RemapCom: Optimizing Compaction Performance of LSM Trees via Data Block Remapping in SSDs

Yi Fan [†], Yajuan Du [†,*], and Sam H.Noh [‡]
[†] Wuhan University of Technology, Wuhan, China
[‡] College of Engineering, Virginia Tech, Blacksburg, Virginia, USA
{fy2685018622,dyj}@whut.edu.cn, samhnoh@vt.edu.cn
[*] The corresponding author

***Abstract*—In LSM-based KV stores, typically deployed on systems with DRAM-SSD storage, compaction degrades write performance and SSD endurance due to significant write amplification. To address this issue, recent proposals have mostly focused on redesigning the structure of LSM trees. In this paper, we observe the prevalence of data blocks that are are simply read and written back without being altered during the LSM-tree compaction process, which we refer to as Unchanged Data Blocks (UDBs). These UDBs are source of unnecessary write amplification leading to performance degradation and shortening of SSD lifetime. To address this duplication issue, we propose a remapping-based compaction method, which we call RemapCom. RemapCom considers the identification and retention by designing a lightweight state machine to track the status of the KV items in each data block as well as designing a UDB retention strategy to prevent data blocks from being split due to adjacent intersecting blocks. We implement a prototype of RemapCom on LevelDB by providing two primitives for the remapping. Compared to the state-of-the-art, evaluation results demonstrate that RemapCom can reduce the write amplification by up to 53%.**

***Index Terms*—LSM-tree, Compaction, SSDs, Data Remapping**

I. Introduction

With the development of big data techniques, the structure of Log-Structured Merge (LSM) tree [1], [2] has been widely used in key-value (KV) based databases.

By directly appending data, random KV writes can be converted into sequential ones, resulting in improved write performance. KV data are first organized to be data blocks, and then multiple data blocks are formed into a Sorted String Table (SSTable). These SSTables are stored into multiple levels on disks. When the storage volume increases, compaction is invoked to merge new data and old data in selected SSTables to be new SSTables, which would be written back to disks [3]. This induces extra I/O overhead and duplicate writes. For Solid-State Drives (SSDs)) with limited endurance, these duplicate writes would severely consumes its Program and Erase (P/E) cycles and aggravates the write amplification problem.

In order to address the duplication issue in an LSM tree based KV store, this paper proposes a remapping-based compaction method, which we call RemapCom. First, we notice that the size of data blocks (typically 4KB) is often similar with the size of an SSD page. Thus, we determine the data block to be the remapping granularity. Second, we consider the identification and retention of duplicate data during compaction, which we refer to as unchanged data blocks (UDBs). Then, in order to identify UDB, we design a lightweight state machine to track the status of the KV items in each data block, with the assistance of a lazy write-back scheme. Subsequently, to take full advantage of the remapping benefit, we design a UDB retention strategy to prevent data blocks from being split due to adjacent intersecting blocks. This improves the ratio of UDB and further optimizes system performance. Finally, we implement two primitives in RemapCom to remap the new logical address onto the old physical address of these UDBs to avoid duplicate writes.

The contributions of this paper are summarized as follows.

- We perform a preliminary study to uncover the prevalence of UDB in real-world benchmarks.
- We propose RemapCom, an SSD remapping-based compaction method for LSM-based KV stores.
- We design a UDB retention strategy in RemapCom to increase the ratio of UDB.
- We implement RemapCom in LevelDB by providing two primitives: `getLPN` and `remap` to support data block remapping.
- We evaluate RemapCom in real-world benchmarks and experimental results show that it can reduce write amplification by up to 53%.

The remainder of the paper is organized as follows: Section II introduces the background and motivation of RemapCom. Section III presents its detailed design. Section IV demonstrates experiment setup and results. Section V introduces related works and Section VI concludes this paper.

II. Background and Motivation

In this section, we first present the background of flash-based SSDs and SSD remap strategy. Then, we present the basics of LSM-trees and then discuss the motivation of this paper.

A. Remapping-based SSDs

In flash-based SSDs, there are two core components, the SSD controller and flash chips. The SSD controller handles read and write operations and manage the flash arrays via Logical-to-Physical (L2P) address mapping, garbage collection (GC), and other functions. Flash chips are organized hierarchically to store data [4]. Page is the unit to perform read and write, while block is the unit to perform data erasures [5]–[7].

