On the Design of SMR HDD Block Device Driver

Jingpeng Hao, Xubin Chen, Yifan Qiao, Yuyang Zhang, Tong Zhang
Electrical, Computer, and Systems Engineering Department, Rensselaer Polytechnic Institute
Troy, USA, haoj@rpi.edu

***Abstract*—This paper studies the design of host-managed SMR HDD. To obviate any changes to existing filesystems and applications, this paper focuses on making the host-side block device driver solely responsible for managing SMR HDD. The key is to make block device driver elastically utilize the host-side memory resource to realize address translation and mitigate SMR HDD performance degradation. Under this framework, this paper presents a set of techniques, including defragmentation-centric write buffer management, fragmentation-adaptive tree-based address mapping, and fragmentation-aware GC, which together can efficiently utilize host memory resource to realize address translation and mitigate drive performance degradation. Using a variety of disk access IO traces and benchmarks, we carried out experiments that well demonstrate the effectiveness of the proposed design techniques.**

***Index Terms*—Shingled magnetic recording (SMR), hard disk drive, address translation, write buffering, defragmentation**

I. Introduction

This paper studies the management of SMR (shingled magnetic recording) HDD (hard disk drive). By trading the convenient in-place update feature for smaller disk track pitch, SMR technology [1], [2] is the most economically viable option to sustain the continuous bit cost reduction of HDD. The absence of in-place update forces SMR HDD to internally employ a per-zone append-only data structure. As a result, SMR storage inherently mismatches with the conventional block device interface, and is natively subject to performance degradation caused by data fragmentation and garbage collection (GC). To deploy SMR HDD in computing systems, one option is to make SMR HDD *host-managed*: SMR HDD simply exposes itself as an append-only zoned storage device, and host-side software stack must accordingly adjust their data management and operations in order to fully comply with the per-zone append-only constraint. As a result, host-side software stack must handle data fragmentation and GC on their own. The host-managed approach simplifies the SMR HDD implementation at the cost of higher design complexity of the host-side software stack.

Under the host-managed design framework, one essential question is which layer in the software stack should be responsible for realizing address translation and mitigating SMR drive performance degradation. With the better knowledge about the data, upper-level software layers (i.e., filesystem, and even applications) should be able to more effectively manage SMR HDD. Nevertheless, this demands intrusive and potentially significant changes to existing upper-level software stack, leading to high development/deployment cost and hence a high barrier for its practical adoption. Therefore, this work focuses on making the block device drive solely responsible for realizing host-side management, which keeps the upper-level software stack completely intact.

To facilitate the implementation of such SMR HDD block device driver, this paper presents the following specific design techniques: (1) *Defragmentation-centric write buffer management*: We propose a write buffer management strategy that prioritizes on minimizing the LBA-PBA mapping fragmentation[1]. Meanwhile, we employ the simple concept of data journaling to enable the use of write buffer without sacrificing the data persistency. (2) *Fragmentation-aware GC*: We propose a method that chooses the GC candidate zones by cohesively considering the storage space reclaim efficiency and LBA-PBA mapping fragmentation. Sharing the common theme of reducing LBA-PBA mapping fragmentation, these two techniques are motivated by the following observation: Modern filesystems (e.g., ext4 and XFS) use extent-based and delayed space allocation to reduce the fragmentation of each file over the LBA space. Meanwhile, files on HDD are typically accessed in a coarse granularity (e.g., 64KB and 256KB). Lower LBA-PBA mapping fragmentation can be leveraged to reduce the memory usage of LBA-PBA address translation. In particular, we propose a *fragmentation-adaptive address mapping tree* to implement the LBA-PBA translation, which can make the mapping memory usage proportional to the LBA-PBA mapping fragmentation. This can be considered as a tree-based fragmentation-adaptive LBA-PBA address mapping.

To evaluate the effectiveness of the proposed design techniques, we carried out a variety of experiments using the Systor'17 Traces [3] and the big data benchmark suite HiBench 7.0 [4]. Regarding the proposed defragmentation-centric write buffer management, we compared it with the classical LRU-based management. Under no more than 5GB write buffer capacity, the experimental results show that, compared with LRU-based write buffer management, the proposed design approach can reduce the drive average read latency and 99-percentile read latency by up to 91% and 93%, respectively. In the case of using conventional table-based LBA-PBA address translation, the proposed defragmentation-centric write buffer management can reduce the write amplification by up to 93%, compared with LRU-based management. Regarding the proposed fragmentation-adaptive tree-based LBA-PBA address

[1] LBA-PBA mapping fragmentation occurs when continuous LBAs are mapped onto discontinuous PBAs.

mapping, the experimental results show that it can effectively reduce the address mapping memory usage. For example, for the Bayes benchmark (active dataset size of 262GB) in the HiBench benchmark suite, the proposed tree-based design approach only consumes less than 10MB of memory when the write buffer capacity is only 512MB. Regarding the proposed fragmentation-aware GC candidate zone selection, we compared it with the conventional practice that always selects the GC candidate zones solely based on the per-zone garbage rate. The experimental results show that, compared with the conventional practice, the proposed fragmentation-aware GC candidate zone selection can achieve significantly smaller LBA-PBA mapping fragmentation. By collectively demonstrating promising effectiveness of the proposed design approaches, the experiments show that the host-assisted device-managed SMR HDD design strategy indeed could be a practically viable option for future computing systems to seamlessly deploy SMR HDD.

II. Proposed Design Techniques

A. Defragmentation-centric Write Buffer Management

In order to effectively reduce the LBA-PBA mapping fragmentation, the SMR HDD block device driver must use a write buffer, through which it can re-order and coalesce write requests. In adaptation to the runtime system memory usage, the block device driver can elastically adjust the write buffer size. To ensure the data persistency, the block device driver employs an on-disk journal as a backup for the write buffer. This is further illustrated in Fig. 1: The block device driver allocates a certain amount of host DRAM as a write buffer, and meanwhile relies on data journaling to ensure the data persistency. Its practical implementation involves two design issues: (1) write buffer management, and (2) on-disk journal space recycling. This subsection presents a set of techniques to address these two issues. In the remainder of this subsection, we first present the basic underlying design concept, and then elaborate on the specific implementation details.

