Implementing Flash-Cached Storage Systems Using Computational Storage Drive with Built-in Transparent Compression

Jingpeng Hao[†], Yifan Qiao[†], Xubin Chen[†], Ning Zheng[‡], Yang Liu[‡], Jiangpeng Li[‡], Qi Wu[‡], Tong Zhang[†‡]

[†] Rensselaer Polytechnic Institute, NY, USA

[‡] ScaleFlux Inc., CA, USA

***Abstract*—This paper studies utilizing the growing family of solid-state drives (SSDs) with built-in transparent compression to simplify the data structure of cache design. Such storage hardware allows the user applications to intentionally under-utilize logical storage space (i.e., sparse LBA utilization, and sparse storage block content) without sacrificing the physical storage space. Accordingly, this work proposed an index-less cache management approach to largely simplify the flash-based cache management by leveraging SSDs with built-in transparent compression. We carried out various experiments to evaluate the write amplification and read performance of the proposed cache management, and the results show that our proposed index-less cache management can achieve comparable or much better performance than the conventional policies while consuming much less host computing and memory resources.**

I. Introduction

This paper studies how one could leverage the emerging computational storage drive (CSD) [1], [2] to innovate the design of flash-cached data storage systems. The simple concept of empowering storage devices with computing capability can trace back to over 20 years ago [3]–[5] and lately received significant resurgent interest (e.g., see [6]–[12]). As one special family of CSD products, CSD with built-in transparent compression (referred to as TC-CSD) internally performs hardware-based per-4KB data compression/decompression, which is transparent to the host operating systems and user applications. Currently, TC-CSD is the only type of CSD that has been successfully commercialized and deployed in the production environment [13], [14].

In addition to its obvious benefit of lowering the storage cost at zero CPU overhead, TC-CSD enables unique opportunities for system-level innovations by supporting two types of sparsity: (1) Sparse LBA (logical block address) space: TC-CSD can expose a sparse LBA space that is much larger than its internal physical storage space. (2) Sparse sector content: Since special data patterns (e.g., all-zeros) can be highly compressed, we could leave one 4KB sector partially filled with valid user data, without sacrificing the true physical storage cost. With these two types of sparsity, TC-CSD decouples the logical storage space utilization efficiency from the physical storage space utilization efficiency [15]. This allows systems to *purposely under-utilize* the logical storage space in order to employ simpler data structures and algorithms, without compromising the true physical storage cost. Simpler data structures and algorithms may lead to higher system performance and/or lower CPU/memory cost.

This work envisions flash-cached storage systems using TC-CSD as the caching device. In particular, we focus on block-level caching that manages the cached storage systems at the block layer, being transparent to upper-level file systems and user applications. Although block-level caching can be conveniently deployed without any changes to the upper-level software, its implementation is subject to two major issues: (1) *Cache management overhead*: Conventional implementation of a cached storage system uses a hash-based (or tree-based) indexing data structure to maintain the mapping between a cached HDD sector and its location on the caching device. The management of such indexing data structure could incur non-negligible CPU and memory overhead. (2) *Impact of HDD data compression*: Because of the very low IOPS and long access latency of HDDs, modern computing systems can apply compression over HDDs without noticeable CPU and performance overheads. HDD data compression typically operates with a relatively large chunk size (e.g., 32KB, 64KB, or even 256KB) to improve the total storage cost saving. As a result, if the HDD data compression is handled by the upper-level software (e.g., file system or user applications), data will be cached in the unit of compressed data chunks. This could degrade the caching device capacity usage efficiency, especially under workloads dominated by small-size random data access. Moreover, accessing compressed data chunk on the caching device involves fetching a large data chunk and carrying out CPU-based decompression. This could result in significant under-utilization of the IOPS and bandwidth performance of the flash-based caching device.

To address the above two issues, this paper presents a design framework of block-level TC-CSD-cached storage systems with the following two features: (1) *Index-less cache*

management: Leveraging the sparse LBA space of TC-CSD, we can manage the cache using a simple bitmap other than conventional hash/tree-based indexing data structures. This could largely reduce the cache management complexity and CPU/memory cost. We also developed techniques that can further reduce the bitmap memory footprint and leverage the bitmap data structure to make cache eviction more HDD-friendly. (2) *Transparent compression over HDDs*: Regarding the second issue (i.e., impact of HDD data compression), the most fundamental solution is to make HDD data compression occur beneath the block-level caching layer. Hence, both HDD data compression and caching are transparent to file systems and user applications, and block-level caching operates over the uncompressed storage space. As a result, HDD-resident data are compressed with relatively large chunk size, while data are cached in the unit of 4KB sectors. This will allow systems to fully utilize the IOPS and bandwidth of the caching device. Leveraging the sparse sector content enabled by TC-CSD, we developed a simple data structure to manage the realization of HDD data compression. It supports workload-adaptive variable compression chunk size, and can completely avoid read-modify-write operations in the presence random updates on HDD-resident data.

II. Background and Motivation

A. Basics of TC-CSD

Loosely speaking, any data storage device that can carry out data processing tasks beyond its core storage function can be called a computational storage drive. The simple concept of empowering storage devices with additional computing capability can trace back to over 20 years ago [3]–[5]. Computational storage complements with CPU to form a heterogeneous computing system. Compared with its CPU-only counterpart, a heterogeneous computing system not surprisingly can achieve higher performance and/or energy efficiency for many applications, as demonstrated by prior research (e.g., see [6], [9]–[12]). This work focuses on computational storage drive with built-in transparent compression (TC-CSD) [14], [15], which can be seamlessly deployed in today's computing infrastructure without any changes to existing software stack.

Fig. 1 illustrates the structure of TC-CSD: Inside the drive controller chip, data compression and decompression are carried out directly on the IO path by dedicated hardware engine, and the FTL (flash translation layer) manages the mapping/indexing of all the variable-length compressed data blocks. TC-CSD enables the following two types of *sparsity*:

LBA space sparsity: Let C_P denote the total physical capacity of NAND flash memory inside the drive, and C_L denote the capacity of LBA storage space being exposed by the drive. To enable host materialize the benefit of transparent compression, as illustrated in Fig. 2(a), storage drive must be able to expose a sparse LBA space, i.e., C_L is much larger than C_P. This is conceptually similar to the concept of thin provisioning.