The SSD remap strategy is a way to reuse the duplicate data involved in data copying, data moving, and journaling

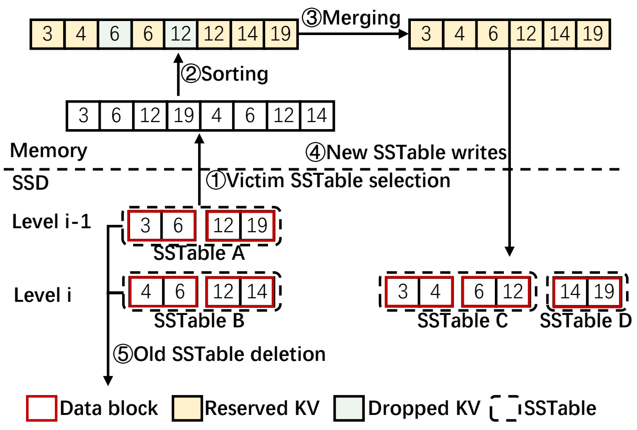

Fig. 1. Compaction process of LSM-tree.

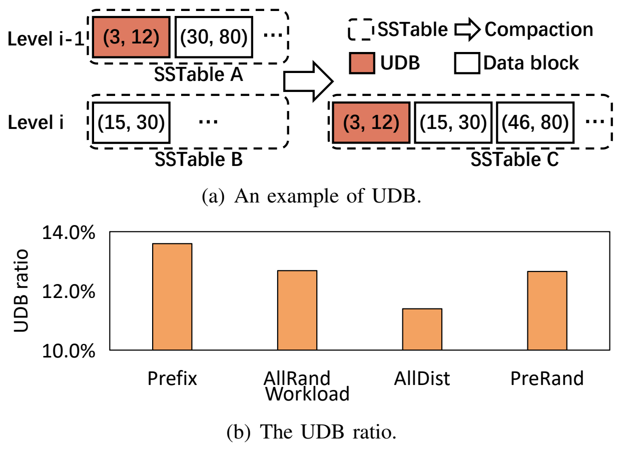

Fig. 2. Prevalence of UDB in real-world benchmarks.

by just modifying the mappings in FTL [8]–[10]. When the host moves data from old pages to new ones, the conventional approach would first have the data copied into new pages and have the FTL establish the new mappings, then mark the old pages invalid and delete the old L2P mappings. In the remapping approach, changing the old L2P mapping to the mapping between the new logical address and the old physical address is all that is needed, and the old data still remains valid. Compared with the conventional approach, the remapping approach reduces write amplification, thereby improving write performance and the endurance of the SSD.

Mapping consistency is an issue that needs to be considered in the remapping approach [8]–[11]. Due to the out-of-place update characteristics of flash memory, after remapping, the relevant P2L mappings in the Out-of-Band (OOB) space of flash pages cannot be modified even as the L2P mappings have changed. This becomes a problem as a wrong L2P table may be rebuilt during GC and power-off recovery. Existing works propose to maintain a remap table to solve this mapping consistency problem [8]–[10].

B. Log-structured Merge Trees

Log-structured merge trees organize data to be a memtable in memory and SSTable in SSDs [2], [12]. In SSDs, SSTables are organized in multiple levels. KV items in an SSTable is organized into multiple data blocks, each of which has a certain size (often 4KB by default) and is sorted in order of keys. For example, in Figure 1, SSTable A in Level $i-1$ contains two data blocks. One contains keys of 3 and 6, while the other contains keys of 12 and 19. As the storage volume increases, SSTable in a higher level would be compacted into a lower level. Compaction process can be summarized into five steps as shown in the example of Figure 1.

Step ①: Victim SSTable selection. Select an SSTable in the level of L_{i-1} as a victim and find SSTables in the next lower level L_i whose key ranges overlap with the victim SSTable.

Step ②: KV sorting. The KV items in victim SSTable A and overlapped SSTable B are combined together and sorted in ascending order of keys.

Step ③: KV merging. The sorted KV items are merged. In detail, compaction traverses the sorted KV items one by one. The KV items would be dropped when they are denoted as invalid or are old versions of the same keys. For example, the KVs of 6 and 12 marked as green in Figure 1 are dropped. Otherwise, the KV item would be reserved. Finally, the reserved KV items are re-organized as new SSTables.

Step ④: New SSTable writes. The re-organized new SSTables are written back into SSDs.

Step ⑤: Old SSTable deletion. The old SSTables are deleted when the new SSTable write is finished.

Note that all the reserved KV items in this example, marked as yellow in Figure 1, are read into memory and then written back to the SSD remain unchanged. That is, they are all duplicate data viewed from the SSD level.

