2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)

2025 IEEE International Symposium on High Performance Computer Architecture (HPCA) | 979-8-3315-0647-6/25/$31.00 ©2025 IEEE | DOI: 10.1109/HPCA61900.2025.00053

Reviving In-Storage Hardware Compression on ZNS SSDs through Host-SSD Collaboration

Yingjia Wang†, Tao Lu‡✉, Yuhong Liang†, Xiang Chen‡, and Ming-Chang Yang†✉
†The Chinese University of Hong Kong ‡DapuStor Corporation

***Abstract*—Zoned Namespace (ZNS) is an emerging SSD interface with great potential for performance and cost in large-scale cloud SSD deployments. Enabling in-storage hardware compression on ZNS SSDs is promising for further enhancing the cost-effectiveness of ZNS-based storage infrastructures. However, based on our investigation, existing solutions based on the host-transparent methodology are sub-optimal on ZNS SSDs due to two intrinsic challenges: (1) locating compressed chunks and (2) harvesting space savings from compression.**

In this paper, we for the first time revisit the compression storage system architecture on the emerging ZNS SSD and propose to decouple compression execution with indexing. We propose CCZNS (CC: collaborative compression), an advanced ZNS interface that revives in-storage hardware compression on ZNS SSDs through a novel host-SSD collaborative approach. We present how CCZNS can benefit host software by performing a case study on RocksDB and ZenFS. Extensive experiments demonstrate that the CCZNS-based storage system significantly outperforms existing system solutions.

I. Introduction

Zoned Namespace (ZNS) is an emerging SSD interface that eliminates the block interface tax in terms of performance and cost [20], [30]. Logically, ZNS divides the storage space into *zones*, each of which can be read randomly but only written *sequentially*. Each zone is associated with a set of physical flash blocks and thus can not be overwritten until explicitly reset by the host. On the one hand, ZNS naturally couples logical and physical addresses in a zone, avoiding the fine-grained page-level mapping overhead. On the other hand, ZNS eliminates device-level garbage collection, and host software can holistically optimize data placement and reclamation, leading to better system performance and reduced over-provisioned flash space. As the global data volume rapidly explodes, ZNS SSD is a promising low-cost storage solution scaling for the zettabyte data era and has demonstrated the potential in large-scale cloud environments [29], [103].

With the same goal of lowering system costs, *data compression* is a crucial technique for modern storage systems. However, host-side compression solutions (i.e., using host CPUs or external/on-chip hardware accelerators) face challenges in performance. Performing compression by host CPUs can significantly degrade system performance while consuming excessive CPU resources [62], [69]. On the other hand, despite the higher compression performance, external/on-chip hardware accelerators exhibit unsatisfactory performance when data is (de)compressed in small chunks due to round-trip data movements via the PCIe interface.

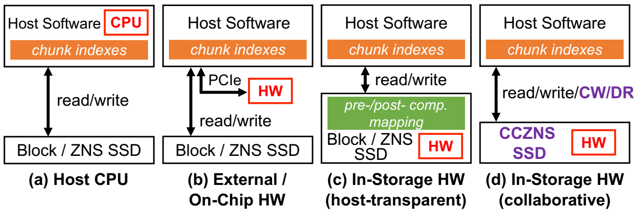
The emerging *in-storage hardware compression*, which has gained industry support [8], [14], [16], can well address the challenges mentioned above. FPGA- or ASIC-based hardware compression engines are integrated into the SSD, shifting the compression execution from the host to the SSD itself. This delivers a boosted compression performance (which can also scale with the number of SSDs), significantly relieves host CPU usage, and eliminates the data round-trip overhead between external/on-chip accelerators. Our commercially deployed ASIC-based compression engines can achieve compression and decompression throughput of up to 11GB/s and 14GB/s, respectively, with only about 5% extra area size on the SSD controller.

Given these merits, enabling in-storage hardware compression on ZNS SSDs is promising for augmenting cost-efficient ZNS-based storage infrastructures. Nevertheless, potential solutions oriented from the existing *host-transparent* methodology are sub-optimal on ZNS SSDs due to two intrinsic challenges: (1) locating compressed chunks and (2) harvesting space savings from compression (see §II-C). The state-of-the-art host-transparent attempt [93] significantly sacrifices write amplification and write performance, defeating the potential of in-storage hardware compression in the first place.

In this paper, we revisit the compression-enabled storage system architecture on the emerging ZNS SSD. Our key insight is that, *given the coupled logical and physical addresses within each zone, host software can leverage existing fine-grained indexes to directly manage compressed data inside the SSD*, which can well resolve the aforementioned challenges. However, since compression execution and indexing are now decoupled into opposite sides, there exist two key obstacles to such cross-layer management. The first is to maintain a uniform view of data, because in-storage compression alters both data addresses and sizes, which can not be automatically synchronized to the host. The second is to bypass the logical block granularity (e.g., 4KB) entrenched in the I/O stack and achieve high space savings as the host-side compression solutions (i.e., at byte granularity).

We propose *CCZNS* (CC: collaborative compression), an advanced ZNS interface that revives in-storage hardware compression through a novel host-SSD collaborative approach. Unlike existing compression-enabled storage system architectures, CCZNS offloads only compression execution to the SSD but responds the post-compression information to the host for index maintenance (see Figure 1). CCZNS performs cross-

979-8-3315-0647-6/25/$31.00 ©2025 IEEE
DOI 10.1109/HPCA61900.2025.00053



(a) Host CPU (b) External / On-Chip HW (c) In-Storage HW (host-transparent) (d) In-Storage HW (collaborative)

Fig. 1. **Existing and our proposed compression-enabled storage system architectures.**

layer management through a set of CC-specific read/write commands (i.e., CW and DR), integrating compression offloading and information responding into traditional read/write commands through the NVMe metadata field. To maintain a uniform data view between host and SSD, we implement bi-directional NVMe metadata transfer in the NVMe driver of the Linux kernel, so that host software can extract the returned information from the SSD and update the indexes accordingly (see §III-B). To harvest the space saved by compression maximally, CCZNS revamps both CC-specific commands and storage management as byte-addressable (see §III-D). CCZNS is also carefully refined to collaborate with host software with rich and diverse characteristics, such as different compression granularity (see §III-C) and compatibility requirements (see §III-E).

We present how CCZNS can benefit host software by performing a case study on RocksDB [13], the mainstream application of ZNS, and its filesystem backend ZenFS [21] (see §IV). We use both micro- and macro-benchmarks and diverse comparison groups (i.e., host CPU using different algorithms, ASIC-based external Intel®QAT card [18], and the state-of-the-art work [93]) to evaluate CCZNS comprehensively (see §V). Experiments on raw SSD access show that, due to hardware compression engines, the read and write throughput of CCZNS can exceed the theoretical flash bandwidth to up to 1.4× and 3.1×, respectively. Evaluation on RocksDB shows that the CCZNS-based storage system achieves the best-level write volume reduction and host CPU usage, as well as up to 3.6× and 1.3× throughput improvement in YCSB Load (write-only) and other macro-benchmarks, respectively. Compared to the state-of-the-art work BalloonZNS [93], CCZNS further reduces write volume by 12.6% while improving write throughput by 65.3%. We open-source the code for public use on GitHub[1].

II. Background and Motivation

A. *Flash-based SSD and Zoned Namespace (ZNS)*

Flash-based SSD adopts a multi-level architecture including channel, die, plane, block, and page. The flash die is the minimum unit of parallel operations while the flash page is the minimum unit of storage. To improve SSD firmware management efficiency, flash blocks are typically grouped into superblocks, each of which comprises flash blocks across many flash dies to fully leverage flash parallelism.

The block interface has been a widely used standard for flash-based SSDs. Despite providing a simple abstraction for users, it necessitates a burdensome flash translation layer (FTL) inside the SSD to hide the underlying flash erase-before-write characteristics. This leads to the well-known *block interface tax* [30], [31], [59], which is further exacerbated as the flash technology is evolving towards high-density forms [71], [88]. On the one hand, the logical-to-physical address mapping table requires large on-board DRAM (e.g., about 0.1% of the SSD capacity [53]), which considerably raises the hardware cost. On the other hand, the FTL performs background garbage collection (GC) to reclaim available space, which is the culprit of performance degradation and instability; therefore, block-interface SSD typically employs a non-trivial amount of flash over-provisioned (OP) space to counteract the effects of GC, which is reported to be up to 50% in Facebook's CacheLib [28].