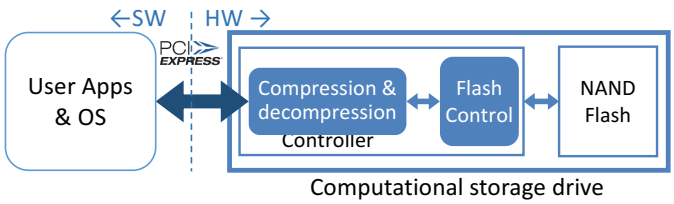

Fig. 1. Illustration of a CSD with built-in transparent compression.

Sector content sparsity: Since special data patterns such as all-zero and all-one can be highly compressed, we can leave one 4KB sector partially filled with valid user data without sacrificing the true physical storage cost, as illustrated in Fig. 2(b). This suggests that we can intentionally sparsify the content of each 4KB sector without sacrificing the true physical storage cost. This work aims to explore the above two types of sparsity to improve the implementation efficiency of flash-cached storage systems that use TC-CSD as the caching device.

B. Flash-Cached Storage Systems

Given the IO access locality in most real-world workloads and the significant performance/cost difference between the NAND flash memory and magnetic recording technologies, integrating a flash-based cache into HDD storage systems is a natural option to economically improve the storage system performance. Meanwhile, because of the very low IOPS and long access latency of HDDs, modern computing systems can realize data compression over HDDs to reduce the total storage cost without incurring noticeable CPU and performance overheads. Moreover, it is not uncommon that data compression over HDDs can even improve the storage system performance.

This work is interested in deploying/managing the flash-based cache at the block layer, being transparent to the upper-level file systems and user applications. Such block-level caching (e.g., Open CAS [16], FlashCache [17], and Bcache [18]) can be conveniently deployed without demanding any changes to the upper-level software stack. However, in return for its deployment convenience, block-level caching is subject to the following two implementation issues:

Cache indexing overhead: Let $\mathbb{L}_c$ and $\mathbb{L}_b$ denote the LBA space of the caching device and the back-end hard HDDs, and let $\mathbb{L}_b^{(c)} \subset \mathbb{L}_b$ denote the cached data set. In conventional implementation of a cached storage system, one must use an indexing data structure (e.g., hash-based or tree-based) to maintain the mapping between $\mathbb{L}_b^{(c)}$ and $\mathbb{L}_c$. Since block-level caching manages cache in the unit of 4KB sectors, its indexing data structure must maintain the mapping between each cached sector $LBA_b \in \mathbb{L}_b^{(c)}$ and its location on the caching device $LBA_c \in \mathbb{L}_c$. As a result, the cache indexing complexity is linearly proportional to the cache storage capacity. This can lead to high indexing CPU/memory cost for large-scale storage systems. For example, to deploy an 8TB flash-based cache over 256TB HDDs, block-level caching needs to maintain

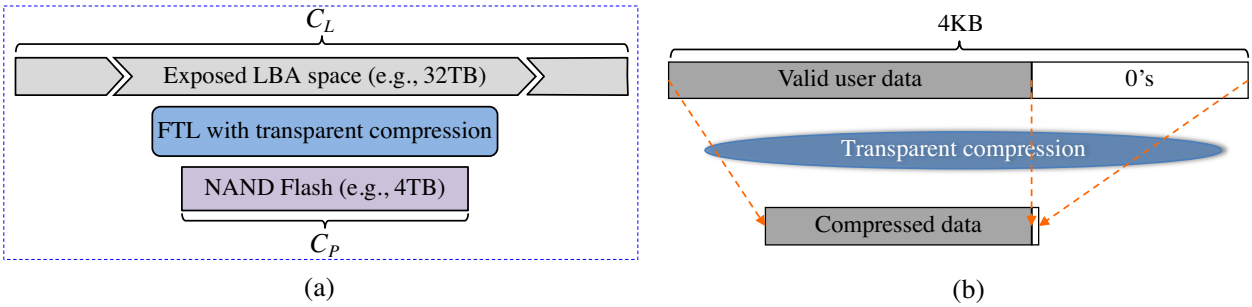

Fig. 2. Illustration of (a) LBA space sparsity, and (b) intra-sector content sparsity, which are both enabled by TC-CSD.

all the indexes through an in-memory key-value store with 2 billion entries (i.e., 8TB/4KB). Assuming 16 bytes per entry, the indexing could consume 32GB memory. Moreover, management of 2 billion key-value pairs could incur non-negligible CPU overhead.

Impact of HDD data compression: Because of the very low IOPS and long access latency of HDDs, modern servers can readily apply coarse-grained data compression over HDDs (e.g., 32KB, 64KB, or even 256KB per compression chunk) without noticeable CPU and performance overheads. In fact, it is not uncommon that data compression over HDDs can even improve the storage system performance. However, if HDD data compression is handled at the upper level (e.g., file system and user applications), data will be cached in the unit of compressed data chunks. This could degrade the caching device capacity usage efficiency, especially under workloads dominated by small-size random data access. Moreover, accessing compressed data chunk on the caching device involves fetching a large data chunk and carrying out CPU-based decompression. This could result in significant under-utilization of the IOPS and bandwidth performance of the flash-based caching device.

III. Proposed Design Solution

This section presents a TC-CSD-based block-level caching design framework, where the key is to address the above two implementation issues by leveraging the two types of sparsity of TC-CSD (i.e., LBA sparsity, and sector content sparsity). We will first introduce an index-less cache management approach that leverages the sparse LBA space of TC-CSD to largely simplify the cache management. Then we will present a data structure to strongly simplify LBA-PBA address translation by leveraging the sparsity of TC-CSD.