C. Motivation

During compaction, there exists data blocks whose entire KV items do not change before and after compaction. We define such a data block as an Unchanged Data Block (UDB). Consider the example given in Figure 2(a), where the red box is a UDB. In this example, we use (K_s, K_e) to represent the range of each data block in SSTables where K_s and K_e are the start and end keys, respectively. The data block with a range of (3,12) is considered to be a UDB.

We perform a preliminary study on the Mixgraph benchmark [13]. Mixgraph is collected from Facebook's social graph workload with query composition and key access patterns, and contains four workloads of Prefix Dist, All Random, All Dist, and Prefix Random. The details of these workloads are introduced by Cao et al. [13]. We collect the ratio of UDB by directly comparing the data blocks before and after compaction. 50 million mixed read-write requests are issued in the four workloads. The results are presented in Figure 2(b) where we observe that the UDB ratios for the four workloads are 13.6%, 12.7%, 11.4%, and 12.7%, respectively.

These results show the prevalence of UDBs in the real world, and if we can apply the SSD remapping approach to remap these UDBs, write amplification due to compaction can be

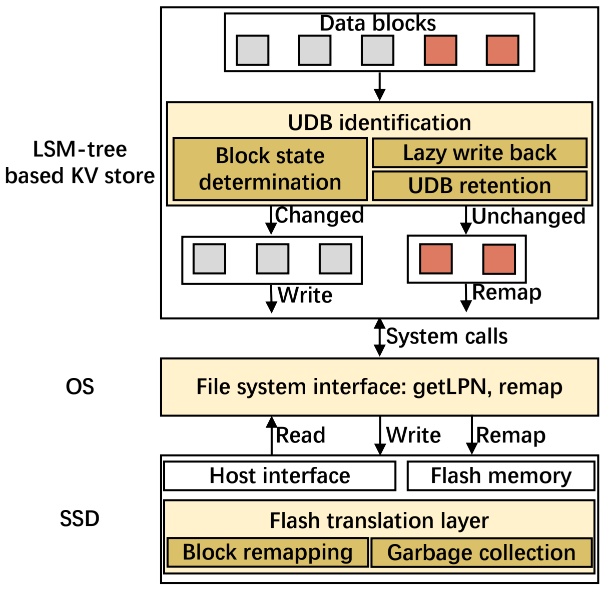

Fig. 3. Architectural overview of RemapCom.

reduced, thereby improving compaction performance as well as elongating SSD lifetime. To this end, we propose RemapCom that exploits this observation.

III. REMAPCOM: REMAPPED COMPACTION

In this section, the architectural overview of RemapCom is first illustrated, and then each component of RemapCom is presented in detail.

A. Overview of RemapCom

Figure 3 presents the architectural overview of RemapCom as integrated with an existing remapping-based SSD. RemapCom comprises three main components, namely, UDB identification, implementation interfaces, and data remapping. UDB identification is designed within the LSM-tree based KV store, with three key mechanisms. Specifically, block state determination checks the KV items with a lightweight state machine to determine whether the data block is unchanged or not, lazy write back is used to buffer the KV items temporarily before the final state is determined, while UDB retention aims to generate more UDBs than that are obvious. Two primitives of `getLPN` and `remap` are designed in the file system level. Data remapping is implemented within the remapping-based SSD, mainly maintaining the original remapping method but considering the new primitives.

B. Block State Determination

To determine the state of data blocks, RemapCom utilizes a lightweight state machine that is integrated into the KV merging process of SSTable compaction. When compaction begins, a state machine is created in memory, while when the last KV item is merged, it is destroyed. The state machine determines the data block's state as RemapCom traverses the sorted KV items during merging as described below.

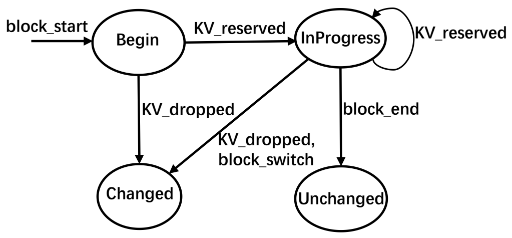

Fig. 4. The lightweight state machine.

A data block can be in four states: *Begin*, *InProgress*, *Changed*, and *Unchanged* as shown in Figure 4. *Begin*: A data block is in this state when RemapCom traverses the first KV item of the data block.

InProgress: A data block is in this state when no KV changes are detected in the data block up to the current time, i.e., this data block has the potential to be a UDB.

