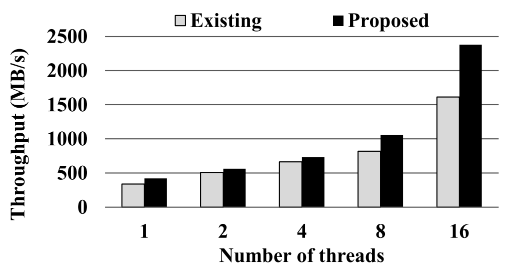
FIGURE 8. Dbbench throughput with fillsequential and fillrandom workload using SATA SSD.

A. MICRO-BENCHMARK

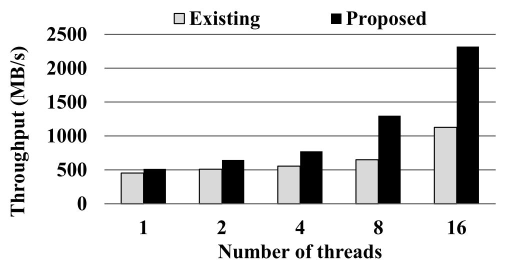
For micro-benchmark, we used dbbench benchmark which is the default benchmark used in LevelDB [14]. We configured the benchmark to insert 100 million key-value pairs with a value size of 100 bytes. We ran a benchmark with sequential and random insert workload with the thread count of 1, 2, 4, 8, and 16.

Figure 8 shows the sequential and random insert performance using SATA SSD with the existing and proposed schemes with a different number of threads. In the case of sequential insert workload, as shown in Figure 8(a), the proposed scheme increased the performance of the KV store by 1%, 45%, 43%, 67%, and 47% when 1, 2, 4, 8, and 16 threads were used, respectively. When the number of threads is 1, the performance of the proposed scheme is similar to that of the existing scheme since the requests are issued sequentially. When multiple threads are used, the proposed scheme outperforms the existing scheme as the requests from multiple threads create a random insert pattern in the perspective of the storage device. By reshaping random insert requests to sequential insert requests, the proposed scheme can improve performance.

In the case of random insert workload, as shown in Figure 8(b), the proposed scheme increased the performance by 1%, 7%, 65%, 201%, and 281% at the thread count of 1, 2, 4, 8, and 16, respectively. This is because our proposed scheme transforms random insert workload into sequential insert workload, improving node access pattern in the KV store and exploiting sequential write performance of SSDs. Note that performance improvement increases in both sequential and random insert workload as the number of threads increases. This is because, as there is more number of KV store threads, the KV store can issue more insert requests to SSD, resulting in more performance gain.

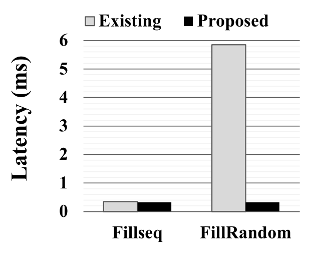


(a) fill-sequential

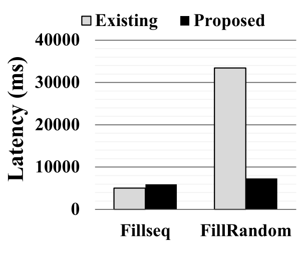


(b) fill-random

FIGURE 9. **Dbbench throughput with fillsequential and fillrandom workload using NVMe SSD.**



(a) Average latency

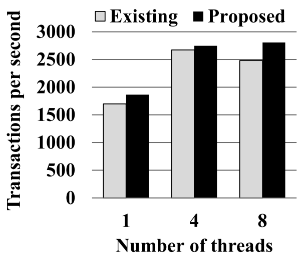


(b) Tail latency

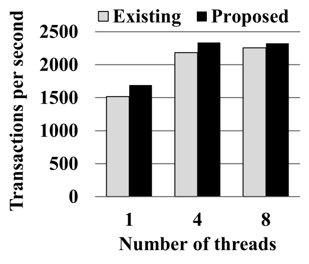
FIGURE 10. **Average and tail latency using SATA SSD.**

Figure 9 shows the sequential and random insert performance using NVMe SSD with the existing and proposed schemes. As shown in Figure 9(a) and 9(b), the proposed scheme improves the performance of the KV store by up to 47% and 106% in the case of sequential and random insert workloads, respectively. The performance improvement using NVMe SSD is higher than that of SATA SSD. This is because NVMe SSDs offer higher parallelism compared with SATA SSDs. By reshaping keys into sequential order, our proposed scheme can allow KV stores to issue I/O requests faster than the existing scheme, exploiting high parallelism of NVMe SSD. Thus, these results show that our scheme can improve the performance in both SATA based and NVMe based SSDs.