A. Index-less Cache Management

The key idea is to make TC-CSD caching device expose an LBA space that is identical to the LBA space of the back-end HDDs. As a result, for any cached sector, its LBA address on the back-end HDDs is directly used as its LBA address on the TC-CSD caching device. In conventional design practice, the LBA space $\mathbb{L}_c$ of the caching device is much smaller than the LBA space $\mathbb{L}_b$ of the back-end hard HDDs (i.e., $|\mathbb{L}_c| << |\mathbb{L}_b|$. As a result, we must deploy an indexing data structure (i.e., a tree/hash-based key-value store) to explicitly maintain the mapping between the on-HDD LBA of a cached sector and its location on the caching device. In contrast, the TC-CSD caching device exposes an LBA space that is identical to the LBA space of the back-end HDDs. For any cached sector with on-HDD LBA $L_i \in \mathbb{L}_b$, the LBA address of its cached version on the TC-CSD caching device is also L_i. This obviates the implementation of an indexing data structure.

By leveraging the sparse LBA space of TC-CSD, we propose a data structure based on bitmap that can largely simplify the cache management compared with the conventional data structure for cache management. The key idea to design the cache bitmap is second-chance scan. In the bitmap, each entry represents a sector. Each entry has four bits. The first bit denotes whether this sector is in the cache (1) or not (0). The second bit denotes whether this sector is clean (0) or dirty (1). The third and fourth bits are used to denote whether the data are hot (01,10,11) or cold (00). When a sector is accessed, its last two bits add 1 unless they are 11. For example, if a sector whose last bits are 01 is accessed, then its last bits become 10. When the amount of data in cache reaches a predefined threshold, some data need to be evicted from cache. The bitmap will be scanned to determine which sectors should be evicted. After a sector is scanned, its last two bits subtract 1 unless they are 00. These data have a higher priority to be evicted: 1) clean data; 2) cold data; 3) dirty data with good continuity. The reason why we consider continuity as one of the factors is that if data is compressed when written to disk, then data would have to be written sequentially on disk rather than be updated in place, data with poor continuity on disk would have a poor compression rate, and cause long latency because disk may need to fseek many locations to finish a read request with large size.

To better illustrate our cache design, we assume the logic capacity is 2TB and the cache capacity is 200GB. We also assume that a certain amount of data (between 1GB and 2GB) should be evicted from cache when the amount of data in cache reaches a predefined threshold. Firstly, we need to evict the cold and clean data from cache. And if the amount of

these cold and clean data is less than 1GB, then we need to find and evict 1GB of cold and dirty data with relatively good continuity. Thus the total amount of evicted data is between 1GB and 2GB. To be more specific, our design is illustrated as follows:

We generate a table with 2000 entries. Each entry represents 100 MB dirty and cold data in cache, and they are ranked in LBA. Each entry records two values: $FirstLBA_{dc}$(the first LBA of dirty and cold data) and $Cont_{dc}$ (the continuity of dirty and cold data)

Initialize each entry's two values above as 0. We need to scan the bitmap to determine what data should be evicted from cache, and here are the steps:

1) Use $Size_{cc}$ to record the size of clean and cold data (i.e., data whose bits are 1000) and initialize it as 0. Use $Size_{dc}$ to record the size of dirty and cold data (i.e., data whose bits are 1100) and initialize it as 0. For the first entry, scanning from the minimum LBA (sector), and use the first LBA whose bits are 1100 (i.e., dirty and cold) as the value of the entry's $FirstLBA_{dc}$.

2) If the current sector's bits are 1000, remove this sector's data from cache, change its bits to 0000, and add 4096B to $Size_{cc}$. If $Size_{cc}$ is less than 1GB then move to the next sector. Otherwise, stop scanning, skipped the following steps and the data evicting work is over.

3) While if the current sector's bits are 1100, add 4096B to $Size_{dc}$. If the current sector's bits and its next sector's bits are both 1100, then add 1 to $Cont_{dc}$. And it's easy to see that if m continuous sectors' bits are all 1100, then add m*4096B to $Size_{dc}$ and add m-1 to $Cont_{dc}$.

4) Do the same procedure for the next sectors until $Size_{dc}$ equals to 100MB, then we get the table's first entry's two values: $FirstLBA_{dc}$ and $Cont_{dc}$. Then set $Size_{dc}$ as 0 again and we move to the next sector and similarly we can get the table's other entries' $FirstLBA_{dc}$ and $Cont_{dc}$. Assume the last entry is the nth entry, n is generally much less than 2000 though the table is allocated up to 2000 entries, because the 100MB data that each entry represents are only dirty and cold data.

5) Rank the n entries according to its value of $Cont_{dc}$, then we choose 10 entries with the largest $Cont_{dc}$. We can find these dirty and cold data by scanning the bitmap again via each entry's $FirstLBA_{dc}$ and then find these data's location in cache by leveraging the sparse LBA space of TC-CSD, then we evict and write these data to disk. Since each entry represents 100MB of dirty and cold data, finally we evict 1GB dirty and cold data with good continuity and less than 1GB clean and cold data from cache.

6) Usually the data evicting work could be done after the 5 steps above. But we need to consider the situation when the amount of cold data (no matter clean or dirty) is less than 1 GB. If this case happens, we can record the information of clean and relatively cold data (bits are 1001) and the information of dirty and relatively cold data (bits are 1101) during the second scanning with the similar steps above. We can also always record this information during the first scan to reduce the total times of scanning.

What should be pointed out is that the evicted 1GB of dirty and cold data have very good continuity but perhaps not the best continuity. However, to find the 1GB dirty and cold data with the best continuity, a complex data structure should be maintained. We need to record each continuous dirty and cold data including the single dirty and cold sector if both of its previous and next sectors are not dirty and cold, and finally rank them according to continuity to find the 1GB dirty and cold data with the best continuity. This would no doubt cost much higher storage and computational resources. Furthermore, we design experiments to compare our algorithm with this ideal algorithm in Section IV, and the results show that the difference between the two algorithms' performances is negligible while our design costs much less storage and negligible computational resources.