Changed/Unchanged: A data block is in either of these states once its final state has been determined by the state machine. If the state is *Unchanged* it is a UDB or otherwise, it is a block whose content has changed.

Traversing each KV item would trigger one of the following events in the state machine, which takes the block into one of above four states.

block_start/block_end: First or last KV item in block has been traversed.

KV_reserved: This event is triggered when the KV item that RemapCom is checking in the block is to be reserved. In Figure 5, the first two KV items with Key 15 and Key 26 in block A are both reserved ones. Thus, the first item transitions the state from *Begin* to *InProgress*, while the second item transitions from *InProgress* to *InProgress* in Figure 4.

KV_dropped: This event is triggered when the checked KV item is to be dropped, that is, become invalid. For example, the KV item with Key 30 of block A in Figure 5 is a dropped item. Accordingly, the state of the data block transitions to *Changed*. This can happen from *Begin* or from *InProgress*.

block_switch: This event is triggered when the RemapCom traversal moves from one block to another block. Recall that RemapCom is in the process of merging two SSTables, and this is happening with KV items in two data blocks of the SSTables. From a single data block point of view, switching traversal by RemapCom to a different data block means that the next KV item traversed is in a different block. This, in turn, means that the content of the new block will include KV items from two different blocks. Thus, the state of the block transitions to *Changed*. For example, a block_switch happens in Figure 5 where RemapCom traverses to Key 30 in Block B after traversing Key 30 in Block A. As a result, the newly generated data block contains KV items from block A and block B, which means block A is not a UDB.

C. Lazy Write Back

For RemapCom, to take advantage of remapping, write back of blocks must be delayed until their state is determined. Thus,

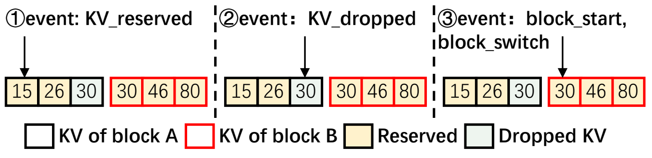

Fig. 5. Examples of events in the state machine.

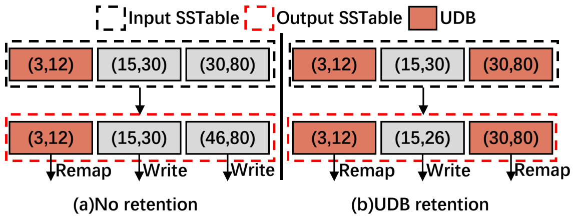

Fig. 6. An example of UDB retention.

a data buffer is used to cache the KV items when their state is *InProgress*. The KV items are, thus, lazily written back to SSDs under two scenarios. First, when the state transitions to the *Changed* state, the traversed KV items buffered in the data buffer are written into the output SSTable just like the original compaction process. Second, the state is transitioned to *Unchanged* state. This means that traversed KV items in the data buffer form a UDB. Thus, RemapCom writes this UDB by just using a remapping command, instead of the write command, to write these KV items into the output SSTable.

D. UDB Retention

Even if a data block is determined to be in an *Unchanged* state by the state machine, it may be split into adjacent data blocks due to the flash page alignment, eventually converting it to a changed block. For example, in Figure 6(a), data blocks with key ranges of (3, 12), (15, 30), and (30, 80) are compacted into three data blocks with the key ranges of (3, 12), (15, 30) and (46, 80). Although the data block with key range (30, 80) is determined to be a UDB, one of its KV item with Key 30 is used to fill up the left space of its adjacent block.

To address this issue, RemapCom uses a strategy that we call UDB retention, which allows the UDBs determined by the state machine to remain a UDB as they are written to the SSD. That is, in the example above, UDB retention allows the (30, 80) key range block to be retained as a UDB. With this strategy, RemapCom can generate more UDBs. However, internal fragmentation will incur in the adjacent block wasting some storage space. For example, as shown in Figure 6(b), the original (15, 30) block will not fill entirely, being written as a (15, 26) block. There exists a trade-off between storage space waste due to internal fragmentation and benefits by block remapping, and the final results would be shown in Section IV.

E. Primitives and Block Remapping

RemapCom designs two primitives that are encapsulated into the file system interface, enabling the application layer to directly utilize the remap function.

`getLPN(fileno, offset)`: This primitive obtains the logical page number (LPN) of the UDB. Given the SSTable's file number, `fileno`, and its offset in SSTable, `offset`, the LPN of the UDB is obtained by using the *ioctl* system call.

