KVIMR: Key-Value Store Aware Data Management Middleware for Interlaced Magnetic Recording Based Hard Disk Drive

Yuhong Liang, Tsun-Yu Yang, and Ming-Chang Yang,
The Chinese University of Hong Kong

https://www.usenix.org/conference/atc21/presentation/liang

This paper is included in the Proceedings of the
2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

KVIMR: Key-Value Store Aware Data Management Middleware for Interlaced Magnetic Recording Based Hard Disk Drive

Yuhong Liang, Tsun-Yu Yang, and Ming-Chang Yang
The Chinese University of Hong Kong

Abstract

Log-Structured Merge-Tree (LSM-tree) based key-value (KV) store provides write-intensive applications with high throughput on Hard Disk Drive (HDD). Recently, the emerging Interlaced Magnetic Recording (IMR) technology makes the IMR based HDD become another desirable option to construct a cost-effective KV store because of its high areal density.Nevertheless, we observe that deploying LSM-tree based KV store on IMR based HDD may suffer noticeable degradation on throughput of incoming reads/writes. Thus, this paper presents KVIMR, a data management for constructing a cost-effective yet high-throughput LSM-tree based KV store on IMR based HDD. KVIMR is architected as a *middleware*, interposed between LSM-tree based KV store and IMR based HDD, to embrace the compatibility for mainstream LSM-tree based KV store implementations with limited modifications. Technically, KVIMR adopts a novel *Compaction-aware Track Allocation* scheme, which leverages the special properties behind the compaction process to remedy the throughput degradation. KVIMR further utilizes a novel *Merged RMW* approach to improve the efficiency of persisting a multi-track-sized file of KV store into IMR tracks with the ensured crash consistency. Our evaluations on several well-known LSM-tree based KV store implementations reveal that KVIMR not only improves the overall throughput by up to 1.55× under write-intensive workloads but even achieves 2.17× higher throughput under high space usage of HDD, as compared with the state-of-the-art track allocation scheme for IMR.

1 Introduction

Persistent key-value (KV) stores have gained popularity in diverse data-intensive applications, ranging from electronic commerce [15], internet services [43], to cloud environments [13, 33], because of its high efficiency in insertions, point and range queries, and deletions. Among various implementations of KV store, Log-Structured Merge-Tree (LSM-tree) [38] based KV stores (e.g., BigTable [13], Cassandra [34], LevelDB [21], HBase [26], HyperLevelDB [17] and

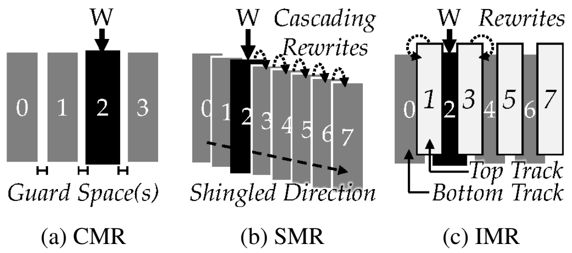


Figure 1: Track Layouts of CMR, SMR, and IMR.

RocksDB [18]) demonstrate its strength in delivering high write throughput on the mechanical hard-disk drive (HDD). The reason is that the LSM-tree takes advantage of the sequential access strength of HDD [38] by first batching the inserted KV pairs in memory and then persisting the batched data from memory into HDD sequentially.

With the explosive growth of data, how to construct a *cost-effective* KV store has become another vital challenge. However, the most common type of HDD, i.e., the *Conventional Magnetic Recording (CMR)* based HDD, has reached its bottleneck in providing higher areal density [11] to lower its *cost-per-GB* because of the super-paramagnetic effect [37]. Among various novel technologies that can break such areal density limitation [12, 16, 28, 30–32, 35, 40, 44, 46, 55], *Shingled Magnetic Recording (SMR)* technology [10, 47] is an eligible alternative of CMR for offering higher areal density gain with limited manufacturing changes [49, 51]. Different from the CMR track layout in which a small *guard space* is introduced to every two physically-adjacent tracks (as shown in Figure 1a), every SMR track is partly overlapped by its subsequent one(s), making that SMR tracks are placed closer to each other (as shown in Figure 1b). However, with this "shingled" track layout, data updates to a single SMR track may incur the time-consuming *track rewrites* over a great amount of subsequent tracks along the shingled direction, so as to avoid losing any valid data [10, 51].Although lots efforts have been made to tackle the track rewrite issue for SMR based HDD at different layers [10, 27, 39, 41, 42, 51–53], *the applicability of SMR based HDD is still limited because of its long write tail-latency* [51].

More recently, the emerging *Interlaced Magnetic Recording (IMR)* technology [28] has been validated for being effective in delivering *even-higher* areal density with *reduced* track rewrite issue than the SMR [22], making the IMR become a more desirable successor to the CMR. In general, the areal density gain of IMR is mainly attributed to the linear density increasing of tracks with different laser currents [22, 23]. As shown in Figure 1c, IMR embraces a new "interlaced" track layout to organize tracks into *top tracks* and *bottom tracks*, with each bottom track overlapped partially by two adjacent top tracks. With this interlaced track layout, any top tracks can be freely updated *without* incurring any track rewrites; whilst updates to a bottom track would only introduce *at most two* track rewrites on its two adjacent top tracks. Moreover, to conduct track rewrites with ensured crash consistency, the *read-modify-write (RMW)* approach [25] is presented.

Inspired by the *ever-high* areal density and the *reduced* track rewrite issue of IMR technology, this paper for the first time, seeks for the possibility in constructing a *cost-effective yet high-throughput* LSM-tree based KV store on IMR based HDD. Nevertheless, we observe that deploying LSM-tree based KV stores on IMR based HDD may still suffer noticeable degradation on throughput of incoming reads/writes. The reason is that the RMW process of IMR based HDD may amplify the background I/Os introduced by the compaction process of LSM-tree based KV store, and such amplified background I/Os may thereby degrade the efficiency of the compaction process and further slow down the throughput of incoming reads/writes [9, 39, 53]. Moreover, none of the existing designs for managing IMR tracks is able to well remedy such throughput degradation, because the special software behavior, hid behind compaction process of the LSM-tree based KV store, is out of their design considerations.

With the goal of alleviating the noticeable throughput degradation caused by the IMR technology, this paper presents KVIMR, a data management middleware for constructing a *cost-effective yet high-throughput* LSM-tree based KV store on IMR based HDD. In particular, KVIMR is architected as a *middleware*, sitting between LSM-tree based KV store and IMR based HDD, in order to facilitate 1) the support for various existing implementations of LSM-tree based KV store with limited modifications and 2) the direct and efficient management on IMR based HDD.

To make KVIMR be aware of the software behaviour of LSM-tree based KV store with limited overhead, the most key semantic information (i.e., the "level" information) regarding the data of LSM-tree is passed to KVIMR along with data writes. Given the "level" information as a clue, KVIMR introduces a novel *Compaction-aware Track Allocation* scheme to allocate tracks for data of LSM-tree KV store according to the "level" information. This scheme effectively remedies the throughput degradation and improves the compaction efficiency by 1) minimizing the time-consuming RMWs when persisting data files (i.e., *SSTables*) of LSM-tree based KV store and 2) efficiently accessing the data files of LSM-tree based KV store during the compaction process. To further improve the compaction efficiency when the time-consuming RMWs are inevitable, KVIMR employs a novel *Merged RMW* approach to efficiently persist a multi-track-sized data file of LSM-tree based KV store into IMR tracks. Its key idea is to re-order the multiple "track-by-track" RMWs into a single "merged" RMW while still nicely ensure the crash consistency. With this approach, KVIMR significantly reduces the number of required `sync`-like functions, which have adverse effects on I/O performance of HDD [2, 24, 45], and avoids redundant track rewrites caused by the track-by-track RMW approach.

KVIMR is developed in `C++` and provides a POSIX complaint interface for the existing LSM-tree KV store implementations to interact with IMR based HDD. Specifically, we modify three well-known LSM-tree KV stores, i.e., RocksDB [18], LevelDB [21], and HyperLevelDB [17], by replacing the native file operations with a similar set of file operations provided by KVIMR to access the data files in an emulated IMR based HDD. Our evaluations on these three LSM-tree based KV store implementations reveal that KVIMR not only improves the overall throughput by up to 1.55× under write-intensive workloads but even achieves 2.17× higher throughput under high space usage of HDD, as compared with the state-of-the-art track allocation scheme for IMR.

The rest of this paper is organized as follows: Section 2 presents the background and motivation regarding this work. Section 3 introduces the design of *KVIMR*. Then, Section 4 provides the implementation details and demonstrates the evaluation results. Finally, Section 5 presents relevant studies and Section 6 concludes this work.

2 Background and Motivation

2.1 LSM-Tree based KV Store

Because of the strength in delivering high write throughput on the mechanical HDD, Log-Structured Merge-Tree (LSM-tree) [38] inspires many well-known key-value (KV) stores, such as RocksDB [18], LevelDB [21], and HyperLevelDB [17]. In general, these LSM-tree based KV stores embrace a similar design concept, borrowed from LSM-tree, on managing KV pairs in the HDD, and support a similar set of KV operations such as `put`, `get` and `delete` operations.