Since we use TC-CSD as cache and TC-CSD internally performs hardware-based per-4KB data compression/decompression, thus we can further compress our bitmap in cache. Compared to the whole logic LBAs, only a small part of LBAs is in cache. So most bits of the bitmap are continuous zeros. That means our bitmap could be compressed to cost negligible storage resources. We can separate bitmap to many parts and then compress them. For each read or write operation, we only need to decompress, update and compress the related part. Since it's done in cache, the latency it costs is negligible compared read or write latency of disk.

For the system that only supports fixed-size compression, our cache design also applies to it by adjusting the length of the evicted data. For example, if the compression's fixed-size is 16kB (4 sectors), and assuming LBAs from 7 to 101 is one of the dirty data we choose to evict from the cache management algorithm, instead of evicting all of these data, we just evict totally 92 LBAs (from 8 to 99). That is, always make sure the first LBA to evict is the multiple of 4 sectors and the last LBA to evict is the multiple of 4 sectors minus 1. By doing this, it could also strongly mitigate read-modify-write and write amplification caused by fixed-size compression. We carried out related experiments in Section IV to further discuss it.

B. LBA-PBA Address Translation

Apart from strongly simplifying the cache management compared to the conventional ones, we also propose a translation design that strongly simplifies the data structure of LBA-PBA address translation by further leveraging the fact that TC-CSD internally performs hardware-based per-4KB data compression/decompression. Conventional LBA-PBA address translation maintains a Hash table or tree structure that costs large storage resources. And it becomes even more complex and costs more computational resources if data is compressed when being written to disk. Data compression on the disk can save space, but may cause long latency if the size of data

compression is fixed because fixed-size compression can cause lots of read-modify-write when part of data in the compression unit needs to be updated. We propose a translation design that avoids read-modify-write by using a very flexible data structure that allows data compression of any size. Our translation design is illustrated as follows.

1) We use a 4kB sector to store the information of n LBA-PBA mapping, n is fixed and these n LBAs are continuous.

2) If all of these n continuous LBAs are compressed together and then written to disk, then we just record the first LBA (assuming it is 1), the first related PBA (assuming it is 101) and the length l (l is the length after compression).

In this case, except the 12 bytes of information, all of the other bits of the 4kB sector are zeros. Because TC-CSD internally performs hardware-based per-4KB data compression/decompression, we can compress the 4kB to only several bytes. Thus the information of n LBA-PBA mapping costs negligible storage resources in fact. If a segment of the LBAs (from LBA m_1 to LBA m_2) are updated and compressed to a new location (assuming the new location's first PBA is 999 and assuming the length of the new compressed data is l_1), Then the n LBAs' mapping information is recorded as Table I shows: In Table I, the first column records the first LBA of

TABLE I
MAPPING INFORMATION.

1	m_1-1	1	101	l
m_1	m_2-m_1+1	m_1	999	l_1
m_2+1	n-m_2	1	101	l

the continuous LBAs that belong to one compressed data. The second column records the length of continuous LBAs that belong to one compressed data. The third column records the first LBA of the compressed data. The fourth column records the first PBA of the compressed data. And the fifth column records the length of the compressed data. This table is simple but supplies all information for the LBA-PBA address mapping. For example, if we want to read data whose LBAs are from m_2+1 to n, by checking the table, we can find and read the data whose first PBA is 101 with the length of l. After decompressing the data, we can get the data from the difference between the requested first LBA and the first LBA of the compressed data.

3) In most cases, n continuous LBAs are compressed not together but separately and are written down in many different locations on disk, so we need to keep each compressed data's mapping and update information as step 2 introduces. Because our cache management evicts dirty data with good continuity, so the storage space to record the mapping information is small. That means in the 4kB sector, there is much free space whose bits are all zeros, resulting in negligible storage cost after compression.

4) In the extreme case when each two adjacent LBAs' PBAs are not adjacent with each other, we need to record each LBA's PBA. Thus when we choose the value of n, we need to make sure that 4kB can record all of the n LBA-PBA mapping information in the worst case. However, our cache management ensures that the dirty data written down to disk have very good continuity, so this extreme situation hardly happens.

When a segment of LBA-continuous dirty data is chosen to be evicted from cache, it is compressed as a whole and then written to disk, and then we need to decompress, update and compress its related 4kB sector that records its LBA-PBA mapping information. Each time when data is updated, we also update the garbage rate of the zone where the invalid PBAs belong to, for the purpose of garbage collection.

IV. EVALUATION

A. Experimental Setup

We ran all the experiments on a server with a 24-core 2.6GHz Intel CPU, 64GB DDR4 DRAM, 10 2TB 7200rpm SATA HDDs, and a 3.2TB SSD with built-in transparent compression that was recently launched to the commercial market by ScaleFlux [13]. The server runs Linux Kernel 4.10.0 in the Ubuntu 16.04.03 distribution. This SSD carries out hardware-based zlib compression on each 4KB sector along the IO path. Operating with PCIe Gen3$\times$4 interface, this SSD can achieve 3.2GB/s sequential throughput, and 650K (520K) random 4KB read (write) IOPS (IO per second) over 100% LBA span, which is similar to leading-edge commodity NVMe SSDs.

We used three traces (LUN0, LUN2 and LUN4) from Systor'17 Traces [19] and five benchmarks (Bayes, Kmeans, PageRank, Sort, and TeraSort) from the big data benchmark suite HiBench 7.0 [20]. 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. LUN0, LUN2 and LUN4 traces have a relatively smaller average size of read and write request compared to Bayes, Kmeans, PageRank, Sort and Terasort traces.

B. Write Amplification

We first carried out experiments to evaluate how fixed-size compression could affect write amplification directly (i.e., without any cache). Fig. 3 shows the write amplification under different traces when the compression unit is 16kB, 32kB, 64kB and 128kB respectively. From the results, we can conclude that fixed-size compression unit can cause severe write amplification especially when the compression unit is large. As the compression unit increases from 16kB to 128kB, the write amplification becomes more severe for all traces, especially for LUN0, LUN2 and LUN4 traces because their requests' average size is relatively smaller. When the compression Unit is 128kB, the write amplification of LUN0, LUN2 and LUN4 can reach up to almost 7, which is a very severe problem. However, this problem can be mitigated via our cache management.