Zoned Namespace (ZNS) is an emerging SSD interface that aims to avoid the block interface tax [20], [30]. ZNS divides the storage space into zones, each of which can only be written sequentially and is typically mapped to a flash superblock. Compared to the preceding Open-Channel interface [31], ZNS provides a moderate zone abstraction that not only supports cross-layer optimization (e.g., data placement, garbage collection, and I/O scheduling) but also facilitates easier adoption and software upkeep. ZNS eliminates the block interface tax in the following two aspects. First, due to the sequential write constraint, the logical and physical addresses within a zone are naturally coupled. Thus, ZNS SSD only requires a coarse-grained zone-level mapping table, which substantially lowers the on-board DRAM equipment (e.g., 10-20× smaller than that in traditional SSDs [27]). Second, ZNS shifts GC control upper to the host and eliminates the device-level GC. The host can then leverage software behaviors and workload characteristics for holistic management, leading to higher throughput, better performance predictability, and lower write amplification. As a result, ZNS can greatly reduce the amount of flash OP space [30], [48].

B. *Compression-Enabled Storage System Architectures*

Existing compression-enabled storage systems[2] adopt four representative architectures (i.e., host CPU compression, external hardware compression, on-chip hardware compression, and in-storage hardware compression). In the following, we present a comprehensive study of them, and their comparisons are demonstrated in Table I.

1) Host CPU Compression: Host CPU compression directly uses CPU cores to (de)compress data on the host-side I/O path. Meanwhile, host software maintains the indexes of chunks to ensure correct decompression afterward. Here, a *chunk* represents an independent compression unit, and an *index* includes the logical address and size of a chunk (at

[1] https://github.com/yingjia-wang/CCZNS

[2] In this paper, we focus on lossless data compression.

TABLE I
COMPARISONS OF REPRESENTATIVE COMPRESSION-ENABLED STORAGE SYSTEM ARCHITECTURES [37], [98], [100].

	CPU	External	On-Chip	In-Storage
Cost reduction	✓	✓✗	✓	✓
Low CPU overhead	✗	✓	✓	✓
Hardware-accel. perf.	✗	✓✗	✓✗	✓
Perf. scalability	✗	✗	✗	✓

byte granularity). Host CPU compression supports various compression algorithms and levels simply by invoking their software libraries. *Lightweight compression algorithms* (e.g., Snappy [17], LZ4 [12]) provide fast compression speeds with compromised compression ratios. On the other hand, *standard compression algorithms* (e.g., ZSTD [23], deflate/Gzip [40]) can achieve higher compression ratios for more storage savings, albeit at the cost of higher computational overhead.

Due to its ease and flexibility, host CPU compression is the most popular method without dependencies on hardware accelerators. However, *CPU is difficult to balance compression ratio and system performance*. Employing standard compression algorithms not only considerably degrades the system performance but also leads to substantially high CPU utilization that potentially jeopardizes other compute-intensive tasks. Google's datacenters consume 2.9% of CPU cycles on (de)compression tasks and 10%-50% in some key services, even so, 95% data has to sacrifice compression ratio for balancing performance [62]. Pangu, the unified storage system for Alibaba Group and Alibaba Cloud, faces the CPU bottleneck that limits its theoretical throughput [69].

2) External Hardware Compression: To improve compression performance and alleviate host CPU utilization, previous studies explore how to offload compression onto external hardware compression accelerators, such as GPU [75], [85], [102], FPGA [33], [41], [79], DPU [70], and ASIC [49]. However, these PCIe-connected accelerators necessitate software adaptations and also round-trip data transfer between host and accelerators. The indexes of chunks are still maintained by host software and updated after chunks are compressed and transferred back to the host.

Due to the round-trip data transfer, *offloading compression to external hardware accelerators is not efficient for small chunks* [49], [55]. Many host software organizes data into smaller chunks to reduce computation and mitigate I/O latency during decompression. For example, both RocksDB [13] and RedHat VDO [2] employ 4KB-chunk-based compression by default. QZFS, a filesystem integrating ASIC-based Intel®QAT card, still uses CPU compression when the chunk size is less than 4KB [49]. Cache in modern datacenters also organizes data into small chunks for low-latency read services [26], [55], [96]. These common cases can not reap significant performance improvement from external hardware accelerators.

In addition, *employing these external accelerators for compression leads to the occupancy of valuable PCIe lanes and is also detrimental to system costs.* Since these accelerators are designed to be either general-purpose or support multiple functions, users have to afford higher costs of the devices and functions.

TABLE II
COMPRESSION THROUGHPUT OF ON-CHIP/EXTERNAL QAT UNDER THREE DATASETS WITH A WIDE RANGE OF COMPRESSION RATIOS (CR). THE CHUNK SIZE IS 4KB.

	ooffice(CR≈1.6)	**mozilla(CR≈2.6)**	**nci(CR≈5.9)**
On-Chip QAT	1446.6MB/s	1919.3MB/s	2782.2MB/s
External QAT	1263.8MB/s	1629.0MB/s	1857.6MB/s

3) On-Chip Hardware Compression: Intel has announced the on-chip integration of accelerators with some of the latest CPU products (e.g., Intel®Xeon®Scalable Processors) [64], [98]. Compared to the off-chip or external form, this architecture not only achieves higher cost efficiency but also enables the accelerators to exploit (1) the CPU's powerful memory subsystem and (2) a hardware/software co-designed ecosystem consisting of SoC-level hardware features and comprehensive software stacks [98].

However, *on-chip hardware compression can not fundamentally change the landscape where the small-chunk compression performance is unsatisfactory*, since on-chip accelerators are still PCIe components and data movements between the CPU and the accelerator are still constrained. We conducted experiments on an Intel®Xeon®Platinum 8458P Processor [19] with on-chip QAT and compared the results with those on an external QAT card (i.e., Intel®QuickAssist Adapter 8970 [18]). Particularly, we use QATzip [11] to compress three datasets in the popular Silesia corpus with a wide range of compression ratios [15]. The results are shown in Table II. We can see that, compared to external QAT, the small-chunk performance of on-chip QAT is higher but still much below the specified bandwidth (e.g., a dozen GB/s).

4) In-Storage Hardware Compression: Recently, computational SSDs with in-storage hardware compression have emerged and gained industry traction (e.g., CSD series from ScaleFlux [14], SmartSSD from Samsung [16], and Roealsen6 from DapuStor [8]). These SSDs incorporate hardware compression engines inside and shift (de)compression tasks from the host to the SSD itself. To comply with the block interface, these SSDs perform compression in a host-transparent manner, and we call them *TCSSDs* (TC: transparent compression) in the following for simplicity.

To summarize, TCSSD exhibits the following advantages:

- *High compression performance.* Our ASIC-based compression engines that have been commercially deployed offer up to 11GB/s and 14GB/s compression/decompression bandwidth with only 5% extra area size on the SSD controller. The inline hardware compression on the device-side I/O path is also efficient for small chunks because it does not involve additional round-trip data movement.
- *Performance scalability.* The compression performance can scale up with the number of SSDs, as each SSD has its own compression engines for performance acceleration.

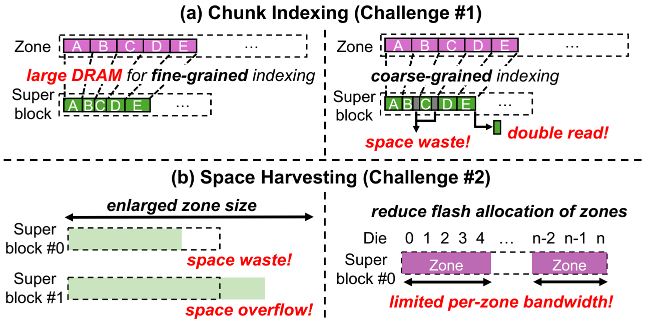


Fig. 2. **Two intrinsic challenges to the host-transparent methodology.**

- *Low CPU overhead.* Host CPU utilization can be significantly reduced through compression offloading. Since the (de)compression bandwidth per CPU core is only hundreds of MBs using standard algorithms [6], [42], the use of in-storage hardware engines can free up dozens of CPU cores for other tasks.

TCSSD has two special features compared to the traditional SSD. First, TCSSD equips a more fine-grained address mapping table (compared with the typical page level) to locate compressed chunks. Second, to reap space savings from compression, TCSSD exposes an enlarged logical volume, the size of which is pre-determined based on an estimated compression ratio in the deployment environment. For example, a 16TB logical volume is exposed if the physical flash capacity is 4TB and the estimated compression ratio is 4.