Figure 10 shows the average and tail latency of the wiredtiger KV store with the existing and proposed scheme when sequential and random insert workload is used. As shown in the figure, the average and tail latency of the existing and proposed scheme is similar when the sequential workload is used. The tail latency of the proposed scheme is slightly higher than that of the existing due to the remapping table management overhead. As the proposed scheme writes



(a) workload A



(b) workload F

FIGURE 11. **YCSB transaction per second using SATA SSD.**

the remapping table for crash consistency, the tail latency was slightly higher. In the case of random workload, both the average and tail latency of the proposed scheme are lower than those of the existing scheme. The average and tail latency of the random workload is similar to those of the sequential workload in the case of the proposed scheme. This is because the proposed scheme transforms random insert requests into sequential insert requests. Thus, the latency is similar for both types of workloads. These evaluation results show that the proposed scheme can improve the performance of KV store without significant overhead in the perspective of average and tail latency.

B. MACRO-BENCHMARK

For macro-benchmark, we used YCSB [22] and Sysbench's OLTP benchmark [23]. Both benchmarks are based on real-world applications and are widely used to evaluate the performance of the database in a complex scenario. To evaluate our scheme, we used MongoDB [8] as a database and used WiredTiger KV store with the existing and proposed scheme as the internal storage engine. Since WiredTiger is a default I/O engine used to store the data for MongoDB, this allows us to evaluate our proposed scheme in a real-world scenario.

1) YCSB

YCSB benchmark is a benchmark based on the cloud data servicing framework. To evaluate our scheme, we used two workloads from the benchmark: workload A and workload F. Workload A is an update heavy workload where 50% of requests are read requests and the other 50% of requests are update requests. On the other hand, workload F is constituted by 50% of read requests and 50% of read-modify-write requests. We choose these two workloads since both workloads have read and update requests which are not supported by dbbench. Thus, these workloads can evaluate our scheme in a complex real-world scenario. For configuration, we used the default configuration.

Figure 11 and 12 show the performance of YCSB benchmarks using SATA and NVMe SSDs. In the case of SATA SSD, as shown in Figure 11, the proposed scheme improves the performance compared with the existing scheme by up to 13% and 12% for workload A and F, respectively. In the case of NVMe SSD, as shown in Figure 12, the proposed scheme improves the performance compared with the

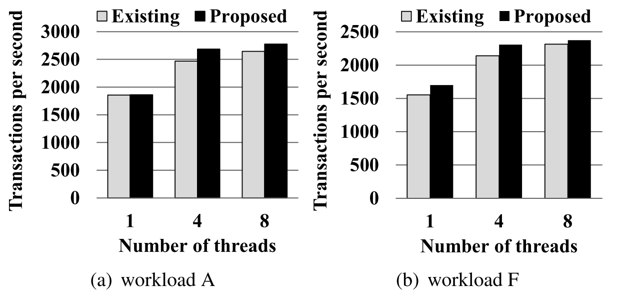


FIGURE 12. Transactions per second in YCSB benchmark using NVMe SSD.

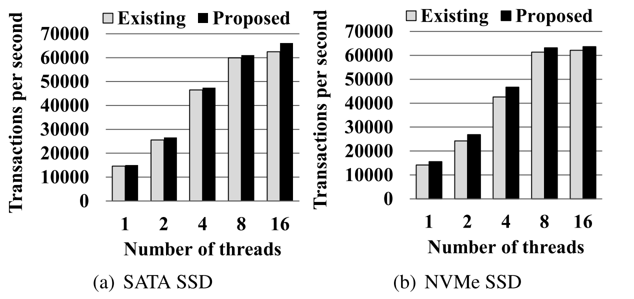


FIGURE 13. Transactions per second with default workloads in sysbench OLTP benchmark.

existing scheme by up to 9% and 10% for workload A and F, respectively. Compared with dbbench, the performance benefit of our scheme is lower because MongoDB database is used on top of WiredTiger KV store. This is because the requests are processed at MongoDB first before being sent to the underlying wiredTiger which creates an overhead that cannot be improved from our scheme. However, since our scheme improves the performance after the requests are processed and sent to the KV store, the performance results using the proposed scheme are higher than that of the existing scheme. Thus, these results show that our scheme can improve performance when complex workloads are used.

2) SYSBENCH

For another macro benchmark, we used online transaction processing (OLTP) workload included in the widely used Sysbench benchmark [23]. Similar to YCSB benchmark, we used MongoDB database and used WiredTiger as the underlying I/O engine. For configuration, we used 2 collections with 10 million documents each and configured the workload so that it performs equal parts of read, update, and insert requests. We report transactions per second as a performance metric.