The `put` operation is for inserting KV pairs into the KV store. To deliver high throughput of `put` operations on HDD, LSM-tree based KV store first batches all the inserted KV pairs in an in-memory sorted skiplist namely *Memtable*. When the Memtable exceeds its size limit (e.g., 64 *MB* in RocksDB [18]), LSM-tree based KV store creates a new Memtable to keep accommodating new KV pairs, while the old Memtable is converted to an immutable in-memory sorted skiplist namely *Immutable Memtable*. In the background, an important thread takes over the Immutable Memtable and persists it into HDD as an in-disk sorted string table (*SSTable*). In addition, all the SSTables in the disk are organized into multiples levels (i.e., `L0`∼`Ln`), similar to the multiple in-disk components of the LSM-tree [38], and the size limit of each level increases exponentially by a factor (e.g., 10 in both LevelDB [21] and RocksDB [18]) from `L1` to larger level(s). Moreover, the SSTable converted from the Immutable Memtable is usually placed at `L0` first; then, once the total size of SSTables of any level `Li` exceeds its size limit, the background thread keeps performing the *compaction process* in a cascading way to compact the KV pairs from smaller levels to larger levels, until all levels are within their size limits. Specifically, the compaction process 1) picks one SSTable in `Li`, 2) merges it with all SSTables with overlapped key ranges in `L(i+1)`, 3) creates new SSTable(s) into `L(i+1)`, and finally 4) deletes all stale SSTables from the KV store sooner or later.

Besides of the `put` operation, LSM-tree based KV store also supports the `get` operation to read out the value associated with the given key. Specifically, to find out the latest version of value, LSM-tree based KV store searches the requested key-value pair(s) from Memtable, Immutable Memtable, and SSTables from smaller levels to larger levels in order. Moreover, LSM-tree based KV store also supports the `delete` operation to remove specific KV pair(s) from the KV store.

2.2 Interlaced Magnetic Recording

Interlaced Magnetic Recording (IMR) [28] is a new disk technology that adopts the "interlaced" track layout to increase the areal density by shortening the distance between adjacent tracks. As shown in Figure 1c, tracks of IMR based HDD are organized into *top tracks* and *bottom tracks*, and each bottom track (e.g., Track #2) is overlapped partially by two adjacent top tracks (e.g., Tracks #1 and #3). However, with this interlaced track layout, *writing data into any bottom track will destroy the (valid) data stored in the two adjacent top tracks*. Thus, in order to protect the (valid) data of top tracks against data loss on writing data into bottom tracks, the *read-modify-write (RMW)* approach [25] is introduced; specifically, it ensures the crash consistency by quarantining that the following sequence of "read-modify-write" can be enforced: 1) "Read": Backing up the valid data of the two adjacent top tracks in a *backup region*; 2) "Modify": Writing the updated data properly to the bottom track; and 3) "Write": Re-writing the backed-up valid data back to the adjacent top tracks.

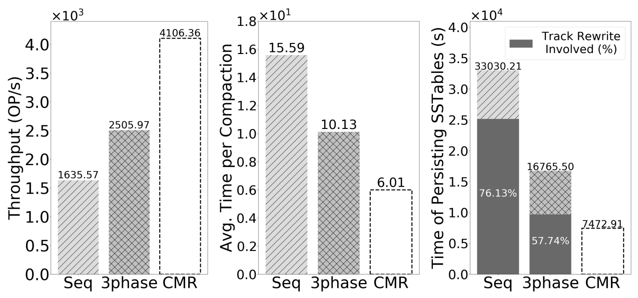
Since the RMW is time-consuming [48], most existing studies for IMR based HDD focus on how to minimize the probability of incurring the RMWs. Specifically, some propose to reduce RMWs via *track allocation* (i.e., how IMR tracks are allocated to accommodate the written data). For example, Gao *et al.* present a *three phase track allocation* to allocate tracks based on three phases of disk space usage [19, 20]: In the first phase, data are placed into *bottom tracks only* until all the bottom tracks are full of data (i.e., 0% ∼ 50% space usage); in the second phase, data are placed into *every other top tracks* until half of the top tracks are used (i.e., 50% ∼ 75% space usage); and in the third phase, data are placed into *the rest of top tracks* (i.e., 75% ∼ 100% space usage). As a result, the first phase would not incur any RMW(s), while the second phase (resp. to the third phase) ensures at most one (resp. to two) top track(s) will be re-written on writing data into any bottom track. Based on the three phase track allocation, Wu *et al.* further propose a *zigzag track allocation* to reverse the track allocation order in the second phase, making every other top tracks be allocated from inner tracks to outer tracks, for better preserving data locality [48, 50].

Another series of studies tries to minimize the probability of incurring the RMWs by considering the *update frequencies of data*. Notably, in the literature, the *frequently-updated data* (resp. to *less-frequently-updated data*) are also referred to as the *hot data* (resp. to *cold data*). For example, Wu *et al.* propose a *top buffer* design to exploit a few top tracks as "write buffer" to place the hot data, and a *block swap* method to exchange the hot data in bottom tracks with the cold data in top tracks, so as to reduce the RMWs caused by updating hot data in bottom tracks [48, 50]. More recently, Hajkazemi *et al.* propose three new track-level techniques, namely *track flipping*, *selective track caching*, and *dynamic track*, to reduce the amount of writes to bottom tracks by moving hot data to top tracks and cold data to bottom ones in different ways [25]. However, please note that, although time-consuming RMWs can be indeed reduced by considering the update frequencies of data, *not all sorts of applications can be benefited from this series of studies*. For example, the data of LSM-tree based KV stores (i.e., SSTable), which is our target application, are written once but *never updated* before being deleted.

2.3 Motivation: Degradation on Throughput and Compaction Efficiency

In order to investigate how serious the performance of LSM-tree based KV store would be degraded by the time-consuming RMW process of IMR technology, we deploy the widely used RocksDB on an 100 *GB*, emulated IMR based HDD. In particular, in order to reasonably allocate IMR tracks for accommodating SSTables, we implement the classical *sequential allocation* scheme [41] (denoted as `Seq`), and the state-of-the-art, IMR based *three-phase allocation* scheme [19, 20, 50] (denoted as `3Phase`). Moreover, to fairly compare and demonstrate that how the performance of IMR based *three-phase allocation* is affected by the RMW process, we also deploy RocksDB on an 100 *GB* CMR based HDD and adopts *three-phase allocation* scheme to allocate CMR tracks for accommodating SSTables (denoted as `CMR`). More details about the implementations can be found in Section 4.1.

Figure 2a shows the performance of randomly loading/inserting 75 millions of 1 *KB* KV pairs, generated by YCSB [14], into RocksDB, where the x-axis denotes different



(a) Throughput (b) Compaction Efficiency (c) Total Time of Persisting SSTables

Figure 2: Performance of Loading 75 Millions of KV Pairs into RocksDB under Different Track Allocation Schemes.

schemes of track allocation and the y-axis shows the throughput (i.e., the number of I/Os per second). It can observed that, although the IMR based `3phase` indeed effectively achieves 1.53 × higher throughput than the classical `Seq`, there still exists a large room for improvement since the IMR based `3phase` also suffers 38.97% degradation on throughput as compared to that of `CMR`. Since it is known that the write throughput of a LSM based KV store could be significantly affected by the compaction process [9, 39, 53], Figure 2b further shows the *compaction efficiency*, which is regarded as the average time required by a compaction process, achieved by different track allocation schemes. It can be observed that, as compared with `CMR`, `Seq` and `3phase` require 2.59× and 1.68× longer time to complete a compaction process on average, and result in the observed the degradation on throughput.

Furthermore, based on our investigation, the observed compaction efficiency degradation of `Seq` and `3phase` can be mainly attributed to the time-consuming RMW process of IMR technology. The reason is that the compaction process involves persisting SSTables into HDD, and the time-consuming RMW process could be thereby frequently triggered during the compaction process. Thus, to explicitly demonstrate how the compaction efficiency is affected by the RMW process, Figure 2c shows the total time of persisting SSTables (during the compaction processes) required by different track allocation schemes and indicates the portion of time spent on rewriting top tracks during the RMW process. It can be observed that, when compared with `CMR`, the total time of persisting SSTables required by `Seq` and `3phase` is noticeably lengthen by around 4.42× and 2.24× respectively. The reasons is that `Seq` and `3phase` entail 25145.76 and 9681.21 seconds on rewriting top tracks during the RMW process, which account for 76.13% and 57.74% of their respective total time of persisting SSTables. Motivated by such observed degradation on throughput and compaction efficiency, a novel middleware KVIMR is proposed in Section 3 to deliver high throughput on IMR based HDD with limited changes to the well-known LSM-tree based KV store implementations.

3 KVIMR

3.1 System Architecture

As shown in Figure 3, KVIMR is architected as a *middleware* between LSM-tree based KV store and IMR based HDD, so as to facilitate 1) the support for various LSM-tree based KV store implementations with limited modifications and 2) the direct and efficient management on IMR based HDD.