C. Motivation

Based on the analysis in §II-B, enabling in-storage hardware compression on ZNS SSDs is promising to constitute more cost-efficient ZNS-based storage infrastructures. However, the *host-transparent* methodology adopted in TCSSDs and the state-of-the-art work Balloon-ZNS [93] is not practical for ZNS. The key idea of Balloon-ZNS is to leverage the per-zone compressibility locality (i.e., the relatively consistent data compression ratio in each zone) for compressibility-adaptive chunk indexing and storage management. Nevertheless, despite the feasibility, the designs for both chunk indexing and storage management suffer from significant trade-offs in cost and/or performance (as introduced below), making the state-of-the-art work far from efficient.

Broadly speaking, the host-transparent methodology inevitably delivers two intrinsic challenges for us to overcome.

Challenge #1: *The fine-grained indexes to locate compressed chunks conflict with the cost objective of ZNS and further worsen the on-board DRAM shortage issue; the compromised coarse-grained approach sacrifices both write amplification and read performance.*

In-storage compression alters both addresses and sizes of the data chunks. Since these changes are invisible to the host, the SSD necessitates *fine-grained* indexes internally to locate compressed chunks. However, ZNS features limited on-board DRAM, mainly for holding the coarse-grained zone-level mapping table and the write buffer [30], [74]. Thus, the fine-grained indexes reintroduce large on-board DRAM and directly contradict one of the primary goals of ZNS (i.e., to lower the device cost).

In addition, as introduced in §II-B4, the SSD logical volume should enlarge with the estimated compression ratio to reap space savings. This significantly expands the logical-to-physical mapping table, which not only increases the DRAM cost but also challenges the high-capacity trend that ZNS caters to. The density gap between DRAM and flash is expected to increase further (e.g., over 1000× in 2030 [5]), while the number of DIMMs inside an SSD is difficult to scale [39], [72], [89].

To mitigate fine-grained indexing overhead, an intuitive strategy is to use *coarse-grained* approaches. For example, Balloon-ZNS [93], the state-of-the-art work, locates chunks in a zone by proportionally scaling down their original addresses, according to the profiled compression ratio in that zone. Nevertheless, the majority of chunks can not be stored as expected. More precisely, the chunk size after compression is most likely different from the pre-allocated scaled-down storage space. As shown in Figure 2a, to follow this coarse-grained contract, smaller-than-expected chunks (i.e., B and C) result in unnecessary storage space waste, which negates the benefit from compression. At the same time, larger-than-expected chunks (i.e., E) must be split with some at the tail stored separately. The tail part can only be read after the main part because its actual address is stored in the main part, which leads to double reads and worsens the read latency. Experiments on RocksDB show that our proposed approach can further reduce the write volume by 12.6% and the average read latency by 16.1% (see §V-C).

Challenge #2: *The explicit zone boundaries of ZNS hinder the host from harvesting space savings effectively; due to the mismatching between zone and flash superblock, either enlarging logical zone size or shrinking physical flash space allocation is not practical.*

According to §II-B4, TCSSD presents storage space savings to the host in the form of a magnified logical volume. However, when this strategy meets the explicit zone boundaries of ZNS, harvesting space savings becomes much more difficult.

Compared to the TB-scale SSD capacity, each zone is only tens of MBs to a few GBs [30], [74]. Thus, it is *hard to determine a uniformly enlarged zone size to the host*, given the distinct real-world data compressibility. As shown in Figure 2b, a conservative estimation may incur excessive space waste in superblocks, offsetting the advantage from compression. On the other hand, though an aggressive estimation may reap the space savings maximally, it would cause frequent space overflow, necessitating host software to handle exceptions (e.g., redirect writes into other zones).

Instead of enlarging the logical zone size as TCSSD does,

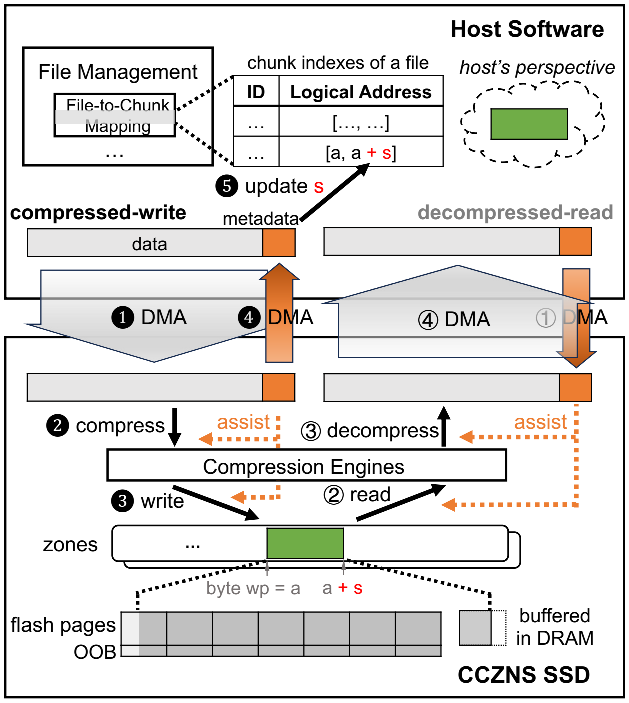


Fig. 3. **Overview of CCZNS.**

another strategy is to *shrink physical flash space allocation of zones.* Balloon-ZNS [93], for instance, groups several zones into a single superblock and assigns each zone to a subset of flash blocks in that superblock (see Figure 2b). Compared to the conventional ZNS which adopts one-to-one mapping between zones and superblocks, Balloon-ZNS restricts the flash parallelism (e.g., dies) each zone can use, resulting in limited per-zone bandwidth and lowering access performance. Experiments on RocksDB show that the average write throughput of Balloon-ZNS is only 60.5% of our proposed approach (see §V-C).

Key Insight. To overcome the challenges due to host transparency, potential solutions including the state-of-the-art work sacrifice cost, write amplification, and/or access performance, thereby failing to unleash the potential of in-storage hardware compression.

In this paper, our key insight is that, *given the coupled logical and physical addresses within each zone, host software can leverage existing fine-grained indexes to directly manage compressed data inside the SSD.* By doing so, the two aforementioned challenges can be well resolved. On the one hand, our approach eliminates redundant SSD-internal indexes, which preserves the DRAM-less or cost merit of ZNS and does not face the on-board DRAM shortage problem even with high flash capacity. On the other hand, our approach can harvest space savings effectively and accurately because the logical zones and the physical flash superblocks are always matched.

III. Design of CCZNS

A. Overview

We present *CCZNS* (CC: collaborative compression), an advanced ZNS interface that revives in-storage hardware compression through a novel host-SSD collaborative approach. Specifically, CCZNS offloads only compression execution to the ZNS SSD but responds the post-compression information to host software for maintenance.

Figure 3 demonstrates the overview of CCZNS. Since compression execution and indexing are now decoupled, CCZNS introduces two specific read/write commands (i.e., *compressed-write* and *decompressed-read*, CW and DR in short) to enable the cross-layer management. When CW and DR are invoked from the host, hardware compression engines can be activated by the SSD controller to (de)compress data on the device-side I/O path. CW and DR leverage the *metadata field* in the NVMe command to (1) provide extra information to assist (de)compression and read/write processes and (2) deliver the information for index management back to the host.

The general flow of writing/reading a chunk via CW/DR is explained as follows. For CW, the chunk and its metadata are first transferred to the SSD via DMA (i.e., ❶). The chunk can then be compressed (i.e., ❷) and written to the zone after compression (i.e., ❸). To report the index information, CCZNS still utilizes the metadata field, overwrites it, and transfers it back to the host (i.e., ❹). Host software can then extract the above information and update the chunk index accordingly (i.e., ❺). Once the index is successfully updated, the view of the chunk is consistent in the perspectives of both host and SSD (i.e., the post-compression state). When this chunk is read later by DR, the host first attaches its index information in the metadata field (i.e., ①). Based on this information, CCZNS can then locate the chunk in the zone, read it (i.e., ②), and send it to the compression engines (i.e., ③). The chunk can be finally transferred back to the host after decompression (i.e., ④).

In the following, we introduce the detailed designs of CCZNS. We first introduce the command formats; particularly, for different file types and compression granularity, we introduce single-chunk and multi-chunk modes of CW and DR in §III-B and §III-C, respectively. In §III-D, we introduce byte-addressable storage management, which accommodates CC-specific commands, manages compressed chunks, and harvests space savings from compression maximally. In §III-E, we introduce how CW is compatible with traditional write within the same zone in practice. In §III-F, we deliver several meaningful discussions.