To evaluate our cache design's write amplification when compression size is fixed, we used LRU (list recently used)

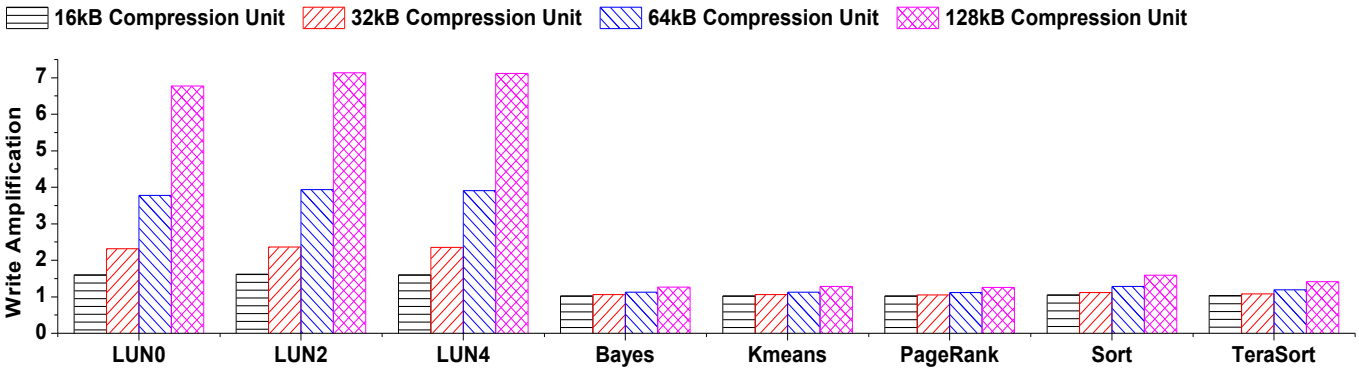

Fig. 3. Measured average write amplification with different fixed-size compression unit

cache as the baseline. Furthermore, we implemented an improved LRU algorithm that half of the evicted data are chosen according to the conventional LRU cache algorithm, while the other half of the evicted data are dirty and cold data that have good continuity, chosen from the rest data in cache. To get the dirty data with good continuity, we maintain a tree data structure, which no doubt makes the improved LRU cost more storage and computational resources. Also, we implemented not only our proposed cache management, but also implemented an ideal proposed cache management mentioned in Subsection III-A from it by maintaining a tree structure to get the dirty and cold data with the best continuity.

Fig. 4 shows measured average write amplification when using different policies with the fixed 32kB compression unit. The results show that the proposed cache management can mitigate write amplification to almost 1(1 means no write amplification), which is very close to the performance of the improved LRU and the ideal version of the proposed. Furthermore, the proposed cache management costs negligible storage space to maintain a bitmap that could be compressed. While the improved LRU and the ideal proposed cost much more storage and computational resources.

C. Read Performance

When the compression unit is not fixed, data could be compressed in any size and then are written on disk sequentially. This does not cause write amplification, but may affect read performance because a read request may fseek two or more times to get the complete data. We carried out experiments to evaluate the read performance of our proposed cache management when the size of the compression unit is not fixed. Apart from the four policies mentioned above, we added two more conventional policies: write through directly and write back in order. Write through directly is a policy that there is no buffer and when a request arrives, directly compress and write it to disk to ensure that data would not be lost. Write back in order is a policy to evict data from cache in the order of entering the cache, which also considers data's safety as a very important factor.

Fig. 5 shows the average number of head seek per read request when using different policies. The results tell that the improved LRU, our proposed cache management and the ideal proposed with the best continuity have the similar and lower average number of head seek, compared to the other policies. The reason is that these three policies evict data with good continuity, while continuity means more continuous LBAs could be compressed together and be written on the same location of disk. Thus it is a high chance that only one head seek is needed to get the complete data of the read request.

Fig. 6 and Fig. 7 show the average and 99-percentile tail read latency when using different policies, under different traces. From the results, we can conclude that the improved LRU, our proposed cache management and the ideal proposed with the best continuity have the very close read performance which is much better than the other three policies. The results well align with the number of head seek per read results shown in Fig. 5.

From the results of write amplification and read performance, we know that our proposed cache management is the best choice among all of the policies mentioned above. Because though the improved LRU policy and the ideal proposed policy have higher performance, the results show that the difference is negligible compared to our proposed cache management. While our cache management costs much less or even negligible resources.

V. Related Work

SSD Caching Aiming to improve mass data storage system performance at relatively small cost overhead, hybrid storage system with SSD caching has been widely studied (e.g., see [21]–[33]). A large body of prior work focused on cache management and eviction policies. For example, Kgil et al. [21] proposed to partition the cache into separate read and write regions to improve both performance and reliability. FlashTier [34] presented an integrated FTL and cache management design approach to reduce the SSD caching implementa-