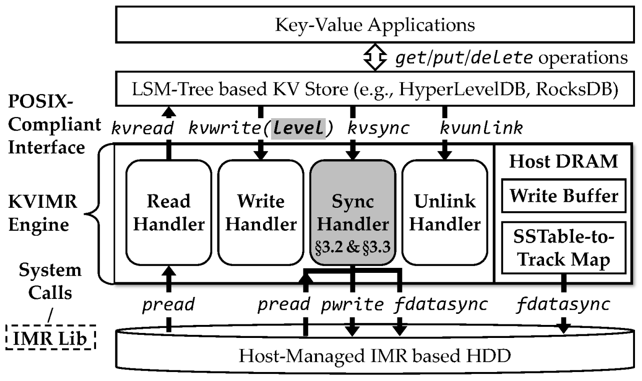


Figure 3: The System Architecture of KVIMR.

To realize this objective, KVIMR provides a POSIX compliant interface [5] so that the upper LSM-tree based KV store can easily access its SSTables through a similar set of common file operations (i.e., `kvread`/`kvwrite`/`kvsync`/`kvunlink` operations) on IMR based HDD with very limited modifications. Besides, to make the KVIMR be aware of the special *compaction behaviour* behind the LSM-tree based KV store, we promote to pass the "level" information of SSTables along with the `kvwrite` file operation on writing SSTables. Notably, since the *level* information is one of key design principles of the LSM-tree, it can be widely found in the mainstream LSM-tree based KV store implementations, such as RocksDB [18], LevelDB [21] and HyperLevelDB [17]. Thus, we argue that *the attempt to pass down the level information will not limit the applicability but instead increase the generality and compatibility of KVIMR*. It is also worth noting that, based on our experience in integrating KVIMR with the three aforementioned LSM-tree based KV store implementations, the modifications of replacing POSIX interface (with file operations provided by KVIMR) and passing level information to KVIMR are just about 100 lines of codes for each KV store implementation.

On the other hand, in our implementation, by employing the existing `pread`/`pwrite`/`fdatasync` system calls with the `O_DIRECT` flag, KVIMR directly manages the data of files (e.g., SSTables) and performs the RMW approach [25] on tracks of the emulated IMR based HDD (please see Section 4.1 for details). Please note that recent researches have demonstrated that the host-managed HDD model and the `libzbc` interface [4] can simplify and facilitate the direct management on SMR based HDD [52, 53]. We envision such host-managed HDD model and library (referred to as

`IMRLib` in Figure 3) will also be available for IMR based HDD with the provision of fundamental functions (such as reading/writing/syncing data and exposing track information), so that a unified application programming interface (API) can be offered for the development of KVIMR. We leave the `IMRLib` extension to KVIMR in the future.

If we take a closer look at Figure 3, the core KVIMR engine maintains an *SSTable-to-Track Map* and allocates a *Write Buffer* in the DRAM space of the host system, and incorporates four *Handlers* to take over the `kvread`/`kvwrite`/`kvsync`/`kvunlink` file operations from the upper LSM-tree based KV store respectively. Their main functionalities and interactions are summarized as follows:

• **SSTable-to-Track Map (or `S2TMap`)** maintains the one-to-many relationship between an SSTable and multiple IMR tracks allocated for it, because the SSTable size in the mainstream LSM-tree based KV store implementations is usually larger than the IMR track size (which is 2 MB [25, 50]). We implement `S2TMap` as `map<sstable id, track list>` and only allocate an IMR track for a single SSTable file for simplicity. Notably, since `S2TMap` is the key metadata of KVIMR, how to maintain and promise the crash consistency of `S2TMap` will be discussed in detail in Section 3.4.

• **Write Handler** is responsible for temporarily buffering the written SSTable in the *Write Buffer* (in the DRAM space) whenever the background thread (i.e., compaction process) of KV store invokes `kvwrite` operations to pass down the SSTable data into middleware, so that the entire SSTable can be later persisted into the IMR based HDD by the *Sync Handler* in one shot. Notably, at runtime, the *Write Buffer* of KVIMR only holds a few SSTables at any given time, since each compaction process/thread only creates one SSTable at a time and will persist it into the storage by the *Sync Handler*. That is, once an SSTable is persisted, KVIMR immediately releases the corresponding DRAM space to keep low demand of *Write Buffer*.

• **Sync Handler** is in charge of persisting the buffered SSTable into IMR tracks (based on different track allocation schemes) whenever the background thread (i.e., compaction process) of KV store needs to ensure the SSTable data are persisted in HDD by invoking `kvsync` operation. In addition, during the process of SSTable persisting, the *Sync Handler* also performs the RMW approach to rewrite tracks (if needed) to ensure the crash consistency of written data, and builds the relationship between the specified SSTable and the allocated tracks in `S2TMap` to facilitate the subsequent accesses to persisted SSTables. Moreover, at runtime, to efficiently determine whether the RMW approach is needed, KVIMR also maintains a bitmap in the DRAM space to keep track of the allocation status of tracks. However, this bitmap does not have to be persisted in HDD since it can be easily rebuilt by scanning `S2TMap`.

• **Read Handler** fetches the requested part of an SSTable from the corresponding track(s) of the IMR based HDD by looking up the `S2TMap` whenever the KV store needs to read the content of SSTable by invoking `kvread` operation (e.g., upon performing the compaction process or servicing `get` operations).

• **Unlink Handler** is to remove an SSTable from HDD and label the corresponding track(s) as *free tracks* (i.e., tracks containing *no* valid data) by resetting the corresponding entries in `S2TMap` whenever the background thread (i.e., compaction process) of KV store invokes `kvunlink` operation to delete an SSTable from HDD.

Among these four handlers, the *Sync Handler* plays the most critical role to affect the throughput of incoming reads/writes, since how SSTables are persisted into tracks of IMR based HDD may largely affect the number of incurred time-consuming RMWs and the efficiency of the background compaction process. Thus, in the following sections, *we shall mainly focus on the design of the Sync Handler*.

3.2 Compaction-aware Track Allocation

Given the *level* information of SSTables as a clue, this section explores how to leverage the special behavior behind compaction process to properly allocate the *two-tier* IMR tracks for accommodating the *multi-level* SSTables. Specifically, Section 3.2.1 first introduces two special properties, namely *compaction frequency* and *compaction locality*, extracted from the compaction process. Then, based on the observed properties and the *level* information of SSTables, Section 3.2.2 introduces a novel *Compaction-aware Track Allocation* scheme to alleviate the throughput degradation by 1) minimizing the time-consuming RMWs and 2) efficiently accessing the SSTables during the compaction process.

3.2.1 Special Properties of Compaction Process

Compaction Frequency. In an LSM-tree based KV store, SSTables of different levels usually have different frequencies of being compacted (i.e., *compaction frequency*) by the compaction process. The reason is twofold: First, as introduced in Section 2.1, the compaction process usually takes place in a cascading way, from smaller levels to larger levels; second, the size limits of different levels increase exponentially along with the levels. As a result, the SSTables of *smaller* levels are more likely to be compacted frequently than that of *larger* levels [29, 54]. In other words, *the lifespan of SSTables of larger levels tends to be longer than that of smaller levels*.

Compaction Locality. Based on our investigation, there exists a very special access locality, namely *compaction locality*, among *different rounds* of compaction process in LSM-tree based KV store. Specifically, as introduced in Section 2.1, a single round of compaction process merges the victim SSTable and the existing SSTables with overlapped key ranges into new SSTables with *close* key ranges. Moreover, since these newly generated SSTables are composed of KV pairs of *close* key ranges, they (or part of them) are more likely to be involved (specifically, `read` and `unlink`) in the latter round(s) of compaction process as compared to other SSTables with *far* key ranges. That is, there exists a special access tendency that *the SSTables generated by a round of compaction process are likely to be involved in the latter round(s) of compaction process*, and we refer it as *compaction locality*.

3.2.2 Design of Compaction-aware Track Allocation

Based on the two observed special properties, this section presents the design of the *Compaction-aware Track Allocation* scheme, which comprises two major steps, namely *Step 1) Level based Bi-Tiering* and *Step 2) Relaxed-Sequential Track Allocation*, in allocating tracks for an SSTable.

Step 1) Level based Bi-Tiering The first step determines *which tier of tracks (i.e., either top tier of tracks or bottom tier of tracks)* should be used to accommodate an SSTable based on its *level* information.

Based on the property of *compaction frequency*, the lifespan of SSTables of larger levels tends to be longer than that of smaller levels. Such property inspires us that, if bottom tracks can be allocated to accommodate SSTables with larger levels (rather than smaller levels), the bottom tracks will be occupied by SSTables with longer lifespan and will not be frequently re-allocated by other SSTables with shorter lifespan. That is, the key idea of the first step is to *allocate bottom tracks to accommodate SSTables of larger levels, so that the probability of incurring the RMWs on allocating bottom tracks could be thereby minimized.*

As shown in Figure 4, in our implementation, we propose to *allocate the bottom tracks only for SSTables of the "current largest" level (e.g., `L5`), while allocate the top tracks for SSTables of the non-largest levels (e.g., `L0`∼`L4`)*. This is because, given that the total capacity of bottom tracks and top tracks in IMR based HDD are roughly the same [19,20] while the size limits among levels in LSM-tree based KV store increase exponentially by a factor (which is usually larger than 2 [17,18,21,36]), the bottom tracks would be fully allocated by SSTables of the largest level when the LSM-tree grows.