B. CC-specific Read/Write Commands

The design highlight of CW and DR is the integration of CC semantics, such as compression offloading and information responding, in traditional read/write commands without additional I/O operations. This is feasible by utilizing reserved bits (up to two) in NVMe command *dwords*, as well as the NVMe *metadata field* that is commonly used for hint transfer or data protection [31]. The size of the metadata field can be flexibly

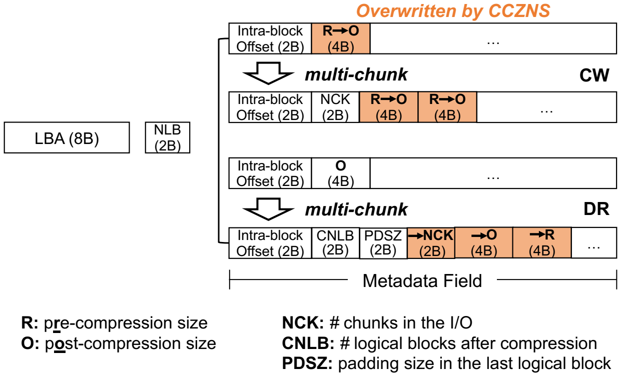


Fig. 4. **Command formats of CC-specific read/write commands.** The orange background indicates that CCZNS will overwrite this value. The left and right sides of −> are the user-specified and CCZNS-overwritten values, respectively.

supported by the SSD firmware according to the host software requirements.

CW and DR are derived from traditional NVMe read/write commands and identified by a reserved bit. As shown in Figure 4, in addition to *logical block address* (i.e., LBA) and *the number of logical blocks* (i.e., NLB) that exists in traditional read/write commands, host software should also provide some extra information to locate a chunk. The extra information includes:

- *intra-block offset*, which jointly cooperates with LBA to specify the byte-granular write location.
- *post-compression size (for DR)*, which specifies the byte-granular address range of the to-be-read chunk.

For CW, CCZNS only additionally requires intra-block offset to check for I/O legitimacy (i.e., whether the write location is aligned with the current byte-granular write pointer, see §III-D). To facilitate host-side index maintenance, CCZNS reports the post-compression chunk size to the host by overwriting the metadata field and transferring the metadata back. Currently, the NVMe driver of the Linux kernel only supports NVMe metadata transfer in the same direction as I/Os (e.g., metadata can only be passed from the host to the SSD in write I/Os). We modify the driver to support the bi-directional NVMe metadata transfer with a few lines of code.

For DR, CCZNS requires not only intra-block offset but also post-compression size because both of them are essential for CCZNS to locate a chunk.

In either block-interface SSD or ZNS SSD, NLB is simply set according to the original data size because neither the I/O stack nor the SSD changes the data size. In contrast, for CCZNS (both CW and DR), NLB should be set to at least the *pre-compression size*. This setting ensures that the I/O stack allocates enough memory space for the chunk to be transferred between host and SSD.

C. Aggregating Multiple Chunks in a Single I/O

We call the commands introduced in §III-B as *single-chunk* CW/DR because each read/write I/O only accommodates one chunk. The single-chunk CW and DR are well-suited for large-size files such as containers or objects. However, when the chunk size is small, the single-chunk commands would lead to a surged number of I/Os and thus considerably degrade the performance. For example, the default chunk sizes of three ZNS-compatible host software (i.e., RocksDB, F2FS, and BtrFS) are 4KB, 16KB, and 4KB, respectively[3], indicating the importance of optimizing small-chunk scenarios.

We present *multi-chunk* CW and DR, which occupy one more reserved bit compared to their single-chunk counterparts (i.e., one for identifying CW/DR and the other for identifying the single-/multi-chunk mode). The multi-chunk commands can collaborate with common system optimization strategies to improve performance in small-chunk scenarios.

Multi-chunk CW can coordinate with *write buffer*, which flushes a batch of chunks together. As shown in Figure 4, the metadata field of multi-chunk CW includes (1) intra-block offset, (2) *the number of chunks in the I/O* (i.e., NCK), and (3) *pre-compression sizes of all chunks*. NCK and pre-compression sizes are essential for CCZNS to partition the chunks and deliver them for compression separately. Once chunks are compressed and written, CCZNS overwrites their pre-compression sizes in the metadata field with the post-compression ones, which are then passed to the host for index maintenance.

Prefetching is a typical case for multi-chunk DR. As shown in Figure 4, the metadata field of multi-chunk DR includes (1) intra-block offset, (2) *the number of logical blocks after compression* (i.e., CNLB), and (3) *padding size (in the last logical block)* (i.e., PDSZ). They along with LBA together specify the actual read address space for CCZNS. Multi-chunk DR does not require the host software to provide post-compression sizes of chunks but leaves CCZNS to locate them by looking up the flash *Out-Of-Band (OOB)* area (see §III-D). This avoids the traverse of chunk indexes to attach all the post-compression sizes in the metadata field, which complicates the command use and degrades software performance. After chunks are read and decompressed, the number of chunks (i.e., NCK) and both pre-/post-compression sizes of chunks should be transferred back to the host. This is to construct temporary mappings of chunks from logical addresses to the addresses in the buffer to accelerate future access, assuming that the prefetched chunks are stored in the buffer.

Multi-chunk DR adopts a simple but effective *prefix-chunk-based prediction* method from [47] to set NLB, which is crucial for preserving enough memory space in the I/O stack for the prefetched data. This method leverages the data compressibility locality and uses the compression ratio of the first chunk to forecast the overall compression ratio in the I/O. Here, the compression ratio is calculated by dividing pre-compression size and post-compression size. For example, if the prefetching size (i.e., CNLB) is 1MB and the compression ratio of the first chunk is 2, NLB is set to 2MB. After chunks

[3] In BtrFS, the compression process handles data in the granularity of 128KB. However, each 4KB is still compressed separately [7].

are read and decompressed, it is possible that the preset NLB is not large enough to hold all the chunks in the read address space. CCZNS prefetches the maximum number of chunks within the NLB capacity and only transfers the number of these chunks and their pre-/post-compression sizes back.

D. Byte-Addressable Storage Management

To accommodate CW/DR and manage compressed chunks, CCZNS introduces *byte-addressable storage management* that can harvest space savings from compression maximally. Overall, the byte-addressable storage management aims to expose the SSD storage space and write pointers of zones in *bytes* (rather than logical blocks).

For traditional write or CW, data or compressed data is still written sequentially in a zone but the write pointer advances the number of written bytes. The latest data less than the minimum flash write granularity (e.g., a flash page) is buffered with capacitor protected [35] and will be merged with subsequent data. For traditional read or DR, CCZNS first aligns the read address space in either logical-block-level (i.e., traditional read) or byte-level (i.e., DR) to flash read boundaries. After reading, both traditional read and DR extract the necessary data but only DR involves decompression before returning the data to the host.

In addition to responding to the host, CCZNS also keeps lightweight metadata of chunks in the *Out-Of-Band (OOB)* area of the flash pages. The metadata of each chunk includes (1) the offset in the page, (2) the post-compression size, and (3) the *feedback size*. Here, feedback size indicates the size reported by the CW to the host, which may be different from the post-compression one (the role is introduced below). When the flash page is programmed, the metadata of new chunks (i.e., the start addresses of chunks are within the page) is also persisted in the corresponding OOB area.

Maintaining such metadata in OOB has several benefits. First, CCZNS can utilize it to check the address legality of DR (i.e. if the read address space preserves the integrity of chunks), and illegal DR can be safely rejected. Second, the metadata enables CCZNS to determine chunk boundaries and decompress chunks independently, which promotes efficient multi-chunk DR that can relieve the host-side overhead. Third, the decoupling of actual post-compression size and feedback size makes CCZNS more resilient. For example, the feedback size can harmonize CW with traditional write in the same zone (see §III-E) and opens up higher potential on optimizations, such as restricting chunks across flash page boundaries [57].

The metadata of chunks incurs insignificant space overhead in OOB. Modern SSDs usually have OOB with 128 to 256 bytes per 4KB flash [45], [58], [89]. Assuming an extreme case that the chunk size is as small as 4KB and the compression ratio is as high as 4 (higher than the common data compression ratio), each 4KB flash accommodates 4 chunks and only requires $(2+4+4)*4=40$B.

E. Harmonize CW with Traditional Writes

CW and DR should be compatible with traditional read/write commands not only at the command level but also applicable in practice. For example, different parts of files can have different semantics and need to be partially compressed. However, mixing CW and traditional writes in the same zone is not practical due to the inconsistent granularity of the write pointer. Simply separating CW and traditional writes into different zones is not desirable due to file fragmentation and limited open zones (e.g., 14 in Western Digital ZN540 [30]).