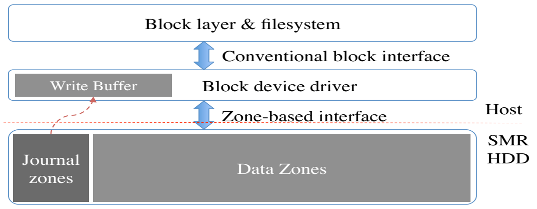

Fig. 1. Illustration of the envisioned host-managed SMR HDD, where the host-side block device driver uses an on-disk journal to ensure the persistency of its write buffer.

1) Basic Design Concept: Aiming to minimize the LBA-PBA mapping fragmentation, we propose the following write buffer management policy: Let the set $\mathbb{L} = \{\mathbb{L}_1, \mathbb{L}_2, \cdots\}$ represent the LBA regions of all the data in the write buffer, where each subset $\mathbb{L}_i$ spans over one contiguous LBA region. None of two subsets are adjacent on the LBA space. We note that each write request received by the block device driver always spans over contiguous LBAs. Given an incoming write request spanning over the LBA region $\mathbb{L}_{new}$, the block device driver will update the set $\mathbb{L}$ as follows:

- If $\mathbb{L}_{new}$ does not overlap with or is not adjacent to any subsets in $\mathbb{L}$, we simply insert $\mathbb{L}_{new}$ into the set $\mathbb{L}$.
- Otherwise, suppose $\mathbb{L}_{new}$ overlaps with and/or is adjacent to j subsets. Then we merge $\mathbb{L}_{new}$ with these j subsets to generate a new subset in $\mathbb{L}$. Meanwhile, we remove these j subsets from the set $\mathbb{L}$.

Let C_w denote the total capacity of the write buffer, and let $|\mathbb{L}_i|$ denote the number of contiguous LBAs covered by $\mathbb{L}_i$. Once the block device driver has updated the set $\mathbb{L}$ upon a new write request, it will carry out the following operation:

- If $\sum_{\forall i} |\mathbb{L}_i| \leq C_w$ (i.e., the write buffer can absorb the incoming write request), then we accordingly update the content of the write buffer. Meanwhile, we log the write request in the on-disk journal.
- Otherwise (i.e., the write buffer cannot absorb the incoming write request), let $\mathbb{L}_m$ denote the subset that spans over the largest LBA region in the updated $\mathbb{L}$, i.e.,

$$|\mathbb{L}_m| \geq |\mathbb{L}_i|, \ \forall \mathbb{L}_i \in \mathbb{L} \setminus \mathbb{L}_m. \tag{1}$$

  Then we evict all the data associated with $\mathbb{L}_m$ from the write buffer to SMR HDD, and accordingly update the SMR HDD LBA-PBA mapping. Meanwhile, we remove the subset $\mathbb{L}_m$ from $\mathbb{L}$, and log the data eviction action in the on-disk journal.

In addition to managing the write buffer, the block device driver should also manage the on-disk journal. For each zone in the on-disk journal area, let the set $\mathbb{D}^{(k)} = \{\mathbb{D}_1^{(k)}, \mathbb{D}_2^{(k)} \cdot \cdots\}$ represent the LBA regions of all the data on this zone that still reside in the write buffer, where each subset $\mathbb{D}_i^{(k)}$ spans over one contiguous LBA region that maps to one contiguous PBA region on the zone. When the block device driver admits or evicts data into/from the write buffer, it must accordingly update the related $\mathbb{D}^{(k)}$'s. In the on-disk journal area, we keep only one zone as open and can only journal data into this open zone. Once the open zone is completely filled, we will seal this zone, and designate another empty zone as the open zone. When the number of sealed zones in the on-disk journal area exceeds a pre-defined threshold, the block device driver will recycle a sealed zone as follows: Among all the sealed zones in the on-disk journal area, we choose the one that contains the minimal amount of active data, i.e., the one with the minimum $\sum_{\forall i} |\mathbb{D}_i^{(k)}|$ among all the sealed zones. Let $\mathbb{D}^{(s)}$ denotes the set associated with the chosen sealed zone. Each subset $\mathbb{D}_i^{(s)} \in \mathbb{D}^{(s)}$ must be equal or belong to one subset $\mathbb{L}_j \in \mathbb{L}$. Therefore, we can form a set $\mathbb{L}^{(s)} = \{\mathbb{L}_j, \cdots\}$ that just sufficiently covers the entire LBA regions of $\mathbb{D}^{(s)}$. Accordingly, the block device driver will evict the data associated with $\mathbb{L}^{(s)}$ from the write buffer to SMR HDD. As a result, $\mathbb{D}^{(s)} = \varnothing$ for the chosen sealed zone, and it can be readily recycled.

2) Practical Implementation: The above presents the basic design concept underlying the proposed defragmentation-centric write buffer management strategy. Its practical implementation is non-trivial and must effectively manage $\mathbb{L}$ for the write buffer and $\mathbb{D}^{(k)}$'s for all the zones in the on-disk journal area. This subsection elaborates on our proposed implementation approach.