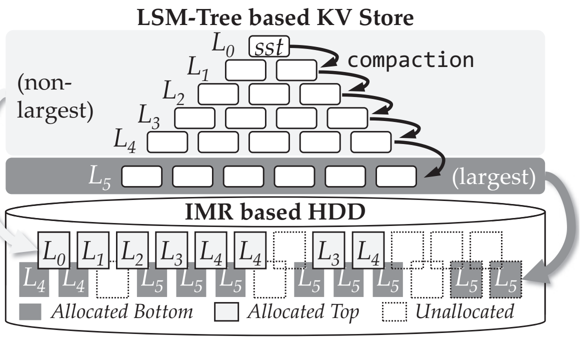


Figure 4: The Level based Bi-Tiering (Step 1).

Moreover, as astute readers may have noticed, when the LSM-tree keeps growing, the size limit of the largest level may become larger than the total capacity constituted by all bottom tracks; as a result, accommodating all the SSTables of the largest level in bottom tracks may become impossible. However, we argue that *this situation will not contradict the design concept of the first step*. The reason is that, although we have no choice to accommodate some SSTables of the largest level in top tracks, all the bottom tracks can still be allocated only by SSTables of the largest level, so that the probability of incurring the RMWs on allocating bottom tracks can be still effectively minimized.

Step 2) Relaxed-Sequential Track Allocation The second step further determines *which tracks in the specific tier* (that is decided by the *Level based Bi-Tiering* (i.e., Step 1)) should be allocated to accommodating an SSTable.

The design principle of the second step is inspired by the following two observations. First, according to the property of *compaction locality*, the SSTables generated by a round of compaction process are likely to be involved in the latter round(s) of compaction process. Second, since IMR based HDD adopts the similar rotational disk mechanisms of CMR based HDD, accessing data sequentially is also much efficient than accessing it randomly in IMR based HDD. Thus, we argue that *it may be beneficial to accommodate the SSTables generated by a round of compaction process into tracks "as sequential as possible"*. The reason is that the SSTables generated by a round of compaction process can be sequentially written into tracks with *high sequential write performance*, and also be sequentially compacted by the latter round(s) of compaction process with *high sequential read performance*.

However, in order to comply with the proposed *Level based Bi-Tiering* (i.e., Step 1) and avoid any data migrations or RMWs, a *Relaxed-Sequential Track Allocation* is instead adopted to *relax* the degree of sequentiality on track allocation from two aspects: First, since all the SSTables generated in a round of compaction process must belong to *same* level, the proposed *Level based Bi-Tiering* would suggest accommodating all of these SSTables in the *same* tier of tracks (i.e., either bottom tracks or top tracks). Thus, instead of allocating the tracks in the *strictly-sequential* order (i.e., one bottom track followed by the adjacent top track), we propose to allocate bottom tracks and top tracks *separately* to accommodate SSTables in the *relaxed-sequential* order (i.e., one bottom track followed by the adjacent bottom track, or one top track followed by the adjacent top track). Second, when allocating tracks in the relaxed-sequential order, we propose to further *relax* the degree of sequentiality by "skipping" the track(s) which are currently allocated by other SSTables or may incur the time-consuming RMWs if possible. However, if none of the unallocated/free bottom tracks can be allocated without incurring RMWs, we propose to allocate the nearest unallocated bottom track to best preserve the degree of sequentiality.

3.3 Merged Read-Modify-Write

To further improve the compaction efficiency when the time-consuming RMWs are inevitable, this section explores how to improve the efficiency of persisting the buffered SSTable from the *Write Buffer* into the allocated tracks. Specifically, Section 3.3.1 first introduces the process of persisting the buffered SSTable when the *Naïve RMW* approach [25] is employed. Then, Section 3.3.2 reveals the potential inefficiency of the *Naïve RMW* approach and introduces a novel *Merged RMW* approach which re-orders multiple RMWs into a "merged RMW" with the improved persisting efficiency and the ensured crash consistency.

3.3.1 SSTable Persisting with Naïve RMW

Algorithm 1 shows the process of *SSTable Persisting with Naïve RMW*, which persists the buffered SSTable (denoted as `SST`) from the *Write Buffer* into the allocated track(s) (denoted as `TrackList`) when the *Naïve RMW* approach [25] is employed. Notably, the notations `SST[i]` and `TrackList[i]` denote that the i^{th} track-size content of the `SST` should be persisted into the i^{th} allocated track in `TrackList`.

First of all, the content of the buffered SSTable are persisted into the allocated tracks on a *track-by-track* basis (i.e., *at the granularity of a track* [25]) (Lines 1∼14). Specifically, if track rewrite(s) are incurred when writing `SST[i]` into `Tracklist[i]`, the *Naïve RMW* is performed as follows (Lines 4∼ 12): First, it performs back-up (Lines 5∼6), followed by invoking a `sync`-like function to ensure the valid data of the adjacent top track(s) are safely persisted into *BackupRegion* (Line 7); Second, it writes the content of `SST[i]` into the `Tracklist[i]` (Line 8), followed by invoking a `sync`-like function to ensure `SST[i]` is safely persisted into the IMR based HDD (Line 9); Thirdly, it performs move-back (Lines 10∼11), followed by invoking a `sync`-like function to ensure the backed-up valid data are safely persisted into adjacent top tracks (Line 12). Otherwise, if track rewrite(s) are not incurred when writing `SST[i]` into `Tracklist[i]`, `SST[i]` are directly written into the `Tracklist[i]` (Lines 13∼14). Finally, a `sync`-like function is invoked to ensure that the entire `SST` is safely persisted into the IMR based HDD (Line 15).

Please be noted, in general, the `sync`-like function (e.g., `sync` [2, 7], `fdatasync` [3], `flush` [4]) is used to ensure that the data previously written by the `write`-like function(s) can be safely persisted into the storage device such as HDD. That is, the `sync`-like functions invoked during the naïve RMWs (i.e., Lines 7, 9 and 12) are actually served as critical *write barriers* to ensure that the correct sequence of "read-modify-write", as described in [25], can be enforced. This also explains why for most of cases, there is no need to invoke a `sync`-like function after writing the `SST[i]` into `Tracklist[i]` which does not incur any track rewrite(s) (i.e., Lines 13∼14). However, if there exist un-synced track(s) which are adjacent to the to-be-written `TrackList[i]`, a `sync`-like function may

Algorithm 1: SSTable Persisting with Naïve RMW

```
   Input: SST: the content of an SSTable to be persisted.
   Input: TrackList: the list of the allocated tracks for SST.
1  for i ← 0 to TrackList.size do
2      if any adjacent tracks of TrackList[i] is unsynced then
3          sync; // ensure valid data are persisted
4      if writing to TrackList[i] incurs top track rewrite(s) then
           /* -- Naïve RMW Begin -- */
5          read the valid data from adjacent top track(s);
6          write the valid data into BackupRegion;
7          sync; // ensure valid data are persisted
8          write SST[i] to TrackList[i];
9          sync;      // ensure SST[i] is persisted
10         read the valid data from BackupRegion;
11         write back the valid data into adjacent top track(s);
12         sync; // ensure valid data are persisted
           /* -- Naïve RMW End -- */
13     else
14         write SST[i] to TrackList[i];
15 sync;      // ensure the entire SST is persisted
```

be invoked to ensure the valid data of un-synced tracks are persisted into HDD by a correct sequence (Lines 2∼3).

3.3.2 SSTable Persisting with Merged RMW

Although the process of *SSTable Persisting with Naïve RMW* (i.e., Algorithm 1) looks simple and elegant, nevertheless, based on our investigation, there exist two sources of inefficiency hid behind it when the size of an SSTable is (much) larger than the size of an IMR track. First, although the `sync`-like functions ensure the *Naïve RMW* approach against unexpected crashes, they also bring severe performance degradation to the whole process. The rationale behind this is that the `sync`-like functions have adverse effects on I/O performance of HDD [2, 24, 45]. Second, since *Naïve RMW* approach must be performed on a *track-by-track* basis, the valid data of some top tracks are actually backed up and written back redundantly, resulting in the doubled `read`, `write` and `sync` workloads.

Thus, in order to further improve the efficiency of persisting the buffered SSTable from the *Write Buffer* into the allocated track(s) when the RMWs are incurred, this section introduces a novel *Merged RMW* approach. Its key idea is to *re-order* multiple *track-by-track* naïve RMWs into a single "merged RMW", so as to significantly reduce the number of required `sync`-like functions and avoid redundant track rewrites while still ensure the crash consistency.