To harmonize CW with traditional writes in the same zone, we propose a new zone abstraction, namely *padding zone*, which differs from a conventional zone by automatically aligning the *last* chunk after compression to the logical block boundary. The zone type (i.e., padding or conventional) is specified via a reserved bit in the explicit zone open command. As introduced in §III-D, with the decoupled feedback and post-compression sizes, CCZNS can report the size after padding to the host but still relying on the actual post-compression size in the OOB to locate the chunk.

The extra space overhead is negligible in both single- and multi-chunk CW, as the I/O size should be large in both modes (i.e., the former has one large chunk while the latter has multiple small chunks). Assuming the post-compression chunk(s) in an I/O are 256KB (i.e., 1MB raw data with a compression ratio of 4), the average space amplification due to padding is only 0.8% (i.e., $4/2=2$KB compared to 256KB).

F. Discussions

Incompressible data handling. It is advisable for host software to determine whether to use CW/DR or traditional read/write commands considering its data patterns, and this strategy is consistent with the software definability nature of ZNS. Specifically, for incompressible data, host software can directly use traditional read/write commands to avoid unnecessary computational and energy overhead.

Meanwhile, the FTL can also skip unrewarded compression by efficiently predicting compression ratio via compressing sampled data [47] or estimating based on data entropy [34]. If the predicted compression ratio is too poor, the FTL directly writes data without compression.

Zone Append support. Zone Append is a new write method of ZNS, which breaks the QD=1 limitation[4] and allows multiple concurrent writes dispatched to the same zone [3]. The actual write address of Zone Append is determined by the ZNS SSD based on their arrival order and reported to the host. Once I/O is completed, the host can then perceive the actual write address and update the index accordingly.

The current (write-based) CW can be easily ported to the Append-based CW. Their only difference is that, for Append-based CW, CCZNS should also return the actual write address but in byte granularity (rather than the logical block). Currently, there is no existing ZNS-compatible host software that

[4] Due to the sequential write constraint of ZNS, Linux kernel employs the *mq-deadline* scheduler to restrict only one write in a zone can be processed at one time [22].

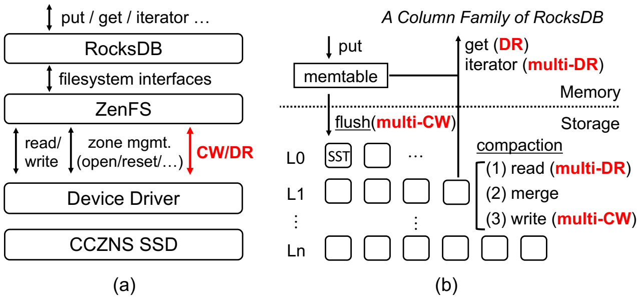


Fig. 5. **Host-SSD collaborative architecture of RocksDB, ZenFS, and CCZNS SSD.**

supports Zone Append in the compressed I/O path[5]. We leave the evaluation of Zone Append-based CW as future work.

Implications on Flexible Data Placement (FDP) [84]. FDP is a recent NVMe proposal aiming to eliminate device-side write amplification, which is also one of the goals of ZNS. Through FDP, host software can issue writes with *hints* to guide data placement, and the SSD controller can store data with different hints into different flash superblocks. Compared to ZNS, FDP does not save on-board DRAM and could not achieve utmost benefits in write amplification and flash over-provisioned space. However, FDP retains backward compatibility of the block interface, which avoids significant host-level adaptations (that ZNS requires) and is thus easy to deploy.

Since FDP still preserves a fine-grained mapping table inside the SSD, the logical and physical addresses are not coupled. This prevents host software from performing consistent and efficient cross-layer management like using CCZNS. However, it is promising for FDP to leverage the concept of host-SSD collaboration to effectively suppress the proportional increase of on-board DRAM with the compression ratio (see §II-B4), which is crucial for reducing hardware cost. We leave this vision as future work.

IV. Unleashing the Potential of CCZNS

A. Overview

Given the fine-grained data indexes inherent in the host software, CCZNS can be widely adapted to offload burdensome (de)compression tasks and improve system performance, such as key-value stores, filesystems, SSD cache, and all-flash array. Moreover, the adaptation can be implemented in one layer, and upper-layer software can benefit from CCZNS without any code modification.

Overall, there are two essential principles for generalizing CCZNS into ZNS-compatible host software. The first is *the requirement of fine-grained host-side chunk indexes.* Fortunately, these are essential in host software (e.g., RocksDB, F2FS) due to the need to manage compressed chunks in byte granularity. The second is that *indexes should be persisted later than chunks or data* (the same as what Zone Append [3] requires). This principle is non-intrusive and in fact crucial for data correctness during recovery. For example, if the system crashes when indexes are persisted but the data is not, the indexes will point to the wrong data.

In the following, we introduce how CCZNS can benefit host software through a case study on RocksDB [13], the mainstream application of ZNS, and its filesystem backend ZenFS [21]. The engineering efforts are moderate, with about 800 LoC in RocksDB and 300 LoC in ZenFS.

B. Case Study on RocksDB and ZenFS

Background. RocksDB [13] is a popular persistent key-value store developed by Facebook and optimized for SSDs. Many large-scale storage services, websites, or software employ RocksDB as the underlying storage engine due to its high performance on key-value services. ZenFS [21] is a filesystem plugin of RocksDB that provides end-to-end compatibility for ZNS (see Figure 5a). ZenFS uses RocksDB's filesystem interfaces to store data files into zones based on their lifetime.

RocksDB adopts the *log-structured merge (LSM)* tree to organize data files (see Figure 5b), with newer ones stored in higher levels (e.g., L0) and older ones squeezing into lower levels (e.g., Ln). For every column family or logical partition in RocksDB, new key-value pairs via `put` operations are first batched in an in-memory *memtable*. Upon reaching a preset size limit, the memtable is marked as immutable and persisted in storage as a *sorted string table (SST)* via the *flush* process. *Compaction* is triggered when a level exceeds its size limit, in which a victim SST is selected and merged with key-overlapping SSTs in the next lower level. Compaction enhances storage space efficiency as deleted data can be eliminated and levels of SSTs become more compact.

RocksDB employs the block-based SST table format by default, which includes *data blocks*, *meta blocks*, *metaindex blocks*, and a *footer*. During SST construction, key-value pairs in the SST are sorted and divided into data blocks (4KB by default). Data blocks are compressed individually and stored compactly from the start of the file. Right after data blocks, there are a few meta blocks. Among them, *index blocks* hold the index entries of data blocks for key-value pair lookups. Each *index entry* contains the maximum key in the corresponding data block, the logical byte offset in the SST, and the post-compression data block size. After compressing the respective data block, an index entry is immediately constructed and added to the index blocks.

Utilizing CC-specific read/write commands. ZenFS currently uses the `pread`/`pwrite` interfaces. For CCZNS, data blocks are operated by CC-specific read/write commands (i.e., CW and DR) to offload the burdensome (de)compression tasks. For other parts in the SST (e.g., index blocks) and other files except for SSTs (e.g., filesystem metadata), CCZNS still employs the traditional read/write commands. The zones storing SSTs are opened in the padding mode to harmonize different write methods (i.e., CW and traditional write).

Extending index entries for decompression. As introduced in §III-B, host software should maintain both post- and pre-

[5] BtrFS has announced the experimental use of Zone Append. However, this only applies to the non-compressed data path and not to the metadata and compressed data paths [25].

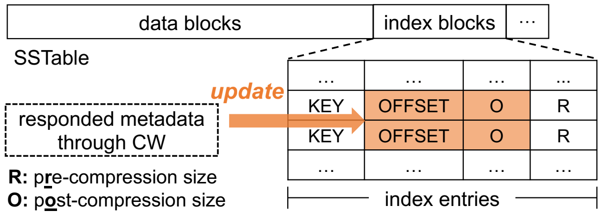


Fig. 6. **New index entry format in CCZNS-aware RocksDB.**

compression sizes. The former is used to locate chunks, and the latter is used to preserve enough space for data retrieval. Therefore, the index entry format is added with an additional field that stores the pre-compression size (i.e., R in Figure 6). This inevitably expands the index blocks, but the overall memory usage increment is acceptable (see §V-C).

Constructing index blocks after persisting data blocks. Since CCZNS can only report the post-compression sizes of data blocks after the completion of CW (similar to Zone Append [3]), we postpone the construction of index entries until all data blocks are persisted. Once constructed, the index blocks are still persisted right after the data blocks.