First, based upon the classical red-black tree structure, we propose an *LBA-size tree* to implement the write buffer management. In Fig. 2, a node's color is bl is the flag of each node in red-black tree struc the tree. As illustrated in Fig. 2, each node tree represents a subset from $\mathbb{L} = \{\mathbb{L}_1, \mathbb{L}_2$ node represents one contiguous LBA region is the starting LBA of the LBA region, anc is the size of the LBA region (i.e., the num LBAs). Upon receiving a new write reque such a node whose key is the same as the st write request after searching the tree, the b first creates a *virtual node*: its key and valu LBA and size of the write request. We te virtual node at the place where the search predecessor and successor of the virtual nc as illustrated in Fig. 2 (a), assuming the vir between 144 and 184, we find the predeces of the virtual node, i.e., node 144 and node node's predecessor and successor could be l according to the node's location. Once we l predecessor and successor of the virtual noc the following different scenarios (Let K_V the virtual node, and let S_V denote the rec virtual node):

1) If the virtual node could not be merged with either its predecessor or successor, we will insert the virtual node into the tree, and re-balance the tree if necessary. For example, if $K_V = 168$ and $S_V = 8$, the tree is updated as shown in Fig. 2 (b).
2) If the LBA range of the virtual node falls into the LBA range of its predecessor, the tree would not be updated at all, and we simply update the existing data in the write buffer.
3) If the virtual node could be merged with the predecessor, the predecessor's recorded size is updated. If the virtual node could be merged with one or even more successors, we delete all of the related successors. For e $K_V = 160$ and $S_V = 48$, then the virtual r be merged with the nodes 144, 184 and 200 is updated as Fig. 2 (c) shows.
4) If the virtual node could be merged with the but not with the predecessor, we update th value (i.e., recorded size) of its successor. If node could be merged with more than one s we delete all of the related successors excep successor. For example, if $K_V = 168$ and then the virtual node could be merged with and 200. The tree is updated as Fig. 2 (d) sh

If there is a tree node whose key is the same as t LBA of the write request, the block device drive create a virtual node. The node's size value is updated if it is less than the size of the write request. If the node with the new size value could be merged with one or several successors, these successors would be deleted and the node's size value will be further updated.

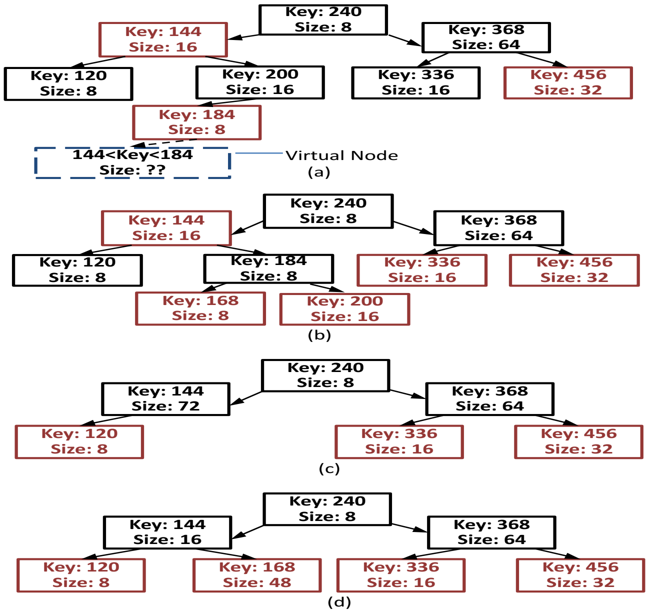

Fig. 2. Examples to illustrate the proposed LBA-size tree (using the red-black tree structure), where the nodes with the red or black color are the red and black nodes in the red-black tree.

To ensure the data persistency, we use an on-disk journal to recover data in case of power failure or system crash. Fig. 3(a) shows the data written on the journal zones and Fig. 3(b) shows the content of a journal file before power off or system crash. For each continuous data segment being written in the journal, the journal file records its starting LBA/PBA, segment size, and a timestamp. When the block device driver evicts one continuous data segment from the write buffer, it will append a record to the journal, where the record contains the starting LBA, size, and timestamp (as shown in Fig. 3(b), the starting PBA is set as -1). In case of power failure or system crash, the block device driver can reconstruct the write buffer content by

LBA: ···	800	801	802	803	520	521	802	803	521
PBA: ···	1000	1001	1002	1003	1004	1005	1006	1007	1008

(a)

Journal File

Starting LBA	Starting PBA	Size	Timestamp
⋮	⋮	⋮	⋮
800	1000	4	100
520	1004	2	101
800	-1	4	102
802	1006	2	103
521	1008	1	104

(b)

Fig. 3. Examples to illustrate the on-disk journal.

B. Fragmentation-Adaptive LBA-PBA Address Translation

In this work, we are only interested in sector-based LBA-PBA mapping (i.e., each LBA can be mapped to any PBA) in order to reduce the write amplification caused by address translation. The most straightforward implementation is a sector-based LBA-PBA mapping table, where each table entry corresponds to one unique LBA. However, it suffers from significant memory usage (i.e., the mapping table size is about 0.1% of the SMR HDD capacity). As discussed above, the block device driver employs a host-side write buffer to reduce the LBA-PBA mapping fragmentation. Intuitively, with sufficiently low LBA-PBA mapping fragmentation, we can use a tree structure to implement sector-based LBA-PBA mapping at small memory usage.