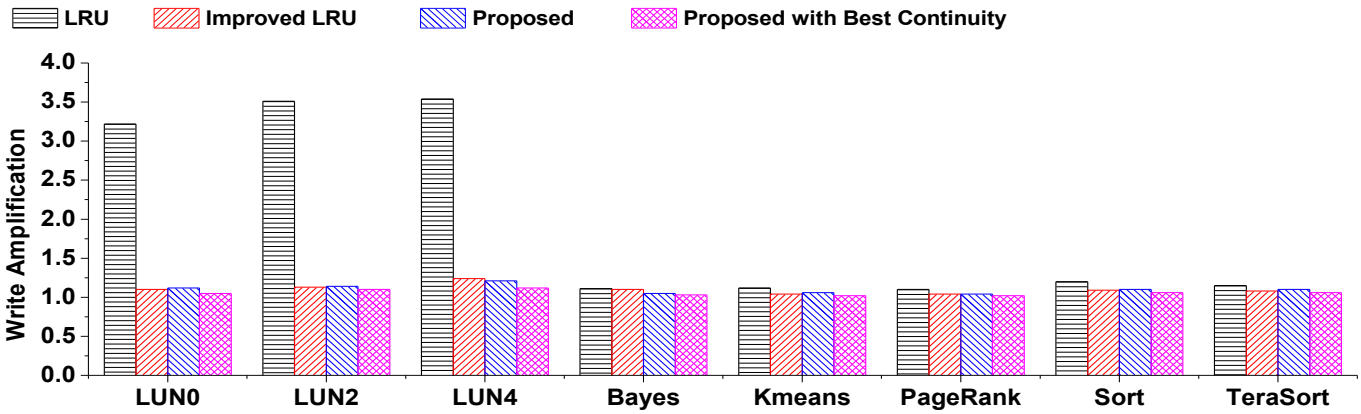

Fig. 4. Measured average write amplification when using different policies (fixed-size compression unit is 32kB)

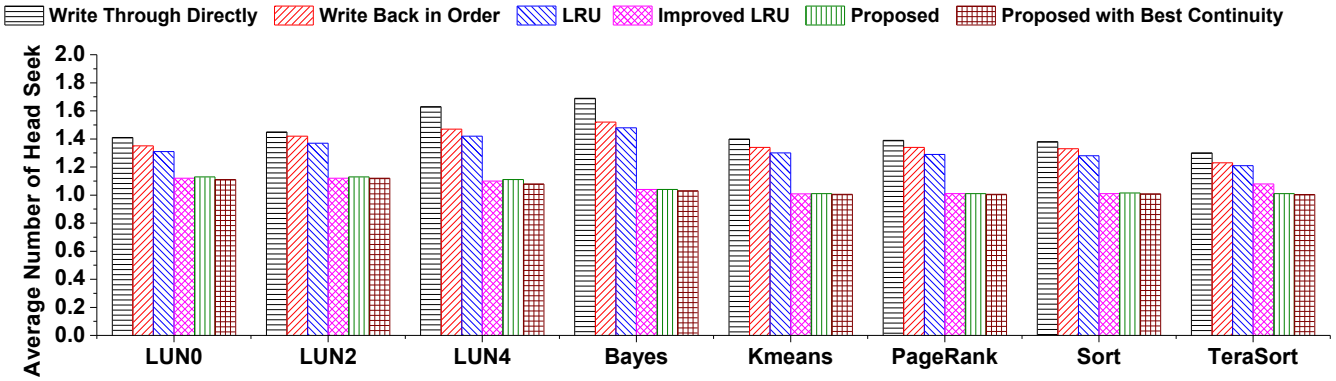

Fig. 5. Measured average number of head seek per read request when using different policies

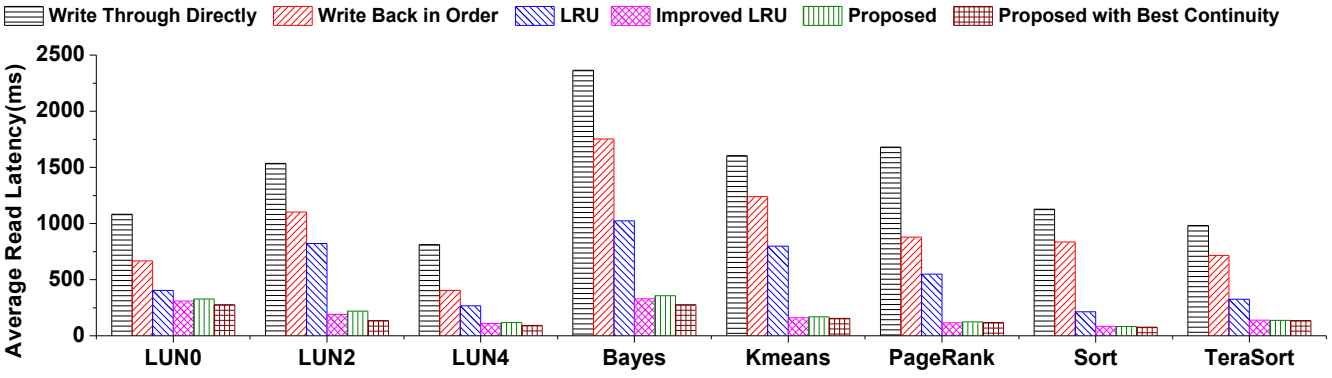

Fig. 6. Measured average read latency when using different policies

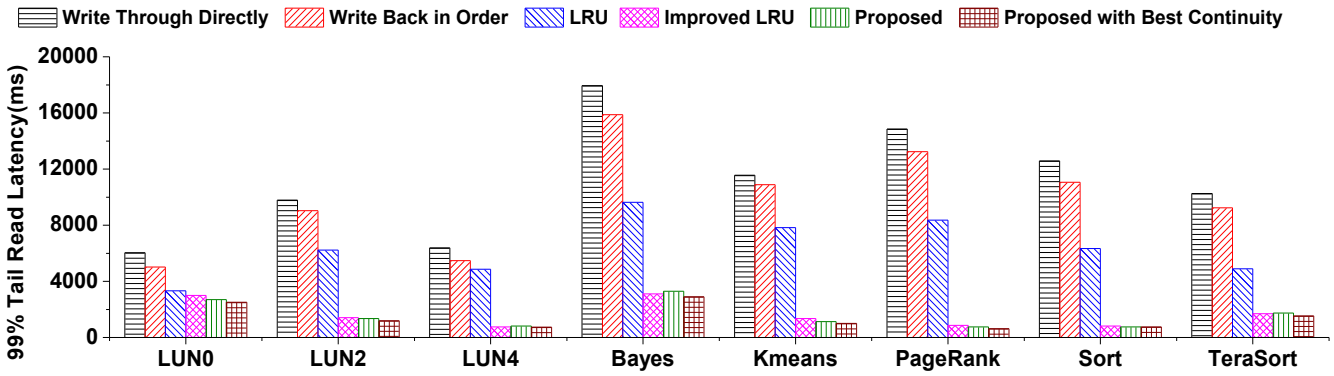

Fig. 7. Measured 99-percentile tail read latency when using different policies

tion complexity. Consistent write-back caching policies were developed in [28], [35]. Cheng et al. [31] developed offline algorithms that can simultaneously improve SSD cache hit ratio and endurance. Ni et al. [30] presented SSD cache eviction schemes geared towards data center workloads. ZoneTier [33] presented SSD-based caching/tiering schemes optimized for SMR drives. Huang et al. [32] developed an adaptive SSD cache data replacement policy.