Figure 13 shows the performance results using SATA and NVMe SSD. As shown in Figure 13(a) and Figure 13(b), the proposed scheme improves the performance by up to 6% and 11% in the case of SATA and NVMe SSDs, respectively. Similar to the results using YCSB, the performance gain of our proposed scheme is hidden by the overhead of using a relational database. However, the performance of the proposed scheme is always higher than that of the existing scheme, suggesting that our scheme can improve the performance of complex workloads.

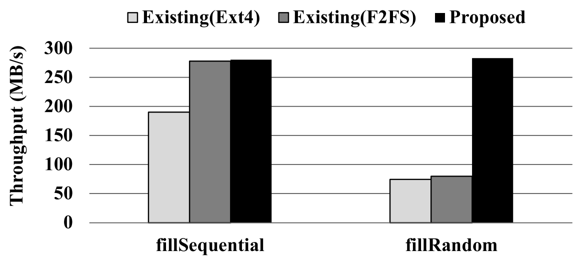


FIGURE 14. Dbbench throughput with F2FS file system.

TABLE 1. Overhead analysis with random insert workload.

Operation	time (ms)	Percentage
Runtime	30,734	100%
Prpoposed scheme	644	2.10%
fwrite_remapping table	577	1.88%
remapping table insert	48	0.16%

C. COMPARISON WITH ANOTHER SCHEME

To compare the proposed scheme with another log-structured scheme, we used F2FS file system which is a widely used log-structured file system [19]. We used F2FS as the underlying file system for the existing wiredtiger KV store. Figure 14 shows the performance results of the existing scheme with ext4 and F2FS, and the proposed scheme. As shown in the figure, the performance of the existing scheme with F2FS is similar to that of the proposed scheme when sequential insert workload is used. This is because F2FS also writes the data sequentially similar to the proposed scheme. Thus, the performance benefit of the proposed scheme can be achieved when F2FS is used. In the case of the random insert workload, the performance of the existing scheme with ext4 and F2FS is similar while the performance of the proposed scheme is higher. This is because the overhead of inserting and rebalancing B+ tree is greater when a random insert workload is used compared with the overhead when a sequential insert workload is used. Thus, even if F2FS writes the data sequentially, the KV store cannot issue the write request fast enough, not utilizing the underlying storage device sufficiently. However, the proposed scheme transforms the random keys into sequential keys, enabling KV store to issue insert requests rapidly and fully exploiting the underlying storage device. Thus, the proposed scheme can improve the performance of KV stores further compared to another log-structured scheme.

D. OVERHEAD ANALYSIS

To analyze the overhead of the proposed scheme, we executed random insert workload and random update workload from dbbench benchmark using identical settings. Table 1 and 2 show the overhead analysis of the proposed scheme. We measured the overall overhead of our proposed

TABLE 2. Overhead analysis with update workload in dbbench.

Operation	time (ms)	Percentage
Runtime	62,351	100%
Proposed scheme	1,788	2.87%
fwrite_remapping table	1,054	1.69%
remapping table search	68	0.11%
remapping table delete	193	0.31%
remapping table insert	86	0.14%

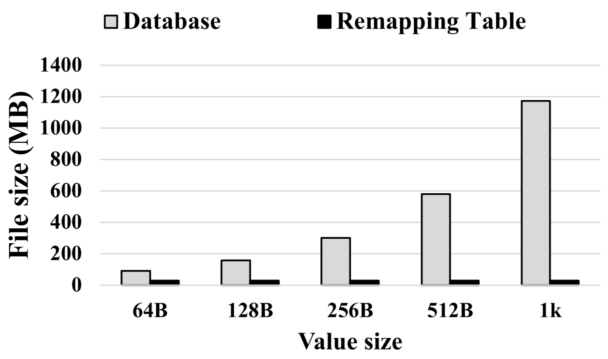


FIGURE 15. File size of database and remapping table.

scheme (Proposed scheme), remapping table persistence overhead (fwrite_remapping table), and remapping table operation overhead (remapping table search, delete, insert). In the case of random insert workload, as shown in Table 1, the overhead of the proposed scheme is less than 3% of the runtime. Most of the overhead from the proposed scheme was from writing a remapping table to a file for crash consistency. Since remapping table insert operations are memory operations, the overhead from the remapping table insert was minor compared to the remapping table write operation. In the case of the update operation, similar to the random insert workload, the overhead of the proposed scheme is less than 3% of the runtime. Although an update operation requires search operation to find the original key, delete operation to delete the original key (on-the-fly cleaning operation), and insert operation to insert a new sequentialized key into the remapping table, the overhead from the remapping table operation is less than 1% of the runtime as they are memory operation. Similar to the random insert workload, writing the remapping table to a file induced the majority of the overhead from the proposed scheme. These overhead analysis results show that the proposed scheme can improve performance with minimal overhead.