Algorithm 2 depicts the process of *SSTable Persisting with Merged RMW*, which persists the buffered SSTable (denoted as `SST`) from the *Write Buffer* into the allocated track(s) (denoted as `TrackList`) by employing the newly proposed *Merged RMW* approach. First of all, different from the process shown in Algorithm 1, this process firstly performs the

Algorithm 2: SSTable Persisting with Merged RMW

```
   Input: SST: the content of an SSTable to be persisted.
   Input: TrackList: the list of the allocated tracks for SST.
   // The parts of SST incurring track rewrites
1  if persisting SST incurs track rewrites then
       /* -- Merged RMW Begin -- */
2      for i ← 0 to TrackList.size do
3          if writing to TrackList[i] incurs track rewrites then
4              read the valid data from adjacent top track(s);
5              write the valid data to BackupRegion;
6      sync;     // ensure valid data are persisted
7      for i ← 0 to TrackList.size do
8          if writing to TrackList[i] incurs track rewrites then
9              write SST[i] to TrackList[i];
10     sync;          // ensure SST[i] is persisted
11     for i ← 0 to TrackList.size do
12         if writing to TrackList[i] incurs track rewrites then
13             read the valid data from BackupRegion;
14             write back the valid data into adjacent top
                 track(s);
15     sync;     // ensure valid data are persisted
       /* -- Merged RMW End -- */
   // The rest parts of SST w/o track rewrites
16 for i ← 0 to TrackList.size do
17     if SST[i] is un-written && TrackList[i] is a bottom then
18         write SST[i] to TrackList[i];
19 sync;     // ensure data to bottom are persisted
20 for i ← 0 to TrackList.size do
21     if SST[i] is un-written && TrackList[i] is a top then
22         write SST[i] to TrackList[i];
23 sync;     // ensure the entire SST is persisted
```

Merged RMW approach to handle all the allocated bottom track(s) that would incur top track rewrite(s) *on a batch basis* as follows (Lines 1∼15): First, it backs up the valid data of all the involved adjacent top track(s) into the *BackupRegion* in a batch (Lines 2∼5) before invoking the *first* `sync`-like function to ensure all of these valid data are safely persisted into the IMR based HDD (Line 6); Second, it writes the content of `SST` into all the involved bottom tracks in a batch (Lines 7∼9) before invoking the *second* `sync`-like function to ensure these written parts of `SST` are safely persisted into the IMR based HDD (Line 10); Thirdly, it moves back all the backed-up valid data from the *BackupRegion* into the involved adjacent top track(s) in a batch (Lines 11∼14) before invoking the *third* `sync`-like function to ensure these backed-up valid data are safely persisted into the IMR based HDD (Line 15).

Next, this process persists the remaining parts of `SST`, which would not incur top track rewrite(s), into the corresponding tracks (Lines 16∼23) as follows: First, this process gives the highest priority to write the un-written parts of `SST` into their corresponding bottom tracks in a batch (Lines 16∼18). This is because, if the un-written parts of `SST` are written into top tracks first, the subsequent writes to adjacent bottom tracks may instead incur extra track rewrites over those "just-written" top tracks. Then, a `sync`-like function should be invoked (Line 19) to ensure the data writes to bottom tracks (if any) are persisted into HDD. Finally, this process writes the un-written parts of `SST` into their corresponding top tracks in a batch, followed by invoking a `sync`-like function to ensure the entire `SST` are safely persisted into HDD (Lines 20∼23).

Compared with the SSTable persisting process with *Naïve RMW* approach (i.e, Algorithm 1), the SSTable persisting process with the *Merged RMW* approach not only significantly minimizes the number of required `sync`-like functions (specifically, at least 1 and at most 5), but also avoids the redundant backup and write-back of top tracks (since the involved top tracks are only backed up and written back once). Moreover, the *Merged RMW* approach nicely ensures the crash consistency for the SSTable by backing up the parts of SSTable residing in top tracks into the *BackupRegion*, before writing any data into bottom tracks that would incur top track rewrites. However, it is worth noting that the *Merged RMW* approach requires a larger *BackupRegion* than the *Naïve RMW* approach, in order to back up the valid data of all the involved adjacent top track(s) in a batch. Specifically, the size of the *BackupRegion* required by the *Merged RMW* approach is just twice as the size of an SSTable, which should be a negligible storage overhead compared with the total capacity of HDD.

Notably, as astute readers may have noticed, an alternative approach, which organizes all tracks into "regions" of a fixed number of consecutive tracks, is also able to reduce the numbers of track rewrites and `sync`-like functions as does KVIMR. That is, this approach may always write a whole SSTable into a fixed-sized region by writing all the bottom tracks then all the top tracks in the region. Although such approach looks simple and effective indeed, however, we argue that it might inevitably waste the disk space (since the actual sizes of SSTables are not fixed and could be smaller than the fixed SSTable size limit in the mainstream LSM-tree based KV store implementations). Thus, KVIMR adopts a more flexible *Compaction-aware Track Allocation* and *Merged RMW* to avoid such space waste issue while reduce the numbers of track rewrites and `sync`-like functions.

3.4 Crash Consistency

Since the proposed KVIMR does not introduce significant changes to the core design of LSM-tree based KV stores, the crash consistency mechanisms of different LSM-tree based KV stores still remain and work nicely in the same way.

KVIMR further ensures the crash consistency regarding its key metadata (i.e., SSTable-to-Track Map (`S2TMap`)) as follows: First, to avoid incurring the time-consuming RMW processes over IMR tracks, we propose to persist the track-level `S2TMap` into a few reserved top tracks of IMR based HDD or other persistent storage device, which can be freely updated, in the system. Second, to better manage the persisting overhead, we propose to persist `S2TMap` whenever the LSM-tree based KV store persists their key metadata (e.g., the *manifest* file) after each successful compaction process. This ensures that the metadata of KVIMR can always be consistent with that of the LSM-tree based KV store, so that both LSM-tree based KV store and KVIMR can be recovered to a consistent state successfully after unexpected system crashes.

4 Evaluation

4.1 Evaluation Setup

In this section, we evaluate the effectiveness of the proposed KVIMR middleware. Since there is no available real products of IMR based HDDs to date, we follow the emulation approach suggested in [39] to emulate an 100 *GB* IMR based HDD with a real CMR based HDD (model no. ST500DM002 [1]) so that the evaluated results can accurately reflect actual performance of the disk internal activities. Particularly, we split the *LBA* range into 2 *MB* tracks as suggested in [25, 50], organize the tracks by the interlaced track layout of IMR, and read/write to tracks with IMR-like restrictions [28]. In addition, we architect the KVIMR as a middleware as introduced in Section 3.1 and develop KVIMR in the user-space using `C++`, and employ the existing `pread`/`pwrite`/`fdatasync` system calls with the `O_DIRECT` flag to directly manage SSTables and perform the RMW approach over tracks of the emulated IMR based HDD. Moreover, the following schemes are implemented and integrated with the KVIMR middleware for evaluation purposes:

- **`Seq`** allocates all tracks in a sequential order (i.e., one track followed by the subsequent one(s)) but skips tracks that have been allocated by other SSTables; and it adopts the *Naïve RMW* approach [25] to perform track rewrites.
- **`3Phase`** allocates the unused tracks in the following order: bottom tracks, every other top tracks, and the rest of top tracks [19, 20]; and it adopts the *Naïve RMW* approach [25] to perform track rewrites.
- **`KVIMR-N`** denotes the proposed *Compaction Aware Track Allocation* scheme (presented in Section 3.2) with the *Naïve RMW* approach [25].
- **`KVIMR-M`** denotes the proposed *Compaction Aware Track Allocation* scheme (presented in Section 3.2) with the proposed *Merged RMW* approach (presented in Section 3.3).

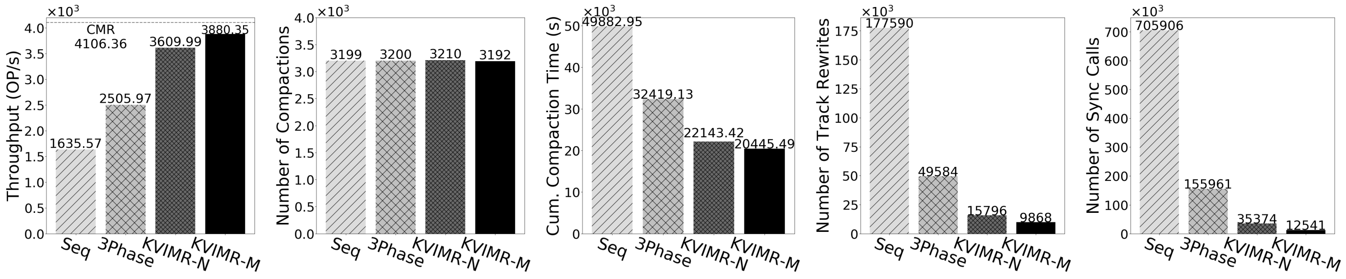
On the other hand, we modify three well-known implementations of LSM-tree based KV store, i.e., RocksDB [18], LevelDB [21], and HyperLevelDB [17], so that they can access its SSTables through a set of POSXI complaint file operations provided by the KVIMR middleware as presented in Section 3.1. The metadata (e.g., manifest, WAL) of these LSM-tree based KV store implementations are currently kept in an SSD alongside (like GearDB's implementation [53]). However, since the size of metadata is relatively small, these metadata can also be managed in a few IMR tracks with little performance impact. Notably, because of limited page length, we mainly demonstrate the evaluation results collected from RocksDB. In addition, we adopt the default settings [18] (e.g., the SSTable size is 64 *MB*, the size limit of level 1 is 256 *MB*, *kL0_SlowdownWritesTrigger* and *kL0_StopWritesTrigger* are 20 and 36 respectively) to configure RocksDB, but further set the *compaction_readahead_size* to 2 *MB* (as suggested in [6]) and *bloom_bits* to 10 (as suggested in [6]) to have better performance on HDD. Moreover, we revise the benchmark tool `db_bench` released with RocksDB, LevelDB, and HyperLevelDB, so as to evaluate the performance of different schemes under various workloads generated by YCSB [14]. All the experiments are conducted on a workstation PC, which is equipped with two Intel(R) Xeon(R) E5-1630 v4 @ 3.70 GHz processors and 16 GB DDR4 DIMM memory, where the operating system is 64-bit Ubuntu 14.04.1 LTS with Linux kernel version 3.13.0.