Transparent Data Reduction. Prior research has well studied the implementation of storage data reduction (e.g., compression and deduplication) with complete transparency to user applications. Transparent data reduction can be realized at filesystem level [36]–[39], block level [40]–[42], and even inside storage hardware [43]–[46]. Prior work [40], [47]–[50] also well studied the potential of applying data reduction to enlarge the effective SSD cache capacity and hence improve cache hit rate. Providing block-level data compression and deduplication support, Linux VDO [41] caries out compression on the per-4KB basis in order to simplify the management at the penalty of compression ratio. Klonatos et al. [40] presents a block-level compression solution that allows coarse-grained compression (e.g., per-64KB) and relies on read-modify-write to handle random updates. Ajdari et al. [42] used FPGA-based accelerator to assist block-level compression and deduplication over large SSD arrays.

SSD Sparse Addressing. Prior work explored the innovation opportunities enabled by making SSD FTL expose a sparse logical address space. For example, FlashTier [34] utilizes SSD sparse addressing to simplify SSD cache management. FlashMap [51] integrates virtual address translation and SSD FTL sparse address translation to efficiently support memory-mapped SSD files. Das et al. [52] presented a solution that leverage SSD sparse addressing to facilitate application-level data compression. DFS filesystem [53] takes advantage of SSD sparse addressing to largely simplify its data management.

VI. Conclusions

With the goal of simplifying the data structure of cache design by leveraging the sparsity of TC-CSD, this paper advocates a index-less cache management approach. The key is to use a second-chance-scan bitmap to manage the data in cache to simplify the cache design, and compress the bitmap to save a lot of storage space by leveraging the property of TC-CSD. From the results, we can conclude that the proposed index-less cache has better performance than the conventional cache policies, and has a good performance as the improved cache policies that are much more complex and cost much larger resources.

References

[1] *SNIA Technical Work Group on Computational Storage*. https://www.snia.org/computational.

[2] T. Coughlin, "When Memory Starts to Think [The Art of Storage]," *IEEE Consumer Electronics Magazine*, vol. 8, no. 3, pp. 90–91, 2019.

[3] A. Acharya, M. Uysal, and J. Saltz, "Active disks: Programming model, algorithms and evaluation," in *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998, pp. 81–91.

[4] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent RAM," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, Mar 1997.

[5] E. Riedel, G. A. Gibson, and C. Faloutsos, "Active storage for large-scale data mining and multimedia," in *Proc. of the International Conference on Very Large Data Bases (VLDB)*, 1998, pp. 62–73.

[6] W. Cao, Y. Liu, Z. Cheng, N. Zheng, W. Li, W. Wu, L. Ouyang, P. Wang, Y. Wang, R. Kuan *et al.*, "POLARDB meets computational storage: Efficiently support analytical workloads in cloud-native relational database," in *USENIX Conference on File and Storage Technologies (FAST)*, 2020, pp. 29–41.

[7] I. L. Picoli, P. Bonnet, and P. Tözün, "LSM management on computational storage," in *Proceedings of the International Workshop on Data Management on New Hardware*, 2019, pp. 1–3.

[8] S. Cho, C. Park, H. Oh, S. Kim, Y. Yi, and G. R. Ganger, "Active disk meets flash: A case for intelligent SSDs," in *Proc. of the International ACM Conference on Supercomputing*, 2013, pp. 91–102.

[9] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query processing on smart SSDs: Opportunities and challenges," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2013, pp. 1221–1230.

[10] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind, "BlueDBM: An appliance for big data analytics," in *Proc. of the International Symposium on Computer Architecture (ISCA)*, 2015, pp. 1–13.

[11] Y. Kang, Y.-S. Kee, E. Miller, and C. Park, "Enabling cost-effective data processing with smart SSD," in *Proc. of IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, May 2013, pp. 1–12.

[12] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson, "Willow: A user-programmable SSD," in *Proc. of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 67–80.

[13] ScaleFlux Computational Storage. http://scaleflux.com.

[14] E. F. Haratsch, "SSD with Compression: Implementation, Interface and Use Case," in *Flash Memory Summit*, 2019.

[15] N. Zheng, X. Chen, J. Li, Q. Wu, Y. Liu, Y. Peng, F. Sun, H. Zhong, and T. Zhang, "Re-think data management software design upon the arrival of storage hardware with built-in transparent compression," in *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[16] Open Cache Acceleration Software (Open CAS). https://open-cas.github.io.

[17] FlashCache. https://github.com/facebookarchive/flashcache.

[18] Bcache. https://bcache.evilpiepirate.org.

[19] 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.

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

[21] T. Kgil, D. Roberts, and T. Mudge, "Improving nand flash based disk caches," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*. IEEE, 2008, pp. 327–338.

[22] T. Pritchett and M. Thottethodi, "Sievestore: a highly-selective, ensemble-level disk cache for cost-performance," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2010, pp. 163–174.

[23] Y. Klonatos, T. Makatos, M. Marazakis, M. D. Flouris, and A. Bilas, "Azor: Using two-level block selection to improve SSD-based I/O caches," in *IEEE International Conference on Networking, Architecture, and Storage*. IEEE, 2011, pp. 309–318.

[24] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer, "Mercury: Host-side flash caching for the data center," in *IEEE symposium on mass storage systems and technologies (MSST)*. IEEE, 2012, pp. 1–12.