Since the proposed scheme writes a remapping table to a file to guarantee crash consistency, it is important to analyze the size of the remapping table. Figure 15 shows the file sizes of the database and remapping table when different key sizes are used. We used sequential insert workload from dbbench benchmark with identical settings. As shown in the figure, the size of the remapping table is identical (32MB) while the size of the database increases as the value size increases. The size of the remapping table is 34% and 2.5% of the database size when the value size is 64B and 1K, respectively. This is because the size of the remapping table is affected by the size of the key since the remapping table stores the original and remapped key values. Thus, the analysis results show that while the proposed scheme writes additional data due to the crash consistency, the overhead becomes less significant as the value size increases.

V. RELATED WORK

A. OPTIMIZING STORAGE SYSTEMS FOR FLASH-BASED SSDs

There have been many studies on understanding the characteristics of SSDs and improving the I/O performance. F2FS [19] is a flash-friendly file system that improved a log-structured file system which is widely used to exploit the performance of SSDs by transforming random requests into sequential requests. SHRD [18] is a request reshaping scheme, which includes a storage device driver and the FTL of an SSD. SHRD improves the spatial locality for FTL mapping table accesses by logging random requests in the storage and reordering these requests. Ziggurat [24] is a tiered storage file system that buffers file writes to NVMM and DRAM. By buffering writes, it sends batched sequential write requests to underlying flash storage which can increase the performance and garbage collection efficiency.

Our study is in line with these studies [18], [19], [24] in terms of transforming random writes to sequential writes to improve data locality inside SSD. In contrast, our study focuses on improving spatial locality in KV stores instead of storage devices and systems (i.e., file systems and SSDs). This allows KV stores with our proposed scheme to perform efficient in-memory operations and utilize various file systems and storage devices.

B. OPTIMIZING KEY-VALUE STORES FOR FLASH-BASED SSDs

There have been many studies on optimizing the KV store. Wisckey [5] improved the performance of KV store by separating keys from values. By storing keys and values separately, it stores values in a log-structured manner, reducing data movement and write amplification. NVMKV [7] is a new SSD that supports KV store operation with the cooperation of FTL inside SSD. By exploiting the internal characteristics of flash memory, it can reduce the write amplification and improve the performance of KV store. Triad [25] optimized data locality of KV store by identifying hot and cold data and storing them separately. By buffering hot data updates in memory, it can reduce frequent I/O operation to disks and perform batched I/O operation on the buffered data. ForestDB [26] improved the performance and reduced the storage overhead of KV stores by utilizing HB+ trie.

Our study is in line with these studies [5], [7], [25], [26] in terms of improving the performance of KV stores by utilizing the characteristics of storage devices. These studies transform the requests by redefining data structures and FTL algorithm to exploit the high performance of modern storage devices. In contrast, our study focuses on improving the performance by reshaping keys inside KV store. This allows our scheme to improve the data locality without modifying the data structure of KV store or the FTL layer inside storage devices.

VI. CONCLUSION

In this paper, we propose a key reshaping scheme for KV store to improve I/O performance of flash-based SSDs. To do this, we design and implement a key reshaping scheme that transforms random insert requests into sequential insert requests. This enables faster insert request processing in KV store and increases data locality in SSD. In addition, to ensure correct read and update operation, we proposed a remapping table that records the relationship between the original and reshaped keys and guarantees the consistency of the KV store and remapping table through WAL. To evaluate the proposed scheme, we used micro-benchmark (dbbench) and macro-benchmarks (YCSB and Sysbench) with MongoDB and WiredTiger KV store. The evaluation results show that our proposed scheme can improve the performance of KV store by up to 281% compared with the existing scheme.

REFERENCES

[1] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 1–26, Jun. 2008.

[3] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom, "SFS: Random write considered harmful in solid state drives," in *Proc. FAST*, vol. 12, 2012, pp. 1–16.

[4] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *Proc. USENIX Annu. Tech. Conf.*, vol. 57, Jun. 2008, pp. 1–14.

[5] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: Separating keys from values in SSD-conscious storage," *ACM Trans. Storage*, vol. 13, no. 1, pp. 1–28, Mar. 2017.

[6] S.-M. Wu, K.-H. Lin, and L.-P. Chang, "KVSSD: Close integration of LSM trees and flash translation layer for write-efficient KV store," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 563–568.