`remap(src_LPN, dst_LPN, length)`: This primitive sends the remap signal to the SSD along with the necessary information for remapping: `src_LPN`, representing the original start LPN of UDB, `dst_LPN`, representing the new start LPN of UDB, and `length`, representing the number of pages involved in the UDB. We extend the `nvme_ioctl` system call to support this primitive.

With these two primitives, there are three main steps for remapping the UDBs in SSDs as illustrated below.

Step 1: Obtain the logical page number corresponding to the UDB in SSD by calling `getLPN`.

Step 2: Send remap requests to the file system by calling `remap`. For the example in Figure 6(a), RemapCom uses remap command to notify the SSD that the UDB with the key range (3, 12) needs to be remapped.

Step 3: Handle remap requests in SSDs. With the assistance of the `remap` primitive, SSD controller can perform remapping just like Remap-SSD [10].

F. Overhead Analysis

The overhead of our proposed RemapCom method is analyzed from three aspects.

State machine: Since we only need to track the state of each block, just two bits are sufficient to represent the four possible states. This means that RemapCom requires only two bits of memory per state machine. Because KV items in one SSTable have no overlaps, only one state machine is needed for each SSTable. Therefore, the maximum number of state machines in memory is the number of input SSTables attending the compaction. As a result, this overhead is minimal.

Lazy write back: The lazy write back strategy requires data buffers in memory to store the temporary KVs from the block while in the InProgress state. Its size is less than one data block. The extra storage space of RemapCom can be ignored.

Remapping-based SSDs: The remap-based SSD needs a remap table to ensure mapping consistency [9], [10]. RemapCom utilizes the same remapping method with existing studies and thus, does not induce extra overhead.

IV. Evaluation

This section evaluates the proposed RemapCom method. The experimental setup is first illustrated and results on microbenchmarks are then shown and analyzed. Finally, the evaluation results on real-world benchmarks are illustrated and analyzed.

A. Experimental Setup

Our experiments are conducted on FEMU [14], a popular NVMe SSD emulator based on QEMU. The Linux kernel of FEMU host system is Linux 5.15 and we format the emulated SSD as an EXT4 file system.

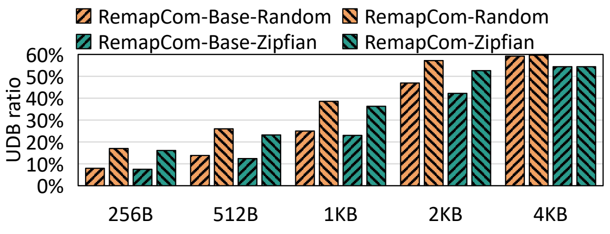

Fig. 7. UDB ratio for different KV sizes.

TABLE I
WRITE AMPLIFICATION REDUCTION IN DIFFERENT LEVELS

Level	1	2	3	4
RemapCom-Base	1%	26%	29%	32%
RemapCom	5%	47%	52%	53%

The emulated SSD has a capacity of 16GB. The flash page size of emulated SSD is 4KB and latencies of page read, page write, and block erase are $40\mu s$, $200\mu s$, $2ms$, respectively.

For the LSM-based KV store, we use LevelDB. Its data block size is 4KB and the SSTable size is 2MB. We denote the original LevelDB as "Baseline", LevelDB only with block state determination and UDB remapping, but no UDB retention as "RemapCom-Base", and LevelDB with all components of RemapCom as "RemapCom".

B. Results on Microbenchmarks

In this section, we use the db_bench microbenchmark for evaluation with two different distributions to the database. Since the *fillrandom* workload of db_bench can only generate random key-value items, we add *zipfian* distribution, which produces key-value pairs that conform to the principle of locality. The total size of the KV items written by the host is 4GB, while the number of KVs written varies depending on the value size. The key size is fixed to 16 bytes while the value size varies according to the KV item size which ranges from 256B to 4KB.

UDB Ratio: Figure 7 shows the UDB ratio during compaction for random write and zipfian workloads. We observe that as the KV size increases, the ratio of UDB in the SSTables increases. This is because, as the size of each KV increases, the number of KVs in each data block decreases, making it more likely for the data block to become an UDB. We also observe that the UDB ratio of RemapCom is higher than that of RemapCom-Base.This is because the UDB retention strategy prevents UDB from mixing with adjacent data blocks. When the KV size equals 4KB, the benefit of UDB retention is not obvious because the KV size is equivalent to the data block size and little mixture happens.