Leveraging multi-chunk-per-I/O optimizations. Adopting multi-chunk CW and DR is crucial for RocksDB to leverage its write buffer and prefetching strategies, greatly reducing the number of I/Os and leading to better performance. When persisting an SST (e.g., in flush or compaction), each flush of the write buffer (1MB by default) corresponds to a multi-chunk CW. Similarly, each prefetching on an SST is viewed as a multi-chunk DR. Prefetching happens either in compaction or iterator operations (i.e., scan).

Crash consistency is not affected. CCZNS does not affect the crash consistency of RocksDB. This is not only because CCZNS follows the principle that the indexes are still persisted later than chunks, which is crucial for data recovery. Also, all our adaptation logic above only applies to the internal processes of SST construction, which is atomic in RocksDB. If the system crashes during SST construction, RocksDB will rebuild this SST again after recovery.

V. Evaluation

A. *Experimental Setups*

Environment. All experiments are conducted in a virtual machine with 32GB memory, 16 cores, and a Linux kernel (version 5.10.9). The virtual machine is running on a physical server equipped with four 24-core/48-thread Intel Xeon Platinum 8260 2.40 GHz CPU sockets.

SSD configurations. Both CCZNS and ZNS SSDs are implemented on a popular SSD emulator FEMU [67], which runs in the virtual machine and thus supports full-stack system research. The flash page size and block size are 16KB and 32MB, respectively. The SSD has 64 parallel flash dies, and each zone consists of 32 blocks across half of the dies (the zone-to-flash mapping is similar to that of Western Digital ZN540 [4]). The zone size is thus 1GB, and the total SSD capacity is configured as 128GB. The TLC flash read, write, and erase latency are set to 90μs, 700μs, and 5ms, respectively [76]. CCZNS supports each 4KB logical block with 64B metadata, which is enough for information transfer.

FEMU extension for emulating hardware compression. CCZNS emulates hardware compression by adding extra latency delay on the critical read/write paths, referring to the statistics of real ASIC-based compression engines. Meanwhile, we use multiple CPU cores to perform compression in parallel, ensuring that data compression can be finished before the emulated I/O delay.

Since the vanilla FEMU dedicates multiple threads (e.g., 8) to process I/Os alternatively (i.e., each thread picks one from NVMe SQ, processes the I/O, and sends it back to the corresponding NVMe CQ), performing CPU compression on the critical I/O path can severely block the I/O. To resolve this issue, we delegate I/O processing and (de)compression into different groups of threads. Specifically, I/Os from an NVMe SQ are handed to different I/O processing threads (8 by default shared between NVMe SQs) for parallel execution. Each I/O processing thread then distributes chunks to compression threads (16 is enough in our evaluation) and collects them before returning to the NVMe CQ. Each thread runs exclusively on a server CPU core and thus the total number of cores reserved for FEMU is $8+8+16=32$. Note that the server CPU overhead here is only for high-quality emulation and does not exist in real products.

Evaluated schemes.

- **NO+ZNS (abbr. NO or N)** directly writes data to the ZNS SSD without compression.
- **Snappy+ZNS (abbr. Snappy or S)** writes data after compression by CPU cores using the Snappy algorithm.
- **QAT(deflate)+ZNS (abbr. QAT or Q)** uses the official QAT plugin for RocksDB [10] to offload compression to the external Intel®QuickAssist Adapter 8970 [18], which employs the deflate/Gzip algorithm[6].
- **ZSTD+ZNS (abbr. ZSTD or Z)** writes data after compression by CPU cores using the ZSTD algorithm (level 1 with the fastest speed); compared to Snappy, ZSTD achieves a higher compression ratio at the expense of performance.
- **ZSTDPC+ZNS (abbr. ZSTDPC or ZP)** is similar to ZSTD but employs 6 threads for parallel compression; we do not observe further performance gains with a larger number of compressed threads.
- **Balloon-ZNS(ZSTD) (abbr. Balloon or B)** is the state-of-the-art work that revives in-storage hardware compression in a host-transparent way [93]; it performs ZSTD compression (level 1) and more design details are introduced in §II-C.
- **CCZNS(ZSTD) (abbr. CC)** is our proposed scheme that revives in-storage hardware compression through host-SSD collaboration and performs ZSTD compression (level 1).

[6] Note that due to different hardware, QAT should not be compared to CC head-to-head. Even so, QAT is representative of the external/on-chip hardware compression architecture where the small-chunk (de)compression performance is unsatisfactory due to data round-trip movements.

TABLE III
PERCENTILE DISTRIBUTION OF VALUE SIZES (IN BYTES) IN TWO REAL DATASETS. **A:** AMAZON. **R:** REDDIT.

	Avg.	**10p**	**25p**	**50p**	**75p**	**90p**	**99p**
A	1173.7	345	465	779	1436	2499	5672
R	947.2	620	676	786	1005	1421	3208

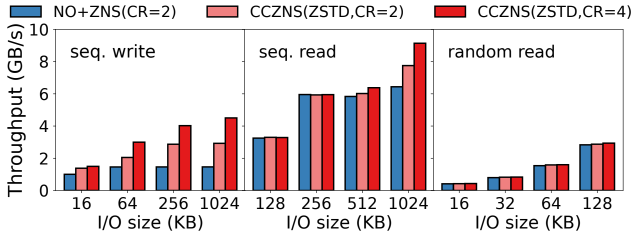


Fig. 7. **Throughput in FIO micro-benchmarks with different I/O sizes. CR:** compression ratio of the data.

B. Raw SSD Access Performance

We first evaluate the raw SSD access performance of CCZNS and ZNS (corresponding to CC and NO). The objective is two-fold: (1) investigating the access performance characteristics compared to the conventional ZNS and (2) evaluating the performance of single-chunk CW/DR commands. We use FIO [9], a popular benchmark tool for testing, and modify it to adapt CCZNS. Considering that real-world data compression ratios are typically less than 4, we use synthetic data with compression ratios of 2 and 4 for evaluation. Figure 7 shows the throughput in three micro-benchmarks (i.e., sequential write, sequential read, and random read). Each benchmark has different I/O size configurations suitable for their patterns and employs 4 threads writing different ranges of zones concurrently. Each thread dispatches I/O requests constantly and thus the access load is high. We omit NO+ZNS(CR=4) because the compression ratio (i.e., CR) does not matter when data is not compressed, and the performance of NO+ZNS(CR=4) is similar to NO+ZNS(CR=2).

In the sequential write workload, CC outperforms NO by 37.6%∼99.9% when CR is 2 and 49.1%∼2.1× when CR is 4. Moreover, CC can exceed the theoretical SSD write bandwidth to up to 2.0× and 3.1× when CR is 2 and 4, respectively. Due to high-performance hardware compression engines, the same flash parallel structure in the SSD can program a larger amount of data simultaneously, which increases the user-perceived write throughput.

In the sequential read workload, CC can also surpass NO and the theoretical SSD read bandwidth, though the gain is smaller and only shown when the I/O size is large, due to the smaller bandwidth gap between hardware compression and flash read. The performance improvement compared to NO is up to 20.4% and 42.0% when CR is 2 and 4, respectively.

In the random read workload, CC shows similar or slightly higher (less than 5%) throughput compared to NO. We also evaluate the average and 99.99p latency in the random read workload. The results of CC and NO are comparable, except that when the I/O size is 128KB, CC can reduce the average latency by 38.8% and 36.0%, and the 99.99p latency by 55.9% and 60.2% when CR is 2 and 4, respectively. The reason behind this is that CC can read fewer flash pages during each read, thus reducing flash die contention and improving the latency.

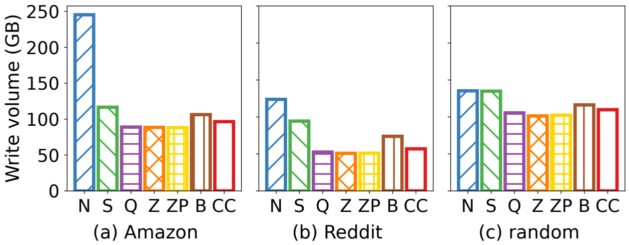


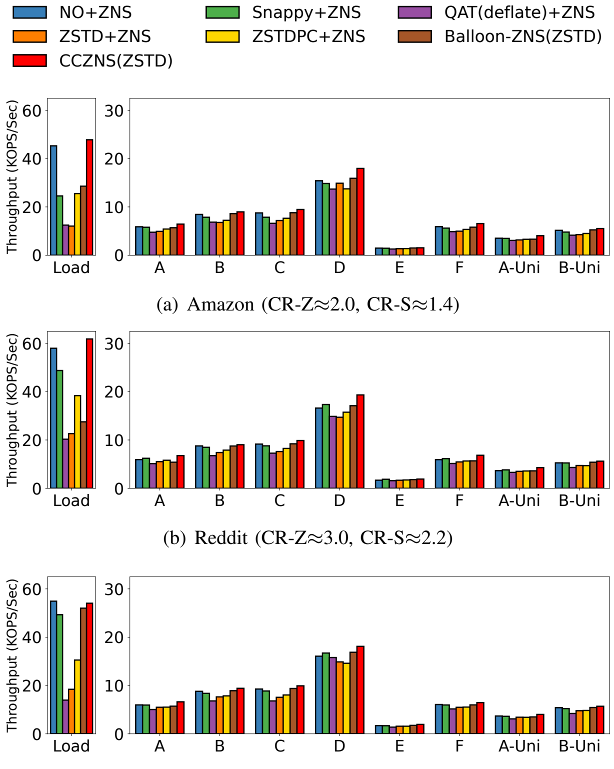
Fig. 8. **Write volume in YCSB Load using different datasets.** The full names of schemes (i.e., x-axis labels) are shown in §V-A.