Based upon the classical red-black tree structure, we propose a sector-based *fragmentation-adaptive address mapping tree* to realize fragmentation-adaptive LBA-PBA address translation (i.e., its memory usage is adaptive to the LBA-PBA mapping fragmentation). The reason why the proposed trees are based on the red-black tree rather than the B+ tree is that the proposed trees are totally stored in DRAM due to their small memory usage. Each node in the proposed mapping tree represents a continuous LBA-PBA mapping region (i.e., data in this region are continuous in terms of both LBA and PBA). Each node's key is the starting LBA of a continuous LBA-PBA mapping region, and there are two values in each node: (1) the size of this region, and (2) the starting PBA of this region. When the block device driver flushes data from the host-side
write buffer to SMR HDD, it first creates a *virtual*
key is the starting LBA of this region, its size va
size of the region and its PBA value is the starting
SMR HDD. If none of the existing nodes in the fragn
adaptive address mapping tree has the same key as tl
node, we temporarily put the virtual node at the pla
the search ends, and find the predecessor and succes
virtual node. For example, as illustrated in Fig. 4(a),
the virtual node's key is between 100 and 320, wo
predecessor and successor of the virtual node, i.e.,
and node 320 in the tree. To maintain the LBA-PBA
tree, we need to handle the following different scei
K_V denote the key of the virtual node, and let S
the recorded size of the virtual node, and let P_V d
recorded PBA of the virtual node):

1) There is a node in the tree whose key is K_V:
   a) If the node's size is equal to S_V, just u
   node's PBA value as P_V.
   b) If the node's size is larger than S_V,
   will be divided into two nodes. For ex
   K_V=100, S_V=50 and P_V=8,100, the maj
   will be updated as shown in Fig. 4(b). The
   node 100 is divided into two nodes: nod
   node 150.
   c) If S_V is larger than the node's size, up
   this node's size value as S_V and PBA val
   d) If the virtual node covers part of the in terms of LBA, the successor's key, recorded size and recorded PBA will be updated. If the virtual node entirely covers the successor in terms of LBA, then we delete the successor, and repeat the process until the virtual node does not cover the next successors in terms of LBA. For example, as shown in Fig. 4(c), assume K_V=100, S_V=400 and P_V=8100, we have that the node 320 will be deleted. Meanwhile, since the virtual node overlaps with the next successor (i.e., node 430), the key, and value size and value PBA of the node 430 are all updated.
2) None of the existing nodes in the tree has the same key as K_V:
   a) If the virtual node does not cover the predecessor and the successor in terms of LBA, then we directly insert the virtual node into the tree.
   b) If the virtual node covers a mid part of the predecessor in terms of LBA, the predecessor is divided into three nodes. For example, assume K_V=120, S_V=50, and P_V=8100, then we update the tree as shown in Fig. 4 (d), where the node 100 is divided into three nodes: node 100, node 120 and node 170.
   c) If only a part of the virtual node covers the predecessor in terms of LBA, then we insert the virtual node and update the predecessor's size value.

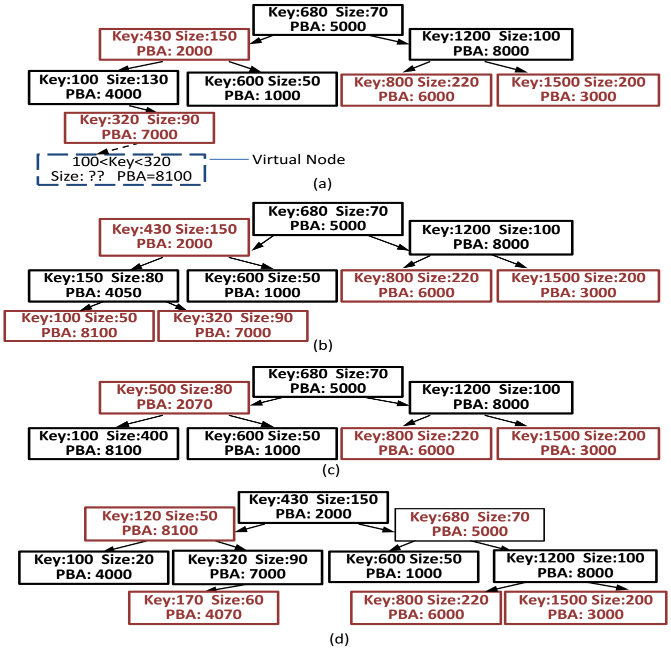

Fig. 4. Illustration of the proposed fragmentation-adaptive LBA-PBA address mapping tree (using the red-black tree structure), where the nodes with the red or black color are the red and black nodes in the red-black tree.

When the block device driver receives a read request, it searches the LBA-PBA mapping tree to obtain the corresponding PBAs. If there is a node in the tree whose key is the same as the starting LBA of the read request, then we can get the information of the related PBA from this node. If the read request's size is larger than the node's size value, then we can obtain the PBAs from this node and its successors. If none of the nodes in the LBA-PBA mapping tree has the same key as the read request's starting LBA, we can find the predecessor from where the search ends according to the feature of tree structure. Then it's easy to determine the starting PBA of the read request.

C. Fragmentation-aware SMR HDD GC

The block device driver is also responsible for scheduling SMR HDD GC, where one critical task is to choose the candidate zone for recycling. Intuitively, one may want to always choose the zone that contains the least amount of valid data. Indeed this intuition works well for SSD because the GC-induced operational latency overhead is strictly proportional to the amount of valid data in the to-be-recycled flash block. Nevertheless, in the context of SMR HDD, the GC-induced operational latency overhead is weakly dependent on the amount of valid data, which can be explained as follows: Let τ_{GC} denote the GC-induced operational latency overhead, which can be expressed as

$$\tau_{GC} = \tau_{seek} + \tau_r + \tau_w, \tag{2}$$

where τ_{seek} denotes the total head seek latency, τ_r denotes the latency of reading all the valid data from the to-be-recycled zone, and τ_w denotes the latency of writing all the valid data to another open zone. Let C_v denote the amount of valid data in the to-be-recycled zone. Clearly, τ_{seek} is independent from C_v. Suppose each zone in SMR HDD contains n_t (e.g., 256) shingled tracks. Since the valid data most likely scatter among all the n_t tracks, regardless of the value of C_v, SMR HDD may always spend close-to-n_t disk rotations to read out all the valid data. Hence, τ_r is almost independent from C_v. The write latency τ_w is linearly proportional to C_v. As long as C_v is substantially smaller than the capacity of one zone, τ_w will be significantly lower than τ_r. Therefore, it is reasonable to draw the conclusion that the GC-induced operational latency overhead τ_{GC} is weakly dependent on the amount of valid data in the to-be-recycled zone.