Write amplification: Figure 8 shows the write amplification results. Compared with LevelDB, RemapCom-Base and RemapCom both reduce write amplification significantly due to remapping the UDB. RemapCom-Base reduces SSD writes by 22.4% in *fillrandom* distribution and 20% in *zipfian* distribution, on average. RemapCom reduces SSD writes of these two types of benchmarks by 35.7% and 32.8% on average, respectively.

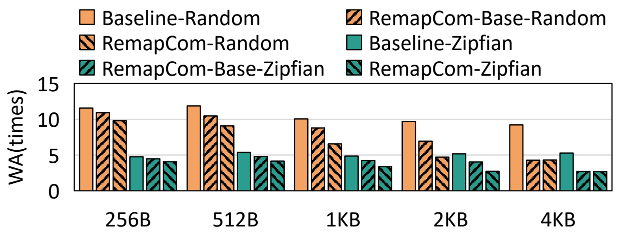

Fig. 8. Write amplification for different KV sizes.

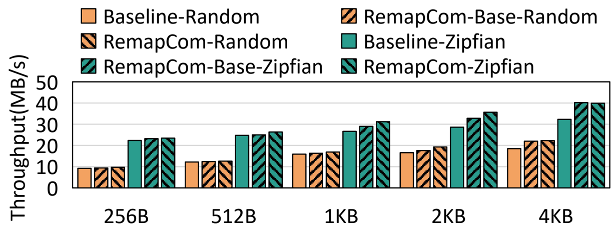

Fig. 9. Write performance for different KV sizes.

As the KV size increases, the benefit on write amplification from both RemapCom-Base and RemapCom becomes more pronounced. This is because there is a larger ratio of UDB for larger KV sizes as shown in Figure 7. We also conduct an detailed analysis of the write amplification in each level of LevelDB. Table I shows the SSD writes reduced by RemapCom-Base and RemapCom compared to LevelDB in each level for 1KB KV size. We find higher reduction in write amplification for higher levels. This is because that data blocks in higher levels are colder. This means that as the levels go up, it becomes more likely that the UDBs from the lower levels trickle up to the upper levels having an accumulation effect of UDBs. Thus, the UDBs increase reducing write amplification further.

Write performance: Figure 9 shows the write throughput results. Compared to LevelDB, RemapCom-Base improves write throughput by up to 30% with *fillrandom* and 25% with zipfian distribution. The reason is thatRemapCom avoids writing UDBs in SSTables, thus mitigating the write performance degradation caused by compaction. The improvement in write performance by RemapCom becomes more pronounced as the KV size increases. This is because when the KV size increases, the effect of RemapCom on write amplification reduction becomes more significant.

C. Real-World Benchmarks

In this section, we consider the performance of RemapCom on two real-world workloads, namely, the YCSB and Mixgraph benchmarks.

The characteristics of the YCSB benchmark workload is given in Table II [15]. For these experiments, we first load 800MB of KV items into the KV store. Then we run the six workloads, A through F, provided by YCSB on the KV store. In the experiments, we consider three different sizes of KV items 256B, 1KB and 4KB, which represent small, middle and large KV items, respectively.

TABLE II
DESCRIPTION OF YCSB WORKLOAD

Workload	Operations and Distribution
Load	100% inserts, uniform
A	50% updates, 50% reads, zipfian
B	5% updates, 95% reads, zipfian
C	100% reads, zipfian
D	5% inserts, 95% reads, latest
E	5% updates, 95% scans, zipfian
F	50% reads, 50% read-modify-writes, zipfian

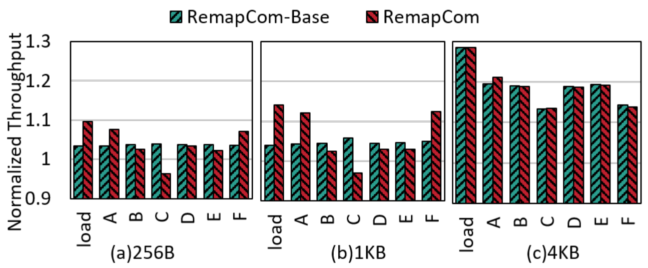

Fig. 10. Throughput of RemapCom normalized to LevelDB for YCSB.

Figure 10 shows the throughput results of RemapCom-Base and RemapCom normalized to the baseline. From this figure, we can obtain the following observations.