Since both CW and DR require additional information transfer through the NVMe metadata field (i.e., 64B per 4KB data), we also analyze the DMA time for metadata transfer in both commands. Across different benchmark configurations, the DMA time for metadata transfer accounts for less than 1% of the total processing time in the SSD, indicating that the metadata transfer has a trivial impact on the I/O latency.

C. RocksDB Performance

We further conduct a system-level evaluation on RocksDB (version 8.11.0) using YCSB benchmarks [38]. In RocksDB, the block size is 4KB (the default), and the block cache is configured to a moderate 4GB that can only hold a fraction of data. We employ direct I/O to bypass the kernel page cache. Since data patterns can significantly affect the (de)compression performance, we employ three datasets with distinct compression ratios. They are two real datasets from Amazon [73] and Reddit [1] (the value size distribution is shown in Table III) as well as one synthetic dataset (random data with 1KB key-value pairs). Among them, Amazon and Reddit datasets try to cover the typical range of real-world compressibility [55], while the random dataset represents a case that is relatively unfavorable for CCZNS. We use all workloads in YCSB for evaluation. In YCSB Load, we insert 20M key-value pairs into the database. In other workloads, we perform 5M operations after completing YCSB Load.

Write Volume. Figure 8 shows the write volume in YCSB Load using different datasets, and a lower write volume contributes to a lower cost. We can see that CC reduces the write volume by an average of 44.5%, 25.2%, and 12.6%, compared to NO, Snappy, and Balloon. Host-side compression using standard algorithms, namely QAT, ZSTD, and ZSTDPC, further reduce the write volume by 6.5%, 8.8%, and 8.6%. Despite that CC has the same compression algorithm and level as ZSTD and ZSTDPC, CC necessities data padding at the end of multi-chunk CWs for compatibility with traditional writes in the same zone (see §III-E). Nonetheless, considering the



(a) Amazon (CR-Z≈2.0, CR-S≈1.4)

(b) Reddit (CR-Z≈3.0, CR-S≈2.2)

(c) random (CR-Z≈1.3, CR-S≈1, relatively incompressible)

Fig. 9. **Throughput in YCSB workloads using different datasets. CR-Z/CR-S:** average compression ratio using ZSTD or Snappy. **Load:** write-only. **A:** 50% reads and 50% updates. **B:** 95% reads and 5% updates. **C:** read-only. **D:** 95% latest reads and 5% writes. **E:** 95% scans and 5% writes. **F:** 50% reads and 50% read-modify-writes.

significant throughput benefits (as shown below), the minor extra write amplification of CC is deemed acceptable.

Throughput. Figure 9 shows the throughput in all YCSB workloads using different datasets. We also supplement YCSB A and B with the uniform access pattern (in addition to the default Zipfian). We first evaluate the load throughput. Across all datasets, CC achieves the best throughput, increasing by 3.6%, 43.7%, 3.58×, 3.21×, 75.1%, and 65.3%, compared to NO, Snappy, QAT, ZSTD, ZSTDPC, and Balloon, respectively. The performance gain is substantial compared to ZSTD and ZSTDPC (which has the best write volume reduction) due to efficient (de)compression loading during SST flush and compaction. Balloon has sub-optimal write performance due to limited flash parallelism each zone can use.

In other read-intensive YCSB workloads except Load, CC also demonstrates the best throughput, increasing by 10.2%, 11.7%, 32.0%, 24.1%, 20.2%, and 10.4% compared to other schemes. The performance gain of CC is mainly from (1) efficient (de)compression offloading and/or (2) high space savings using standard algorithms (e.g., ZSTD), resulting in fewer data stored in the LSM tree and thus higher lookup efficiency. Balloon not only restricts zone-to-flash parallelism but also generates two flash accesses for a proportion of read requests, resulting in sub-optimal read performance.

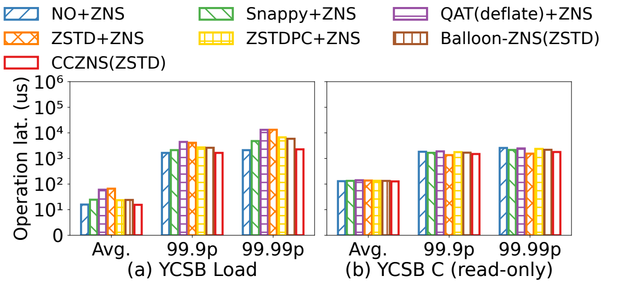


Fig. 10. **Average and tail I/O latency in YCSB Load and YCSB C.** The dataset is Amazon, while others show a similar trend.

TABLE IV
AVERAGE AND MAXIMUM CPU USAGE IN YCSB LOAD. THE DATASET IS AMAZON, WHILE OTHERS SHOW A SIMILAR TREND.

	N	S	Q	Z	ZP	B	CC
Avg. (%)	5.0	7.2	7.8	8.0	21.6	5.2	**5.8**
Max. (%)	12.0	17.1	19.4	17.9	50.0	11.8	**12.3**

TABLE V
AVERAGE SYSTEM MEMORY USAGE (IN GB) IN YCSB LOAD AND YCSB C (READ-ONLY). THE DATASET IS AMAZON, WHILE OTHERS SHOW A SIMILAR TREND.

	N	S	Q	Z	ZP	B	CC
Load	2.77	2.88	2.63	2.63	3.15	2.65	**3.24**
C	6.51	6.68	6.55	6.56	7.05	6.60	**7.25**

Latency. Figure 10 shows the average and tail latency in YCSB Load and C (read-only), respectively. For clear demonstration, the ticks on the y-axis are unequally spaced. In YCSB Load, CC achieves the best-level latency over other compression-enabled schemes, on par with NO. For example, compared to ZSTDPC, CC decreases the average, 99.9p, and 99.99p latency by 49.5%, 49.5%, and 56.8%, respectively. Compared to Balloon, the corresponding values are 50.0%, 47.9%, and 53.7%. In contrast, in YCSB C, the latency improvement in CC is smaller than YCSB Load because the overhead of decompression is less than that of compression.

Host CPU and Memory Usage. Table IV shows the average and maximum CPU usage in YCSB Load. Thanks to the offloaded (de)compression, CC achieves the best-level CPU usage, on par with NO and Balloon. Notably, compared to ZSTDPC, the average and maximum CPU usage is reduced by 73.1% and 75.4%, respectively.

Table V shows the average system memory usage in YCSB Load and C (read-only). While eliminating redundant indexes inside the SSD, CC requires more host memory due to the expanded index blocks. The index blocks are necessary to be maintained during both flush/compaction and key-value pair lookups, leading to a memory usage increase in both workloads. Despite this, the memory usage increment is acceptable. For example, compared to ZSTDPC, it is only 2.9% and 2.8% in YCSB Load and C, respectively.

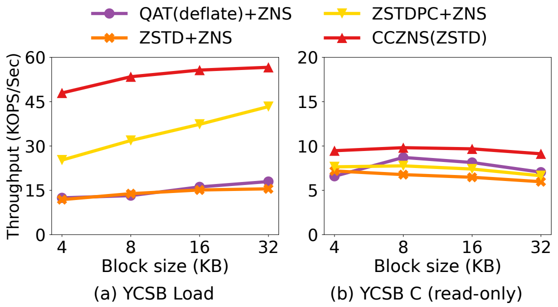


Fig. 11. **Throughput with different data block sizes in YCSB Load and YCSB C.** The dataset is Amazon, while others show a similar trend.