Motivated by the above observation and the importance of reducing LBA-PBA mapping fragmentation in SMR HDD, we propose a fragmentation-aware strategy that chooses the candidate zone by cohesively taking into account of the valid data amount and LBA-PBA mapping fragmentation. This proposed strategy can be described as follows: Among all the sealed zones, we first find a relatively small number n_c (e.g., 20) of zones that contain less amount of valid data than all the others. Within the n_c sealed zones that contain small amount of valid data, we further choose n_r (e.g., 3) zones for recycling, solely based on the fragmentation criterion. In particular, for each zone, let $\mathbb{L}_v^{(k)} = \{\mathbb{L}_{v,1}^{(k)}, \mathbb{L}_{v,2}^{(k)}, \cdots\}$ denote the set of valid data, where each subset $\mathbb{L}_{v,i}^{(k)}$ denotes the LBA region of a valid data segment with contiguous LBA-PBA mapping inside the zone. Define $\mathcal{M}(\mathbb{L}_v^{(k)})$ as the function that merges all the adjacent subsets (i.e., two or more subsets whose LBA regions are adjacent to each other) in $\mathbb{L}_v^{(k)}$ into a single larger subset. Let $|\mathbb{L}_v^{(k)}|$ denote the number of subsets in the set $\mathbb{L}_v^{(k)}$, we have that $|\mathbb{L}_v^{(k)}| \geq |\mathcal{M}(\mathbb{L}_v^{(k)})|$. As discussed above, out of the total n_c zones, we should choose n_r zones that can minimize the LBA-PBA mapping fragmentation. This can be formulated as follows: Let $\mathbb{U}_i^{(n_r)}$ denote the union set of the sets $\mathbb{L}_v^{(k)}$'s of n_r zones. Given the total n_c zones, there are $\binom{n_c}{n_r}$ different union sets $\mathbb{U}_i^{(n_r)}$'s. In order to minimize the LBA-PBA mapping fragmentation, we should choose the union set $\mathbb{U}_d^{(n_r)}$ that can enable the largest number of subsets to be merged, i.e.,

$$|\mathbb{U}_d^{(n_r)}| - |\mathcal{M}(\mathbb{U}_d^{(n_r)})| \geq |\mathbb{U}_i^{(n_r)}| - |\mathcal{M}(\mathbb{U}_i^{(n_r)})|, \; \forall \mathbb{U}_i^{(n_r)}. \tag{3}$$

To practically implement the above fragmentation-aware zone selection, we developed the following approach by leveraging the LBA-PBA mapping tree (discussed above in Section II-B): Within the LBA-PBA mapping tree, let $R_{i,j}$ denote the reduced number of nodes if we choose zone i and zone j for recycling, and let $R_{i,i}$ denote the reduced number of nodes if we choose only zone i for recycling. Then we can carry out fragmentation-aware zone selection using a recursive heuristic method: Suppose we choose the top 10 zones that have less amount of valid data than the others (i.e., n_c is 10), from which we select n_r zones for recycling in a fragmentation-aware manner. If we aim to select only one zone for recycling (i.e., $n_r = 1$), then we simply choose the zone with the largest $R_{i,i}$. If we aim to select two zones for recycling (i.e., $n_r = 2$), we simply select the two zones so that the associated $R_{i,j}$ is the largest. If we aim to select $n_r > 2$ zones for recycling, we first select the two zones with the largest $R_{i,j}$, then we select the remaining zones one-by-one as follows: To select the k-th zone ($k > 2$), let $\mathbb{N}_{k-1}$ denote the set that contains the selected $k-1$ zones so far, and we select the k-th zone so that it has the largest $R_{k,j}$ for all $j \in \mathbb{N}_{k-1}$. Although such as recursive heuristic method does not guarantee to find the truly optimal set of zones, it can achieve a good approximation at very small search computational complexity.

III. Evaluation

We carried out experiments on a server with dual-socket Intel Xeon E5-2630 2.2GHz CPUs (10 cores per socket), and 64GB DRAM. The server runs Linux Kernel 4.10.0 in the Ubuntu 16.04.03 distribution. We use three traces (LUN0, LUN2 and LUN4, and each trace contains 10-hour trace files) from Systor'17 Traces [3] and five benchmarks (Bayes, Kmeans, PageRank, Sort, and TeraSort) from the big data benchmark suite HiBench 7.0 [4]. We set up one master node and three slave nodes to run the benchmarks of HiBench 7.0, and each slave node has a 2TB HDD. The total time to run each benchmark is 10 hours. The traces from HiBench 7.0 have relatively larger average write and read size than traces from Systor' 17. Since we cannot directly implement the proposed schemes inside commercial SMR HDDs due to the limitation of the current commercial SMR HDDs, we simulate the SMR HDD using conventional 2TB SATA HDD in user space and replay the traces mentioned above. We set each zone as 256MB.

A. Defragmentation-centric Write Buffer Management

We first evaluate the effectiveness of the proposed defragmentation-centric write buffer management approach.

1) Improving drive speed performance: For the purpose of comparison, we also implemented the LRU-based write buffer management. Each read request spans over contiguous LBA region, which nevertheless may be broken into multiple disjoint PBA regions on SMR HDD due to the LBA-PBA mapping fragmentation. When the amount of data in the write buffer reaches 90% of the buffer capacity or when the number of sealed zones in the journal area reaches 90% of the total number of journal zones, we flush the data from write buffer to the drive. For all the experiments, we use the fragmentation-adaptive address mapping tree (discussed in Section II-B) to realize LBA-PBA mapping.