[7] L. Marmol, S. Sundararaman, N. Talagala, R. Rangaswami, S. Devendrappa, B. Ramsundar, and S. Ganesan, "NVMKV: A scalable and lightweight flash aware key-value store," in *Proc. 6th USENIX Workshop Hot Topics Storage File Syst. (HotStorage)*, 2014, pp. 1–5.

[8] K. Chodorow, *MongoDB: Definitive Guide: Powerful Scalable Data Storage*. Sebastopol, CA, USA: O'Reilly Media, 2013.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, 2007.

[10] X. Wu, Y. Xu, Z. Shao, and S. Jiang, "LSM-trie: An LSM-tree-based ultra-large key-value store for small data items," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2015, pp. 71–82.

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

[12] A. Petrov, "Algorithms behind modern storage systems: Different uses for read-optimized B-trees and write-optimized LSM-trees," *Queue*, vol. 16, no. 2, pp. 31–51, Apr. 2018.

[13] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-tree filesystem," *ACM Trans. Storage*, vol. 9, no. 3, pp. 1–32, Aug. 2013.

[14] A. Dent, *Getting Started With LevelDB*. Birmingham, U.K.: Packt Publishing Ltd, 2013.

[15] F. Havasi, "An improved B+ tree for flash file systems," in *Proc. Int. Conf. Current Trends Theory Pract. Comput. Sci.* Berlin, Germany: Springer, 2011, pp. 297–307.

[16] H.-W. Fang, M.-Y. Yeh, P.-L. Suei, and T.-W. Kuo, "A flash-friendly B^+-tree with endurance-awareness," in *Proc. 9th IEEE Symp. Embedded Syst. Real-Time Multimedia*, Oct. 2011, pp. 29–36.

[17] Y. Li, P. P. C. Lee, J. C. S. Lui, and Y. Xu, "Impact of data locality on garbage collection in SSDs: A general analytical study," in *Proc. 6th ACM/SPEC Int. Conf. Perform. Eng.*, Jan. 2015, pp. 305–315.

[18] H. Kim, D. Shin, Y. H. Jeong, and K. H. Kim, "SHRD: Improving spatial locality in flash storage accesses by sequentializing in host and randomizing in device," in *Proc. 15th USENIX Conf. File Storage Technol. (FAST)*, 2017, pp. 271–284.

[19] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2fs: A new file system for flash storage," in *Proc. 13th USENIX Conf. File Storage Technol. (FAST)*, 2015, pp. 273–286.

[20] J. Y. Shin, M. Balakrishnan, T. Marian, and H. Weatherspoon, "Gecko: Contention-oblivious disk arrays for cloud storage," in *Proc. 11th USENIX Conf. File Storage Technol. (FAST)*, 2013, pp. 285–297.

[21] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: Current status and future plans," in *Proc. Linux Symp.*, vol. 2. Princeton, NJ, USA: Citeseer, 2007, pp. 21–33.

[22] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput. (SoCC)*, 2010, pp. 143–154.

[23] A. Kopytov, "Sysbench manual," in *Proc. MySQL AB*, 2012, pp. 2–3.

[24] S. Zheng, M. Hoseinzadeh, and S. Swanson, "Ziggurat: A tiered file system for non-volatile main memories and disks," in *Proc. 17th USENIX Conf. File Storage Technol. (FAST)*, 2019, pp. 207–219.

[25] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka, "TRIAD: Creating synergies between memory, disk and log in log structured key-value stores," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2017, pp. 363–375.

[26] J.-S. Ahn, C. Seo, R. Mayuram, R. Yaseen, J.-S. Kim, and S. Maeng, "ForestDB: A fast key-value storage system for variable-length string keys," *IEEE Trans. Comput.*, vol. 65, no. 3, pp. 902–915, Mar. 2016.

SUNGGON KIM received the B.S. degree in computer science from the University of Wisconsin-Madison, Madison, USA, in 2015. He is currently pursuing the Ph.D. degree in computer science and engineering with Seoul National University. He was an Intern at Lawrence Berkeley National Laboratory, California City, USA, from 2018 to 2020. His research interests include file systems, cloud computing, distributed systems, and operating systems.

YONGSEOK SON received the B.S. degree from Ajou University, in 2010, and the M.S. and Ph.D. degrees from Seoul National University, in 2012 and 2018, respectively. He was a Postdoctoral Research Associate at the University of Illinois at Urbana-Champaign. Currently, he is an Assistant Professor with the School of Computer Science and Engineering, Chung-Ang University. His research interests include operating, distributed, and database systems.