4.2 Evaluation Results

4.2.1 Overall Load Performance

This section investigates the overall performance under different schemes by randomly loading/inserting 75 millions of 1 *KB* KV pairs into RocksDB. Figure 5 shows the overall performance results, where the x-axis of each sub-figure denotes different schemes and the y-axis of each sub-figure shows the results from different performance metrics. For ease of the comparison, the throughput of adopting three-phase allocation scheme to allocate CMR tracks for accommodating SSTables (denoted as `CMR` as in Section 2.3) is also plotted in Figure 5 as a horizontal dashed line. From Figure 5a, we can firstly observe that both `KVIMR-N` and `KVIMR-M` achieve significant throughput improvements when compared with `Seq` and `3Phase`. Specifically, `KVIMR-N` achieves 1.44× and 2.21× higher throughput than that of `Seq` and `3Phase`. Moreover, it can be also observed that `KVIMR-M` further improves the throughput by 7.0% as compared to `KVIMR-N`, because of the improved compaction efficiency contributed by the proposed Merged RMW approach. More encouragingly, `KVIMR-M` achieves a comparable throughput as compared to `CMR` with only about 5.5% degradation.

To better demonstrate the compaction efficiency of different schemes, Figures 5b and 5c respectively demonstrate the number of compactions and the sum of execution time of compactions (or *cumulative compaction time* [18]) during the loading process. It can be clearly observed from Figure 5b that all the schemes share similar number of compactions. This is because all the schemes are implemented as middleware (similar to the proposed `KVIMR`), and the design regarding how RocksDB performs compaction processes



(a) Throughput (b) #Compactions (c) Cum. Compaction Time (d) #Top Track Rewrites (e) #Sync Calls

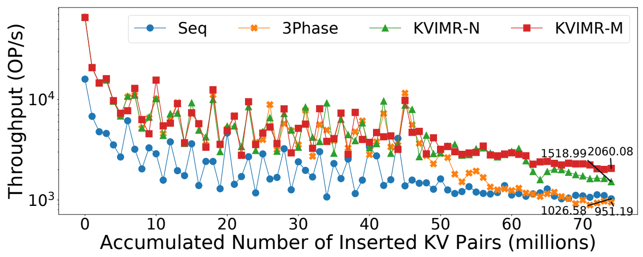
Figure 5: Overall Performance Results of Loading 75 Millions of KV Pairs into RocksDB.

is not changed. However, from Figure 5c, we can observe that, `KVIMR-N` and `KVIMR-M` effectively reduce the cumulative compaction time required to complete a similar numbers of compactions, and thereby greatly improve the compaction efficiency (i.e., the average time required to perform a compaction process). Specifically, `KVIMR-N` largely reduces the cumulative compaction time by 55.61% and 31.69% as compared to `Seq` and `3Phase`. Moreover, `KVIMR-M` further reduces the cumulative compaction execution time by 7.67% as compared to `KVIMR-N`.

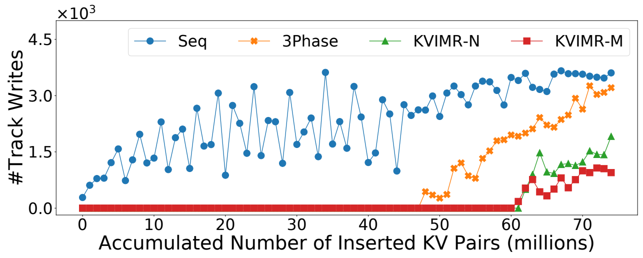
Based on our investigation, the improvement on the overall throughput and the reduction on the cumulative compaction time (achieved by `KVIMR-N` and `KVIMR-M`) can be mainly attributed to the prevention of time-consuming RMW processes, which may incur additional track rewrites and sync calls to slow down the efficiency of persisting SSTables. Figures 5d and 5e reveal the numbers of (top) track rewrites and the numbers of sync calls incurred by different schemes during the loading process. It can be firstly observed that, `KVIMR-N` and `KVIMR-M` significantly reduce the numbers of top track rewrites and sync calls as compared to `Seq` and `3phase`. Specifically, `KVIMR-N` decreases the number of top track rewrites by 91.11% and 68.14% and diminishes the number of sync calls by 94.99% and 77.32% as compared to `Seq` and `3phase`, respectively. The main reason behind such reductions is that, `KVIMR-N` leverages the *compaction frequency* to allocate the IMR tracks for accommodating SSTables, so as to minimize the occurrence of time-consuming RMW processes. It is also worthy to see that, compared to `KVIMR-N`, `KVIMR-M` further reduces the numbers of top track rewrites and sync calls by 37.53% and 64.55%. The is because `KVIMR-M` adopts the proposed Merged RMW to cleverly circumvent the redundant top track rewrites and bound the number of sync calls, when the time-consuming RMWs cannot be fully avoided by the proposed Compaction-aware Track Allocation.

4.2.2 Load Performance Changes

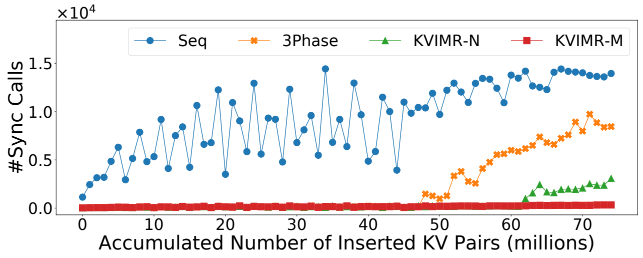
Besides the overall performance results presented in Figure 5, it is also valuable to investigate how the throughput and other performance metrics actually change along the whole loading process. Figures 6a, 6b and 6c respectively show the throughput, the number of incurred track rewrites, and the number of invoked sync calls for loading every 1 million of KV pairs, where the x-axis of each sub-figure denotes the accumulated number of inserted/loaded KV pairs.



(a) Changes in Throughput during Loading.



(b) Changes in the Number of Incurred Track Rewrites.



(c) Changes in the Number of Invoked Sync Calls.

Figure 6: Performance Changes during Loading.

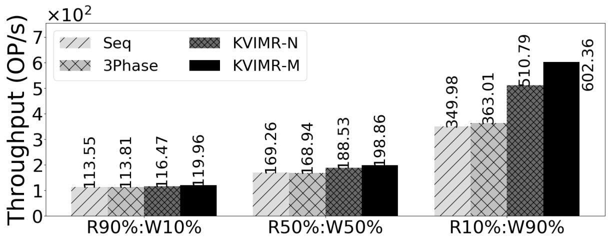
First of all, from Figure 6a, we can observe that, compared with `Seq` and `3Phase`, `KVIMR-N` and `KVIMR-M` lead to the much higher throughputs, almost for every 1 million of inserted KV pairs along the whole loading process. The main reason behind this is that, as revealed by Figures 6b and 6c, `KVIMR-N` and `KVIMR-M` incur much less number of top track rewrites and sync calls during the whole loading process. Interestingly, after inserting about 60 millions of KV pairs, `KVIMR-M` starts to achieve the highest throughput than the rest of schemes. For example, `KVIMR-M` achieves 26.27% higher throughput than that of `KVIMR-N` for the last 1 million of inserted KV pairs. The reason is that, when the RMW processes are inevitable during the late loading process, the proposed Merged RMW approach can cleverly avoid redundant top track rewrites and bound the number of sync calls, resulting in the improved efficiency of persisting SSTables.

By contrast, `Seq` achieves the lowest throughput almost during the whole loading process, and `3Phase` starts to encounter noticeable throughput degradation after inserting about 50 millions of KV pairs and suffers very low throughput as `Seq` at the very end of the loading process. In particular, as observed from Figure 6a, `KVIMR-M` achieves 2.01× and 2.17× higher throughput than that of `Seq` and `3Phase` respectively for the last 1 million of inserted KV pairs. Such serious throughput degradations of `Seq` and `3Phase` can be attributed to their track allocation designs. Particularly, `Seq` writes the SSTable data in a strictly-sequential order (i.e., one track followed by the subsequent one(s)), resulting in that the time-consuming RMW processes are mostly incurred whenever writing to any bottom track(s). On the other hand, `3Phase` only utilizes bottom tracks to accommodate the SSTable data and fully avoids RMW processes during the first phase (i.e., before inserting 50 millions of KV pairs). However, when `3Phase` starts to allocate top tracks to accommodate SSTables afterwards, it may incur more and more RMW processes as the space usage of HDD increases due to the lack of consideration regarding compaction frequency.