Fig. 5 and Fig. 6 show the average and 99-percentile tail read latency when using LRU and the proposed design approach, respectively. Latencies take into account data transfers with LBA-size tree and the on-disk journal. The results show that the proposed design approach can effectively reduce the drive read latency. When the write buffer capacity is 5GB, compared to LRU, the proposed design approach could reduce the average and the 99-percentile tail read latency by 87% and 77% for LUN2 trace, and can reduce the average and the 99-percentile tail latency remarkably by 91% and 93% for PageRank benchmark.

2) Reducing write amplification in table-based address translation: LBA-PBA address translation can be realized by using table or tree data structure. If we use table-based address translation, the LBA-PBA mapping granularity determines the trade-off between write amplification and table memory size. Let n_m denote the number of contiguous LBAs associated with each table entry, i.e., each table entry records a contiguous LBA-PBA mapping over n_m sectors. As we increase the value of n_m, we can reduce the mapping table size, but suffer from a higher write amplification. For example, when we update $n_s < n_m$ sectors, we have to read all the n_m sectors from the drive and write the updated n_m to the drive, leading to a write amplification.

We carried out experiments to measure the write amplification when using table-based LBA-PBA address translation with different mapping granularity: 10 and 100 contiguous sectors. We compared the proposed write buffer management approach with the LRU-based write buffer management. Fig. 7$\sim$Fig. 8 show the measured write amplification under different LBA-PBA mapping granularity. The results show that, by reducing the LBA-PBA mapping fragmentation, the proposed write buffer management approach can enable much smaller write amplification than LRU-based approach. As the mapping granularity increases (e.g., from 10 sectors per table entry to 100 sectors per table entry), the write amplification of LRU-based approach increases much more significantly than that of the proposed design approach. For example, for the trace LUN0 with 1GB write buffer capacity, when we increase the mapping granularity from 10 sectors to 100 sectors, the write amplification of the LRU-based approach increases from 2.1 to 11.8, while the write amplification of the proposed approach increases only from 1.1 to 2.3. The results show that the write amplification of Hibench benchmarks is much smaller than that of Systor'17 traces (i.e., LUN0, LUN2, and LUN4). This is because the Hibench benchmarks have much larger write request size, which naturally leads to small write amplification in the case of table-based address mapping.

B. LBA-PBA Mapping Tree

The purpose of the proposed fragmentation-adaptive LBA-PBA address mapping tree is to make the address translation memory usage become adaptive to the runtime LBA-PBA mapping fragmentation. Hence, once we have employed techniques (e.g., defragmentation-centric write buffer management) to reduce the LBA-PBA mapping fragmentation, it can accordingly reduce the address mapping memory usage.

Table I shows the memory usage of the proposed address mapping tree. For each trace/benchmark, the table shows the active dataset size and the corresponding address mapping tree size under different write buffer capacity. As demonstrated above in Section III-A, as we increase the write buffer capacity, the defragmentation-centric write buffer management can more effectively reduce the LBA-PBA mapping fragmentation. Therefore, as shown in Table I, the address mapping tree size always reduces as the write buffer capacity increases. The results show that, even without using the write buffer, the address mapping tree size still tends to be small, especially for the HiBench benchmarks where the write request size is large. For example, for the Kmeans benchmark with the active dataset size of 151GB, the address mapping tree memory usage is only 8.6MB even without write buffer.

TABLE I
ADDRESS MAPPING TREE MEMORY.

Trace	Active	Buffer Capacity			
	Dataset Size	0	512MB	1GB	5GB
LUN0	125GB	104MB	68MB	55MB	27MB
LUN2	146GB	101MB	86MB	75MB	38MB
LUN4	139GB	96MB	79MB	66MB	31MB
Bayes	262GB	15MB	9.2MB	7.1MB	1.5MB
Kmeans	151GB	8.6MB	2.9MB	1.5MB	0.25MB
PageRank	205GB	11MB	4.1MB	2.5MB	0.58MB
Sort	160GB	17MB	6.7MB	3.8MB	0.79MB
TeraSort	69GB	4.6MB	1.2MB	0.58MB	0.14MB

C. Fragmentation-aware GC

The key idea of fragmentation-aware GC is to cohesively take into account of garbage rate and LBA-PBA mapping fragmentation during the candidate zone selection. The number of nodes in the LBA-PBA address mapping tree can be used as a metric to quantify the overall LBA-PBA mapping fragmentation, i.e., the less number of nodes the address mapping tree contains, the less fragmented the LBA-PBA mapping is.

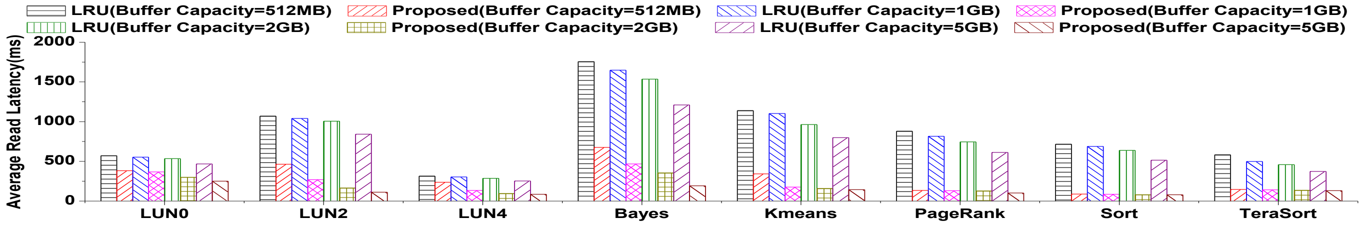

Fig. 5. Measured average read latency when using LRU and the proposed design approach.