Observation 1: For the Load workload, RemapCom-Base increases throughput by 3.4%, 3.9%, and 28% compared to LevelDB for the three KV sizes, respectively. For RemapCom, the improvements are 9.7%, 14.1%, and 28%, respectively.

Observation 2: For the write-intensive workloads A and F, compared to LevelDB, RemapCom's improvements are 7.6%, 12% and 21.1% for workload A and 7.1%, 12.4% and 13.8% for workload F, for the respective KV sizes.

Observation 3: For the read-intensive workload C, we find that read performance in RemapCom-Base obtains 3.9%, 5.9%, and 13.3% improvement for the three KV sizes, respectively. The reason is that the read operations can trigger seek compaction [16], where UDBs can also be remapped. However, compared to LevelDB, RemapCom reduces throughput by 3.8% and 3.1% for 256B and 1KB KV sizes, respectively. This is because UDB retention separates UDB from adjacent blocks, which causes a certain amount of read and write amplification.

Observation 4: RemapCom also achieves an improvement in throughput for workloads with only a small portion of writes (i.e., B, D and E). The average improvements of all the KV sizes are 2.5% , 3.2 and 2.6% for the B, D and E workloads, respectively. These results show that RemapCom can also improve overall performance for read-intensive workloads.

We now show the results for the Mixgraph benchmark [13], which can better emuate the workloads of real-world KV stores such as ZippyDB(a distributed KV store) and UP2X(a distributed KV store for AI/ML services). In these experiments, all four workloads issue 50 million query requests. The test results are shown in Fig. 11. We observe that compared to LevelDB, RemapCom-Base and RemapCom reduces write

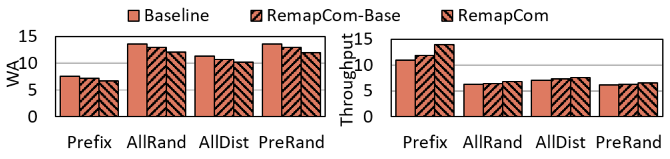

Fig. 11. WA (left) and throughput (right) results of Mixgraph benchmark

amplification by 4.5% and 11%, on average, indicating that significant benefits are possible with RemapCom.

Above results have shown that our proposed RemapCom method can effectively decrease write amplification and improve the performance of SSDs

V. RELATED WORK

There have been numerous studies to optimize compaction performance of LSM-trees. Dayan et al. propose to partition the LSM-tree's largest level into equally-sized files to ease the burden of merging overlapped SSTable files at a time [17]. In order to reduce the effect of frequently updated small KV items during compaction, L2SM removes the hotter and sparser KV items at an early stage [3]. Sun et al. propose to avert rewriting data that do not need to be updated during compaction into SSDs [18]. Thonagi et al. propose a new algorithm to decide whether to perform a partial merge to mitigate the compaction performance [19]. Hu et al. propose to limit the SSTables that participate in compaction thus mitigating the compaction performance [20]. Lee et al. propose to put SSTables with the similar lifetime in the same zone of ZNS, so that ZNS can efficiently clear them with the zone cleaning command [21]. As these mechanisms do not consider the compaction algorithm of the LSM-tree by using data remapping, RemapCom can be integrated with these approaches.

Some works consider to redesign the structure of LSM-trees for SSDs. KVSSD upgrades the existing logical-to-physical (L2P) mapping of the FTL to key-to-physical (K2P) mapping and implements no-copy SSTable compaction through remapping of KV items [22]. As the FTL is changed, KVSSD can only be used specifically for LSM trees and cannot be applied to other applications. By contrast, RemapCom does not specialize FTL for LSM-tree by just adding a new system call. WiscKey separates keys and values so that it can reduce the overhead of rewriting values of KV items [23]. However, it needs a special garbage collector to reclaim free space in the value log.

VI. CONCLUSION

In this paper, in order to improve compaction performance of LSM-tree based KV stores, we design RemapCom to leverage the data block remapping during compaction. RemapCom identifies Unchanged Data Blocks (UDB) of SSTables through a lightweight state machine, increases the ratio of UDB to fully exploit the benefits of remapping, and designs two primitives to implement block remapping. Results on comprehensive benchmarks have verified its effectiveness in reducing write amplification and optimizing write performance.

References

[1] X. Wu, Y. Xu, Z. Shao, and S. Jiang, “LSM-trie: An LSM-tree-basedUltra-LargeKey-Value store for small data items,” in *USENIX ATC*, pp. 71–82, 2015.