[25] D. A. Holland, E. Angelino, G. Wald, and M. I. Seltzer, "Flash caching on the storage client," in *USENIX Annual Technical Conference (ATC)*, 2013, pp. 127–138.

[26] D. Arteaga and M. Zhao, "Client-side flash caching for cloud systems," in *Proceedings of International Conference on Systems and Storage*, 2014, pp. 1–11.

[27] Y. Oh, J. Choi, D. Lee, and S. H. Noh, "Improving performance and lifetime of the SSD RAID-based host cache through a log-structured approach," *ACM SIGOPS Operating Systems Review*, vol. 48, no. 1, pp. 90–97, 2014.

[28] D. Qin, A. D. Brown, and A. Goel, "Reliable writeback for client-side flash caches," in *USENIX Annual Technical Conference (ATC)*, 2014, pp. 451–462.

[29] Y. Chai, Z. Du, X. Qin, and D. A. Bader, "Wec: Improving durability of ssd cache drives by caching write-efficient data," *IEEE Transactions on computers*, vol. 64, no. 11, pp. 3304–3316, 2015.

[30] Y. Ni, J. Jiang, D. Jiang, X. Ma, J. Xiong, and Y. Wang, "S-RAC: SSD friendly caching for data center workloads," in *Proceedings of the ACM International on Systems and Storage Conference*, 2016, pp. 1–12.

[31] Y. Cheng, F. Douglis, P. Shilane, G. Wallace, P. Desnoyers, and K. Li, "Erasing belady's limitations: In search of flash cache offline optimality," in *USENIX Annual Technical Conference (ATC)*, 2016, pp. 379–392.

[32] S. Huang, Q. Wei, D. Feng, J. Chen, and C. Chen, "Improving flash-based disk cache with lazy adaptive replacement," *ACM Transactions on Storage (TOS)*, vol. 12, no. 2, pp. 1–24, 2016.

[33] X. Xie, L. Xiao, and D. H. Du, "Zonetier: A zone-based storage tiering and caching co-design to integrate ssds with smr drives," *ACM Transactions on Storage (TOS)*, vol. 15, no. 3, pp. 1–25, 2019.

[34] M. Saxena, M. M. Swift, and Y. Zhang, "Flashtier: a lightweight, consistent and durable storage cache," in *Proceedings of the ACM European conference on Computer Systems*, 2012, pp. 267–280.

[35] R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao, "Write policies for host-side flash caches," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2013, pp. 45–58.

[36] M. Burrows, C. Jerian, B. Lampson, and T. Mann, "On-line data compression in a log-structured file system," *ACM SIGPLAN Notices*, vol. 27, no. 9, pp. 2–9, 1992.

[37] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum, "The zettabyte file system," in *Proceedings of the Usenix Conference on File and Storage Technologies (FAST)*, vol. 215, 2003.

[38] K. Srinivasan, T. Bisson, G. R. Goodson, and K. Voruganti, "iDedup: latency-aware, inline data deduplication for primary storage." in *USENIX Conference on File and Storage Technologies (FAST)*, 2012, pp. 1–14.

[39] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-tree filesystem," *ACM Transactions on Storage (TOS)*, vol. 9, no. 3, pp. 1–32, 2013.

[40] Y. Klonatos, T. Makatos, M. Marazakis, M. D. Flouris, and A. Bilas, "Transparent online storage compression at the block-level," *ACM Transactions on Storage (TOS)*, vol. 8, no. 2, pp. 1–33, 2012.

[41] Virtual Data Optimizer (VDO). https://github.com/dm-vdo/vdo.

[42] M. Ajdari, P. Park, J. Kim, D. Kwon, and J. Kim, "CIDR: A cost-effective in-line data reduction system for terabit-per-second scale SSD arrays," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 28–41.

[43] F. Chen, T. Luo, and X. Zhang, "CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives." in *USENIX Conference on File and Storage Technologies (FAST)*, 2011, pp. 77–90.

[44] G. Wu and X. He, "Delta-FTL: improving SSD lifetime via exploiting content locality," in *Proceedings of the ACM european conference on Computer Systems*, 2012, pp. 253–266.

[45] A. Zuck, S. Toledo, D. Sotnikov, and D. Harnik, "Compression and SSDs: Where and how?" in *Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW)*, 2014.

[46] X. Chen, Y. Li, J. Hao, H. Shin, M. Suh, and T. Zhang, "Simultaneously reducing cost and improving performance of NVM-based block devices via transparent data compression," in *Proceedings of the International Symposium on Memory Systems*, 2019, pp. 331–341.

[47] T. Makatos, Y. Klonatos, M. Marazakis, M. D. Flouris, and A. Bilas, "Using transparent compression to improve ssd-based I/O caches," in *Proceedings of the European conference on Computer systems*, 2010, pp. 1–14.

[48] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace, "Nitro: A capacity-optimized {SSD} cache for primary storage," in *USENIX Annual Technical Conference (ATC)*, 2014, pp. 501–512.

[49] Q. Wang, J. Li, W. Xia, E. Kruus, B. Debnath, and P. P. Lee, "Austere flash caching with deduplication and compression," in *USENIX Annual Technical Conference (ATC)*, 2020, pp. 713–726.

[50] Y. Jia, Z. Shao, and F. Chen, "Slimcache: An efficient data compression scheme for flash-based key-value caching," *ACM Transactions on Storage (TOS)*, vol. 16, no. 2, pp. 1–34, 2020.

[51] J. Huang, A. Badam, M. K. Qureshi, and K. Schwan, "Unified address translation for memory-mapped SSDs with FlashMap," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015, pp. 580–591.

[52] D. Das, D. Arteaga, N. Talagala, T. Mathiasen, and J. Lindström, "NVM compression—hybrid flash-aware application level compression," in *Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW)*, 2014.

[53] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn, "Dfs: A file system for virtualized flash storage," *ACM Transactions on Storage (TOS)*, vol. 6, no. 3, pp. 1–25, 2010.