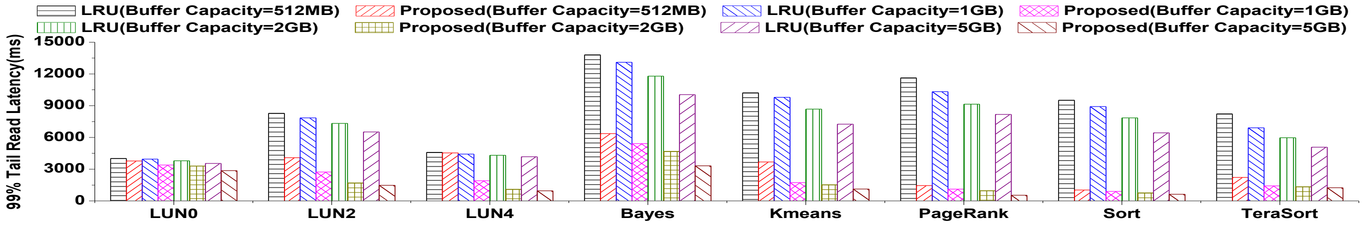

Fig. 6. Measured 99-percentile read latency when using LRU and the proposed design approach.

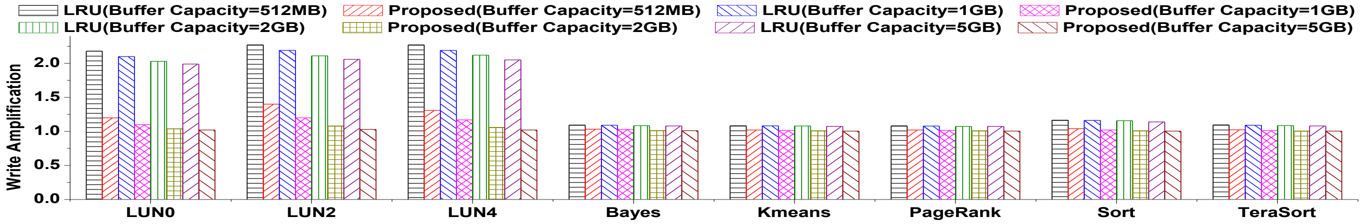

Fig. 7. Average write amplification when each LBA-PBA mapping table entry corresponds to 10 sectors.

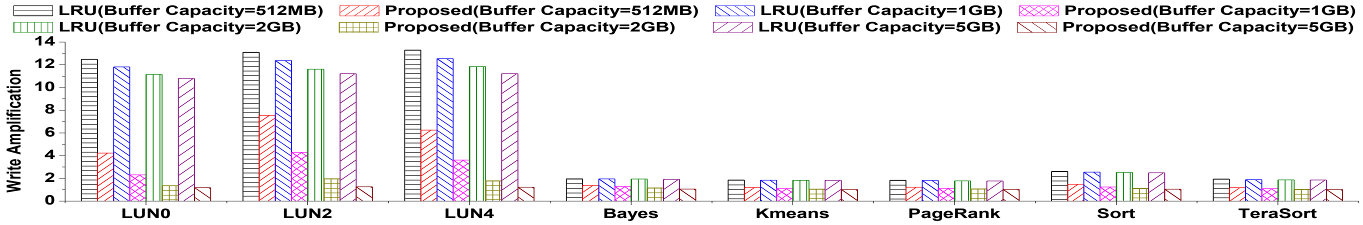

Fig. 8. Average write amplification when each LBA-PBA mapping table entry corresponds to 100 sectors.

TABLE II
ADDRESS MAPPING TREE NODE NUMBER REDUCTION.

Trace	Baseline				Proposed			
	Buffer Capacity				Buffer Capacity			
	0	512M	1B	5B	0	512B	1B	5B
LUN0	2691	547	236	32	8955	2538	822	277
LUN2	392	211	208	25	1518	1281	1006	179
LUN4	712	493	430	193	2955	1724	1138	756
Bayes	34	9	13	0	116	30	29	3
Kmeans	147	32	15	1	707	106	65	5
PageRank	87	27	9	2	286	97	43	9
Sort	47	14	5	0	122	45	25	6
TeraSort	95	13	1	0	202	52	28	4

TABLE III
THE GARBAGE RATE OF THE SELECTED TWO ZONES.

Trace	Baseline				Proposed			
	Buffer Capacity				Buffer Capacity			
	0	512M	1G	5G	0	512M	1G	5G
LUN0	91%	92%	94%	92%	84%	82%	85%	81%
LUN2	97%	96%	95%	93%	96%	92%	85%	88%
LUN4	90%	92%	93%	85%	81%	83%	81%	81%
Bayes	94%	96%	95%	93%	88%	93%	92%	90%
Kmeans	85%	90%	91%	84%	81%	81%	82%	82%
PageRank	92%	87%	89%	88%	82%	82%	83%	82%
Sort	98%	98%	97%	92%	95%	93%	93%	88%
TeraSort	92%	89%	92%	84%	87%	82%	82%	82%

For comparison, we also evaluate a baseline case where the GC candidate zones are selected solely based on the garbage rate. During each GC process, the baseline always selects 2 GC candidate zones that have higher garbage rate than all the other zones. In the case of our proposed fragmentation-aware GC, we first choose the 20 zones that have higher garbage rate than the other zones, from which we select 2 GC candidate zones based on the LBA-PBA fragmentation.

Table II shows that, compared with the baseline, the proposed defragmentation-aware zone selection can significantly reduce the LBA-PBA mapping fragmentation. As we increase the write buffer capacity, the absolute value of the node number reduction reduces. This is because, under larger write buffer capacity, the LBA-PBA mapping fragmentation will reduce, which will reduce the chance that multiple small data segments will be merged into a large segment during GC. Compared with the HiBench benchmarks, we can reduce much more tree nodes for Systor' 17 traces, because of the very small write request size in the Systor' 17 traces.