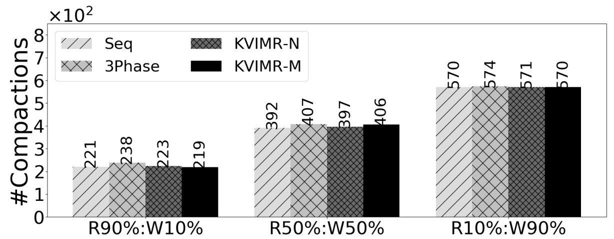
4.2.3 Throughput under Different Workloads

To further investigate the throughput under read-write mixed workloads, we randomly insert and retrieve 7.5 millions of KV pairs after loading the 75 millions of KV pairs into RocksDB. In particular, three workloads of different read/write (R/W) ratios (i.e., R90%:W10%, R50%:W50%, and R10%:W90%) generated by YCSB [14], which follow the Zipfian distribution, are used to represent different *write intensities*. Figure 7a shows the throughputs of different schemes, where the x-axis denotes different R/W ratios and the y-axis shows the throughput. It can be firstly observed that the throughputs of all the evaluated schemes increase as the write intensity (or write ratio) increases. This is because, randomly reading a large amount of KV pairs from LSM-tree based KV store might seriously slow down the overall throughput due to the poor random access performance of HDD.

More importantly, as revealed by Figure 7a, `KVIMR-N` and `KVIMR-M` are more effective in improving the throughput as the write intensity keeps increasing, as compared to `Seq` and `3Phase`. Specifically, `KVIMR-N` and `KVIMR-M` lead to at least 2.28%, 10.22% and 28.93% throughput improvements (than that of `Seq` and `3Phase`) under workloads of R90%:W10%, R50%:W50% and R10%:W90% respectively. This is because, as shown in Figure 7b, for all the evaluated schemes, the number of incurred compactions increases as the write intensity increases. Such increasing number of compactions implies that more SSTables need to be persistted into HDD, leaving a



(a) Throughput under Workloads of Various R/W Ratios.



(b) #Compactions under Workloads of Various R/W Ratios.

Figure 7: Throughput under Workloads of Various R/W Ratios

larger room for `KVIMR-N` and `KVIMR-M` to reduce the number of track rewrites and sync calls for higher throughput gains (under workloads of higher write intensities).

4.2.4 Throughput under Different Sizes of SSTable

To understand how the impact of SSTable size on the throughput, we randomly loading/inserting 75 millions of 1 KB KV pairs into RocksDB under different sizes of SSTable. Specifically, as suggested in the [8], since HDD typically demonstrates better performance under larger sizes of SSTable, this section mainly evaluates the throughput of different schemes when SSTable size ranges from 64MB to 512MB.

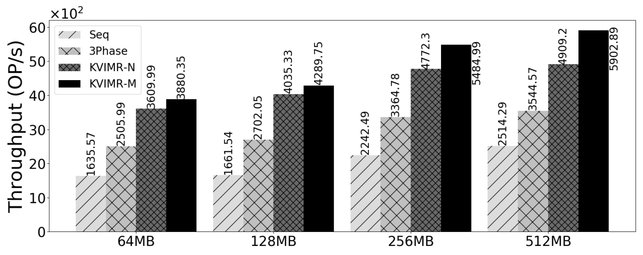
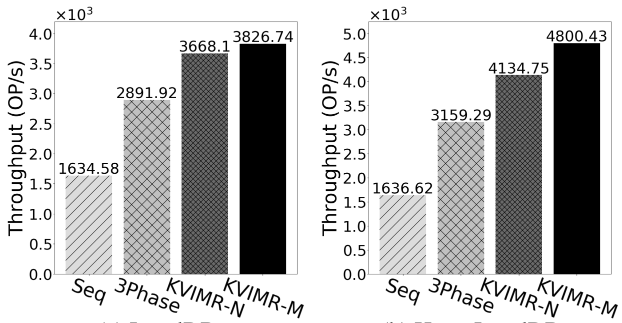


Figure 8: Throughput under Different Sizes of SSTable.

Figure 8 shows the overall load performance results, where the x-axis denotes different SSTable sizes and the y-axis denotes the throughput. It can be firstly observed that all the schemes achieve higher throughput with larger SSTable sizes. Besides, `KVIMR` still effectively and stably achieves better throughput than that of `Seq` and `3Phase` as the SSTable size increases. More interestingly, as compared to `KVIMR-N`, `KVIMR-M` tends to be more effective in improving the throughput when the SSTable size increases. Specifically, the performance gaps between `KVIMR-N` and `KVIMR-M` are 7.0%, 5.9%, 13.0%, 16.8% with SSTables sizes of 64 MB, 128 MB, 256 MB and 512 MB respectively. The key reason of such increasing performance gap is that the Merged RMW adopted by `KVIMR-M` has potential to circumvent more redundant top track rewrites and reduce more number of sync calls with larger SSTable sizes.

4.2.5 Support for Other LSM-tree based KV Stores

To demonstrate the great compatibility of KVIMR, we further conduct the load throughput evaluation (i.e., loading 75 millions of 1 *KB* KV pairs generated by YCSB [14]) on other two well-known LSM-tree based KV stores: LevelDB [21], and HyperLevelDB [17]. Notably, to present the performance results on the same basis, we apply the same configurations adopted by RocksDB to both LevelDB and HyperLevelDB for achieving better HDD performance. That is, we set the SSTable size to 64 *MB*, set the size limit of level 1 to 256 *MB* and set *kL0_SlowdownWritesTrigger* and *kL0_StopWritesTrigger* to 20 and 36 respectively. Moreover, since LevelDB and HyperLevelDB lack the design of *compaction_readahead_size*, we instead set the *block_size* to 2 *MB* to have better performance on HDD as suggested by [6].



(a) LevelDB (b) HyperLevelDB

Figure 9: Throughput of Loading 75 Millions of KV Pairs into Other Well-Known LSM-tree based KV Stores.

Figures 9a and 9b show the overall throughput achieved by LevelDB and HyperLevelDB respectively upon loading 75 millions of 1 *KB* KV pairs. It can be clearly observed that the trends of overall throughputs achieved by the evaluated schemes under LevelDB and HyperLevelDB are quite similar to what we can observe under RocksDB (see Figure 5a). That is, `KVIMR-N` and `KVIMR-M` still effectively achieve better throughputs than that of `Seq` and `3Phase` by at least 55.43% and 21.16% under LevelDB respectively (and by at least 60.41% and 23.59% under HyperLevelDB respectively).

5 Related Work

There exist some mentionable studies proposed to address the track rewrite issue of Shingled Magnetic Recording (SMR) based HDD by introducing the following techniques to the existing LSM-tree based KV store implementations. For example, SMRDB [39] proposes to enlarge the SSTable size to the band/zone size of SMR based HDD to avoid track rewrites, arranges SSTables into only two levels with the key ranges of SSTables overlapped within the same level, and presents a new cost-considered compaction design. Another study called SEALDB [52] proposes to group the SSTables involved in a compaction into a set, and then sequentially writes a set of SSTables into a variable-sized *dynamic band* to mitigate the track rewrite issue. A more recent study namely GearDB [53] proposes to sequentially write the SSTables of the same level into the same *SMR band/zone*, and further introduces a *Gear Compaction* to avoid garbage collection in SMR based HDD.

The aforementioned designs indeed take effect at mitigating the track rewrite issue of SMR; however, they could be *ineffective* when blindly applied to IMR based HDD since SMR and IMR technologies adopt naturally-different track layouts. Specifically, all these designs must write the SSTables into SMR tracks in a "strictly-sequential" order (i.e., one track followed by the subsequent one(s)). As revealed by our evaluation, sequentially writing SSTables into IMR tracks may incur serious throughput degradation, since writing to any bottom track may cause the time-consuming RMW(s) to rewrite its adjacent top track(s) in the IMR based HDD. On the other hand, all these designs are developed by revamping the existing LSM-tree based KV store (i.e., LevelDB [21]). That is, they are all tightly-coupled with one specific implementation of the LSM-tree based KV store, making them costly to be upgraded with the software updates of KV store and integrated with other representative implementations of the LSM-tree based KV store, such as the widely deployed RocksDB [18].

6 Conclusion

This paper presents KVIMR, a data management middleware, to construct a cost-effective yet high-throughput LSM-tree based KV store on IMR based HDD. KVIMR delivers great compatibility for mainstream LSM-tree based KV store implementations without introducing significant modifications by being architected as a middleware sitting between LSM-tree based KV store and IMR based HDD. Technically, KVIMR remedies the throughput degradation resulted from IMR through the proposing of two novel designs: a *Compaction Aware Track Allocation* scheme to minimize the time-consuming RMWs and efficiently access the SSTables during the compaction process, and a *Merged RMW* approach to improve the efficiency of persisting an SSTable into IMR based HDD when the time-consuming RMWs are inevitable. Our evaluations on three well-known LSM-tree based KV store implementations (i.e., RocksDB, LevelDB, and HyperLevelDB) reveal that KVIMR not only improves the overall throughput by up to 1.55× under write-intensive workloads but even achieves 2.17× higher throughput under high space usage of HDD, as compared with the state-of-the-art three-phase track allocation scheme for IMR.

7 Acknowledgements

We thank our shepherd, Raja Appuswamy, and all the anonymous reviewers for their valuable comments and suggestions. This work is supported in part by The Research Grants Council of Hong Kong SAR (Project No. CUHK24209618).

References

[1] Desktop hdd product manual standard models st1000dm003 st500dm002. https://www.seagate.com/www-content/product-content/desktop-hdd-fam/en-us/docs/100768625b.pdf.