[2] P. O’Neil, E. Cheng, D. Gawlick, and E. O’Neil, “The log-structured merge-tree (lsm-tree),” *Acta Informatica*, vol. 33, pp. 351–385, 1996.

[3] K. Huang, Z. Jia, Z. Shen, Z. Shao, and F. Chen, “Less is more: De-amplifying i/os for key-value stores with a log-assisted lsm-tree,” in *ICDE*, pp. 612–623, IEEE, 2021.

[4] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, “Design tradeoffs for SSD performance,” in *USENIX ATC* (R. Isaacs and Y. Zhou, eds.), pp. 57–70, USENIX Association, 2008.

[5] C. Gao, L. Shi, C. Ji, Y. Di, K. Wu, C. J. Xue, and E. H.-M. Sha, “Exploiting parallelism for access conflict minimization in flash-based solid state drives,” *IEEE TCAD*, vol. 37, no. 1, pp. 168–181, 2017.

[6] Y. Du, S. Huang, Y. Zhou, and Q. Li, “Towards ldpc read performance of 3d flash memories with layer-induced error characteristics,” *ACM TODAES*, vol. 28, no. 3, pp. 1–25, 2023.

[7] Y. Du, Y. Gao, S. Huang, and Q. Li, “Ldpc level prediction towards read performance of high-density flash memories,” *IEEE TCAD*, 2023.

[8] W. Kang, S. Lee, B. Moon, G. Oh, and C. Min, “X-FTL: transactional FTL for sqlite databases,” in *SIGMOD*, pp. 97–108, ACM, 2013.

[9] Q. Wu, Y. Zhou, F. Wu, K. Wang, H. Lv, J. Wan, and C. Xie, “SW-WAL: leveraging address remapping of ssds to achieve single-write write-ahead logging,” in *DATE*, pp. 802–807, IEEE, 2021.

[10] Y. Zhou, Q. Wu, F. Wu, H. Jiang, J. Zhou, and C. Xie, “Remap-ssd: Safely and efficiently exploiting SSD address remapping to eliminate duplicate writes,” in *USENIX FAST*, pp. 187–202, USENIX Association, 2021.

[11] Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson, “Improving ssd lifetime with byte-addressable metadata,” in *MEMSYS*, pp. 374–384, 2017.

[12] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti, “Incremental organization for data recording and warehousing,” in *VLDB*, pp. 16–25, Morgan Kaufmann, 1997.

[13] Z. Cao, S. Dong, S. Vemuri, and D. H. C. Du, “Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook,” in *USENIX FAST*, pp. 209–223, USENIX Association, 2020.

[14] H. Li, M. Hao, M. H. Tong, S. Sundararaman, M. Bjørling, and H. S. Gunawi, “The CASE of FEMU: cheap, accurate, scalable and extensible flash emulator,” in *USENIX FAST*, pp. 83–90, USENIX Association, 2018.

[15] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with YCSB,” in *SoCC*, pp. 143–154, ACM, 2010.

[16] S. Ghemawat and J. Dean, “Leveldb.” https://github.com/google/leveldb, 2016.

[17] N. Dayan, T. Weiss, S. Dashevsky, M. Pan, E. Bortnikov, and M. Twitto, “Spooky: granulating lsm-tree compactions correctly,” *Proceedings of the VLDB Endowment*, vol. 15, no. 11, pp. 3071–3084, 2022.

[18] H. Sun, G. Chen, Y. Yue, and X. Qin, “Improving lsm-tree based key-value stores with fine-grained compaction mechanism,” *IEEE TCC*, vol. 11, no. 4, pp. 3778–3796, 2023.

[19] R. Thonangi and J. Yang, “On log-structured merge for solid-state drives,” in *IEEE ICDE*, pp. 683–694, IEEE Computer Society, 2017.

[20] Y. Hu and Y. Du, “Reducing tail latency of lsm-tree based key-value store via limited compaction,” in *SAC*, pp. 178–181, ACM, 2021.

[21] H. Lee, C. Lee, S. Lee, and Y. Kim, “Compaction-aware zone allocation for LSM based key-value store on ZNS ssds,” in *HotStorage*, pp. 93–99, ACM, 2022.

[22] S. Wu, K. Lin, and L. Chang, “KVSSD: close integration of LSM trees and flash translation layer for write-efficient KV store,” in *DATE* (J. Madsen and A. K. Coskun, eds.), pp. 563–568, IEEE, 2018.

[23] 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 TOS*, vol. 13, no. 1, pp. 1–28, 2017.