Table III further shows the garbage rate of the selected two zones during the GC process. Since the baseline selects the candidate zones solely based on the per-zone garbage rate, it can always achieve the highest garbage rate. In contrast, the fragmentation-aware GC has less garbage rate since it takes into account of LBA-PBA mapping fragmentation as well. Nevertheless, when using the fragmentation-aware GC, it can still achieve very high garbage rate, meanwhile it can much better contribute to reducing LBA-PBA mapping fragmentation. For example, for the Systor trace LUN2 under 512MB write buffer, the baseline reduces 211 tree nodes and has a garbage rate of 96%, while fragmentation-aware GC reduces 1281 tree nodes and has a garbage rate of 92%. The results in Table II and Table III suggest that the proposed fragmentation-aware GC can achieve a more desirable trade-off between the GC garbage rate and mapping fragmentation.

IV. Related Work

Prior research has studied the potential of applying write buffer/cache to mitigate the speed performance degradation of SMR HDD. Xie *et al.* [5] proposed an endurable SMR-oriented SSD Caching framework that uses SSD as a cache in front of SMR HDD. Ma *et al.* [6] proposed a persistent cache management scheme for SMR HDD, which can alleviate the hot-data write-back effect. Yang *et al.* [7] developed a virtual persistent cache design approach to improve the write responsiveness of host-aware SMR drives. Prior research also developed techniques to reduce the LBA-PBA mapping memory usage. He *et al.* [8] studied the track-based LBA-to-PBA mapping table and developed a hybrid update strategy that performs in-place update for certain qualified tracks. Shafaei *et al.* [9] proposed a track-based static mapping translation layer, which allows in-place data update by caching data on the adjacent track. Researchers also explored other options to enhance the real-world deployment of SMR HDD. Aghayev *et al.* [10] demonstrated that one could slightly modify existing filesystem in order to embrace the append-only write nature of SMR HDD. Manzanares *et al.* [11] developed a data management approach for SMR HDD that can minimize the metadata overhead of indirect writes. Hajkazemi *et al.* [12] developed techniques that can reduce the head seek latency in SMR HDD. Liang *et al.* [13] proposed a sequential-write-constrained B+-tree index scheme that better embraces the operational characteristics of SMR HDD. Zhang *et al.* [14] developed techniques that can reduce the impact of intra-drive data cleaning process on the drive IO performance.

V. Conclusion

With the goal of allowing computing systems to seamlessly deploy SMR HDD and meanwhile minimizing the performance degradation due to the loss of in-place update, this paper advocates a host-managed design framework. The essence is to make host-side block device driver responsible for realizing LBA-PBA address translation and improving drive performance. The key is to elastically and effectively utilize the host memory resource. Under this framework, we developed several techniques to enhance the block device driver, including defragmentation-centric write buffer management, fragmentation-adaptive tree-based LBA-PBA mapping, and fragmentation-aware SMR HDD GC. We carried out extensive experiments using a variety of IO traces, and the results well demonstrate the effectiveness of the proposed design techniques.

References

[1] K. Miura, E. Yamamoto, H. Aoi, and H. Muraoka, “Estimation of maximum track density in shingles writing,” *IEEE Transactions on Magnetics*, vol. 45, no. 10, pp. 3722–3725, Oct. 2009.

[2] F. Lim, B. Wilson, and R. Wood, “Analysis of shingle-write readback using magnetic-force microscopy,” *IEEE Transactions on Magnetics*, vol. 46, no. 6, pp. 1548–1551, Jun. 2010.

[3] C. Lee, T. Kumano, T. Matsuki, H. Endo, N. Fukumoto, and M. Sugawara, “Understanding storage traffic characteristics on enterprise virtual desktop infrastructure,” in *Proceedings of the 10th ACM International Systems and Storage Conference*. ACM, 2017, p. 13.

[4] *HiBench 7.0*. https://github.com/intel-hadoop/HiBench.

[5] X. Xie, L. Xiao, X. Ge, and Q. Li, “SMRC: An endurable SSD cache for host-aware shingled magnetic recording drives,” *IEEE Access*, vol. 6, pp. 20 916–20 928, 2018.

[6] C. Ma, Z. Shen, Y. Wang, and Z. Shao, “Alleviating hot data write back effect for shingled magnetic recording storage systems,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 12, pp. 2243–2254, Dec 2019.

[7] M. Yang, Y. Chang, F. Wu, T. Kuo, and D. H. C. Du, “On improving the write responsiveness for host-aware SMR drives,” *IEEE Transactions on Computers*, vol. 68, no. 1, pp. 111–124, Jan 2019.

[8] W. He and D. H. Du, “SMaRT: An approach to shingled magnetic recording translation,” in *USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2017, pp. 121–134.

[9] M. Shafaei and P. Desnoyers, “Virtual guard: A track-based translation layer for shingled disks,” in *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, Jul. 2017.

[10] A. Aghayev, T. Ts’o, G. Gibson, and P. Desnoyers, “Evolving Ext4 for shingled disks,” in *USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2017, pp. 105–120.

[11] A. Manzanares, N. Watkins, C. Guyot, D. LeMoal, C. Maltzahn, and Z. Bandic, “ZEA, a data management approach for SMR,” in *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, Jun. 2016.

[12] M. H. Hajkazemi, M. Abdi, and P. Desnoyers, “Minimizing read seeks for SMR disk,” in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2018, pp. 146–155.

[13] Y.-P. Liang, T.-Y. Chen, Y.-H. Chang, S.-H. Chen, K.-Y. Lam, W.-H. Li, and W.-K. Shih, “Enabling sequential-write-constrained B+-Tree index scheme to upgrade shingled magnetic recording storage performance,” *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 5s, oct 2019.

[14] B. Zhang, M. Yang, X. Xie, and D. H. C. Du, “Idler: I/O workload controlling for better responsiveness on host-aware shingled magnetic recording drives,” *IEEE Transactions on Computers*, 2020.