[2] Ensuring data reaches disk. https://lwn.net/Articles/457667/.

[3] fdatasync. https://man7.org/linux/man-pages/man2/fdatasync.2.html.

[4] Hgst. libzbc version 5.4.1. https://github.com/hgst/libzbc.

[5] POSIX: Portable Operating System Interface. https://pubs.opengroup.org/onlinepubs/9699919799/.

[6] Rocksdb tuning guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.

[7] Sync. https://man7.org/linux/man-pages/man2/sync.2.html.

[8] Tuning rocksdb on spinning disks. https://github.com/facebook/rocksdb/wiki/Tuning-RocksDB-on-Spinning-Disks.

[9] Write stalls. https://github.com/facebook/rocksdb/wiki/Write-Stalls.

[10] Abutalib Aghayev, Mansour Shafaei, and Peter Desnoyers. Skylight—a window on shingled disk operation. *ACM Transactions on Storage (TOS)*, 11(4):16, 2015.

[11] Ahmed Amer, JoAnne Holliday, Darrell DE Long, Ethan L Miller, Jehan-François Pâris, and Thomas Schwarz. Data management and layout for shingled magnetic recording. *IEEE Transactions on Magnetics*, 47(10):3691–3697, 2011.

[12] W. A. Challener, C. Peng, A. V. Itagi, D. Karns, Y. Peng, X. Yang, X. Zhu, N. J. Gokemeijer, Y. T. Hsia, G. Ju, R. E. Rottmayer, M. A. Seigler, and E. C. Gage. The road to hamr. In *Magnetic Recording Conference, 2009. APMRC '09. Asia-Pacific*, pages 1–2, Jan 2009.

[13] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[14] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Yahoo! cloud serving benchmark (ycsb), 2010.

[15] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[16] E. A. Dobisz, Z. Z. Bandic, T. W. Wu, and T. Albrecht. Patterned media: Nanofabrication challenges of future disk drives. *Proceedings of the IEEE*, 96(11):1836–1846, Nov 2008.

[17] Robert Escriva, Sanjay Ghemawat, David Grogan, Jeremy Fitzhardinge, and Chris Mumford. Hyperleveldb, 2019. https://github.com/rescrv/HyperLevelDB.

[18] Facebook. Rocksdb: a persistent key-value store for fast storage enviroments., 2011. https://github.com/facebook/rocksdb.

[19] Kaizhong Gao, Wenzhong Zhu, and Edward Gage. Write management for interlaced magnetic recording devices, November 29 2016. US Patent 9,508,362.

[20] Kaizhong Gao, Wenzhong Zhu, and Edward Gage. Interlaced magnetic recording, August 8 2017. US Patent 9,728,206.

[21] Sanjay Ghemawat and Jeff Dean. Leveldb, 2011. https://github.com/google/leveldb.

[22] Steven Granz, Jason Jury, Chris Rea, Ganping Ju, Jan-Ulrich Thiele, Tim Rausch, and Edward Gage. Areal density comparison between conventional, shingled, and interlaced heat-assisted magnetic recording with multiple sensor magnetic recording. *IEEE Transactions on Magnetics*, PP:1–3, 09 2018.

[23] Steven Granz, Wenzhong Zhu, Edmun Chian Song Seng, Utt Heng Kan, Chris Rea, Ganping Ju, Jan-Ulrich Thiele, Tim Rausch, and Edward C Gage. Heat-assisted interlaced magnetic recording. *IEEE Transactions on Magnetics*, 54(2):1–4, 2017.

[24] Jorge Guerra, Leonardo Mármol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. Software persistent memory. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pages 319–331, 2012.

[25] Mohammad Hossein Hajkazemi, Ajay Narayan Kulkarni, Peter Desnoyers, and Timothy R Feldman. Track-based translation layers for interlaced magnetic recording. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 821–832, 2019.

[26] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Analysis of {HDFS} under hbase: A facebook messages case study. In *Proceedings of the 12th {USENIX} Conference on File and Storage Technologies ({FAST} 14)*, pages 199–212, 2014.

[27] Weiping He and David HC Du. Smart: An approach to shingled magnetic recording translation. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pages 121–134, 2017.

[28] Euiseok Hwang, Jongseung Park, Richard Rauschmayer, and Bruce Wilson. Interlaced magnetic recording. *IEEE Transactions on Magnetics*, 53(4):1–7, 2016.

[29] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The multi-streamed solid-state drive. In *6th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, 2014.

[30] P. Kasiraj, R. New, J. de Souza, and M. Williams. System and method for writing data to dedicated bands of a hard disk drive. In *US Patent 7490212*, Feb 2009.

[31] A. Kikitsu, Y. Kamata, M. Sakurai, and K. Naito. Recent progress of patterned media. *IEEE Transactions on Magnetics*, 43(9):3685–3688, Sept 2007.

[32] M. H. Kryder, E. C. Gage, T. W. McDaniel, W. A. Challener, R. E. Rottmayer, G. Ju, Y. T. Hsia, and M. F. Erden. Heat assisted magnetic recording. *Proceedings of the IEEE*, 96(11):1810–1835, Nov 2008.

[33] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. Atlas: Baidu's key-value storage system for cloud data. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14. IEEE, 2015.

[34] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[35] Shaoping Li, Gerardo A Bertero, Michael L Mallary, Ge Yi, and Steven C Rudy. Connection schemes for a multiple sensor array usable in two-dimensional magnetic recording, November 18 2014. US Patent 8,891,207.

[36] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)*, 13(1):5, 2017.

[37] Miha Marolt and Z Jaglicic. Superparamagnetic materials. In *Proceeding of Seminar Ib-4th Year (Old Program), University of Ljubljana Faculty of Mathematics and Physics*, 2014.

[38] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[39] Rekha Pitchumani, James Hughes, and Ethan L Miller. Smrdb: key-value data store for shingled magnetic recording disks. In *Proceedings of the 8th ACM International Systems and Storage Conference*, page 18. ACM, 2015.

[40] HJ Richter, AY Dobin, RT Lynch, D Weller, RM Brockie, O Heinonen, KZ Gao, J Xue, RJM vd Veerdonk, P Asselin, et al. Recording potential of bit-patterned media. *Applied Physics Letters*, 88(22):222512, 2006.

[41] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[42] Mansour Shafaei, Mohammad Hossein Hajkazemi, Peter Desnoyers, and Abutalib Aghayev. Modeling drive-managed smr performance. *ACM Transactions on Storage (TOS)*, 13(4):38, 2017.

[43] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pages 18–18. USENIX Association, 2012.

[44] I. Tagawa and M. Williams. High density data-storage using shingled-write. In *IEEE International Magnetic Conference,*, 2009.

[45] Shucheng Wang, Ziyi Lu, Qiang Cao, Hong Jiang, Jie Yao, Yuanyuan Dong, and Puyuan Yang. {BCW}: Buffer-controlled writes to hdds for ssd-hdd hybrid storage server. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pages 253–266, 2020.

[46] R. Wood, M. Williams, A. Kavcic, and J. Miles. The feasibility of magnetic recording at 10 terabits per square inch on conventional media. *IEEE Transactions on Magnetics*, 45(2):917–923, Feb 2009.

[47] Roger Wood. Shingled magnetic recording and two-dimensional magnetic recording. *IEEE Magnetics Society, Santa Clara Valley*, 2010.

[48] Fenggang Wu, Bingzhe Li, Baoquan Zhang, Zhichao Cao, Jim Diehl, Hao Wen, and David HC Du. Tracklace:

Data management for interlaced magnetic recording. *IEEE Transactions on Computers*, 2020.

[49] Fenggang Wu, Ming-Chang Yang, Ziqi Fan, Baoquan Zhang, Xiongzi Ge, and David HC Du. Evaluating host aware {SMR} drives. In *8th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[50] Fenggang Wu, Baoquan Zhang, Zhichao Cao, Hao Wen, Bingzhe Li, Jim Diehl, Guohua Wang, and David HC Du. Data management design for interlaced magnetic recording. In *10th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[51] Ming-Chang Yang, Yuan-Hao Chang, Fenggang Wu, Tei-Wei Kuo, and David HC Du. On improving the write responsiveness for host-aware smr drives. *IEEE Transactions on Computers*, 68(1):111–124, 2018.

[52] Ting Yao, Zhihu Tan, Jiguang Wan, Ping Huang, Yiwen Zhang, Changsheng Xie, and Xubin He. Sealdb: An efficient lsm-tree based kv store on smr drives with sets and dynamic bands. *IEEE Transactions on Parallel and Distributed Systems*, 30(11):2595–2607, 2019.

[53] Ting Yao, Jiguang Wan, Ping Huang, Yiwen Zhang, Zhiwen Liu, Changsheng Xie, and Xubin He. Geardb: a gc-free key-value store on hm-smr drives with gear compaction. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 159–171, 2019.

[54] Hwanjin Yong, Kisik Jeong, Joonwon Lee, and Jin-Soo Kim. vstream: virtual stream management for multi-streamed ssds. In *10th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[55] J. G. Zhu, X. Zhu, and Y. Tang. Microwave assisted magnetic recording. *IEEE Transactions on Magnetics*, 44(1):125–131, Jan 2008.