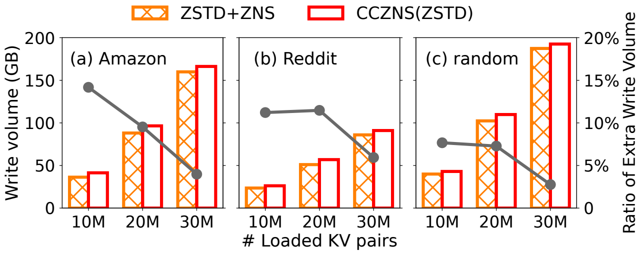


Fig. 12. **Write volumes and ratios of extra write volume (i.e., CC to ZSTD) in YCSB Load under different write loads using different datasets.** The grey lines with markers indicate the ratios of extra write volumes.

Summary of Results. CCZNS not only maximizes the benefits of compression in write volume and cost (with only minor loss), but also achieves high system performance in throughput, latency, and host CPU usage, while no other scheme can offer all these merits.

The side effect of CCZNS is the increase in host memory usage due to expanded index blocks. However, we believe consuming slightly more host memory is preferable to reintroducing high SSD memory (see §II-C) for two reasons. First, since only one field is added to the index entry, the memory usage increment is small. Second, the fast development of memory disaggregation and CXL provides ever-higher memory scalability for cloud and datacenter servers [43], [44], [66].

D. Extended Study on RocksDB

In this section, we analyze (1) the individual effects of multi-chunk commands (rather than single-chunk), (2) the performance sensitivity with different block sizes in RocksDB, (3) the extra write volumes under different write loads, and (4) the performance with extremely incompressible data.

Effects of multi-chunk CW/DR. The multi-chunk mode is proposed to enhance the scenarios where the chunks are small. Otherwise, each I/O can only accommodate one chunk, which increases the number of I/Os and thus leads to degraded system performance. To investigate the individual effects of multi-chunk commands, we separately replace the multi-chunk CW/DR with their single-chunk counterparts and evaluate the performance of CC.

When using multi-chunk CW and *single-chunk* DR, the throughput CC drops to only 55.2% due to the inefficient prefetching during SST flush and compaction stages. When using *single-chunk* CW and multi-chunk DR, CC fails to allocate storage space to accommodate 20M key-value pairs. In this case, every data block (the original size is around 4KB) after compression is padded to the logical block boundary, resulting in high space amplification.

Performance with varied block sizes. The size of the data block (i.e., block size) can affect the (de)compression performance. Thus, we vary the block size to investigate the performance sensitivity of CC by comparing it to the schemes using host CPU compression and external hardware compression. Figure 11 shows the throughput results in YCSB Load and C (read-only). In YCSB Load, CC substantially outperforms other schemes in all configurations, and the average throughput improvements are 3.6×, 3.8×, and 1.6×, respectively, compared to QAT, ZSTD, ZSTDPC. In YCSB C, the performance gain is smaller but still reaches 26.2%, 44.7%, and 29.6%, respectively.

We can also notice that the throughput of QAT can exceed that of ZSTD and ZSTDPC when the block size reaches 8KB in YCSB C. This is mainly because a larger block size can amortize the data round-trip overhead between host and the PCIe-attached QAT accelerator.

Extra write volumes under different write loads. As shown in Figure 8, despite with the same compression algorithm and level, CC incurs slightly more extra write volumes compared to the host-side counterparts (e.g., ZSTD) that achieve the best. To provide more insights on the extra write volume, we vary the number of loaded key-value pairs in 10M, 20M, and 30M, and compare the write volume of CC against ZSTD.

Figure 12 demonstrates the results. We can see that the ratio of extra write volume (i.e., CC to ZSTD) decreases as the loaded data volume increases. The main reason is that, as the loaded data volume rises, the inherent write amplification of the LSM structure increases significantly [82], making the impact of the extra write volume from CC on the overall write volume less. Specifically, when loading 30M key-value pairs, the ratios of extra write volume in three datasets are only 4.0%, 5.9%, and 2.8%, respectively.

Performance with extremely incompressible data. As discussed in §III-F, when the stored data is extremely incompressible, host software can simply switch to traditional read/write commands (rather than the proposed CW/DR) to avoid unnecessary performance overhead.

We experimentally confirm the performance efficiency of this approach under all evaluated workloads in Figure 9. Particularly, CC can achieve similar write volume and throughput compared to the case where compression is disabled (i.e., NO).

VI. Related Work

ZNS interface or ZNS SSD design. Prior works have examined a few limitations of the ZNS interface, including high GC overhead [46], [94], write performance deficiency [3], [24], [92], flash bandwidth underutilization [27], [50], [74], host-managed heterogeneous zone (e.g., SLC/QLC) support [91], and in-storage compression support [93]. With the same goal of Balloon-ZNS [93], CCZNS also aims to revive in-storage hardware compression but with a host-SSD collaborative approach, which fundamentally resolves the intrinsic challenges that the host-transparent methodology confronts.

While ZNS redraws the function responsibilities of host and SSD, the community is continuously rethinking better divisions. For example, to avoid unnecessary data transfer between host and SSD, host-side data migrations are advocated to be offloaded to the SSD via new commands [46], [53], [86]. Similarly, CCZNS unleashes the potential of in-storage hardware compression on ZNS SSDs by rationally shifting the indexing duty (originally on SSDs) back to the host.

In-storage computing. Prior works have widely explored in-storage computing by offloading filesystems [61], [83], [99], key-value stores [54], [58], [63], user-space cache [78], software RAID [95], and also compute-intensive tasks such as approximate nearest neighbor search (ANNS) [90] and compression [14], [16]. Our focus in this paper is compression offloading, which has gained industry support for integration into commercial SSD products.

FusionFS [99] proposes a general I/O abstraction (i.e., CISCOps) that combines multiple I/O and data processing operations for offloading. In contrast, CW and DR are specifically designed for compression offloading on ZNS SSDs, integrating compression semantics into existing traditional read/write commands and carefully designed to meet real-world compression requirements. CW and DR also incorporate a novel information-responding feature to facilitate host-level index management.

In-storage (hardware) compression. Built-in compression can improve the performance and lifetime of SSDs and has been extensively studied over the past decade [34], [56], [60], [65], [68], [77], [97], [101], [104]. This compression architecture becomes more promising when the powerful FPGA- or ASIC-based hardware engines are integrated inside the SSD (i.e., in-storage hardware compression), rather than by embedded ARM cores that have lower computational power.

Numerous efforts have been made to advance in-storage hardware compression. Many works focus on innovating host software to better leverage in-storage hardware compression, including rational database [32], [80], B+ tree [51], [81], hash-based key-value store [36], and log-structured filesystem [87]. From the hardware side, Lee et al. [65] and Park et al. [77] present FPGA-based hardware implementations as well as compression-aware FTL designs. Huang et al. introduce the designs of ASIC-based hardware compression engines [52]. Balloon-ZNS proposes compressibility-adaptive and slot-aligned storage management to enable in-storage hardware compression on ZNS SSDs host-transparently [93].

Existing works are all based on the SSD with the conventional host-transparent architecture. In contrast, we for the first time revisit the system architecture with in-storage hardware compression on the emerging ZNS SSD and propose to decouple compression indexing with execution for better cost and performance.

VII. Conclusion

This paper presents a new compression system architecture to revive in-storage hardware compression on ZNS SSDs, and the core idea is to leverage the existing fine-grained indexes at the host to directly manage compressed data inside the SSD. To this end, we propose CCZNS, an advanced ZNS interface, with novel designs on read/write commands and storage management, to revive in-storage hardware compression in a host-SSD collaborative manner. Comprehensive experiments on RocksDB and ZenFS demonstrate that the CCZNS-based storage system greatly outperforms existing system solutions.

Acknowledgments

We thank our shepherd, other anonymous reviewers, and You Zhou from Huazhong University of Science and Technology for their constructive feedback. This work is supported by the National Key R&D Program of China (No. 2023YFB4502900) and the Research Grants Council of Hong Kong SAR (Project No. CUHK14218522). The corresponding authors are Tao Lu and Ming-Chang Yang.

References

[1] "Reddit Comments and Posts between 2007 and 2013," https://www.reddit.com/r/datasets/comments/3bxlg7/i_have_every_publicly_available_reddit_comment/, 2015.

[2] "Understanding the Concepts Behind Virtual Data Optimizer (VDO) in RHEL 7.5 Beta," https://www.redhat.com/en/blog/understanding-concepts-behind-virtual-data-optimizer-vdo-rhel-75-beta, 2018.

[3] "Zone Append: A New Way of Writing to Zoned Storage," https://www.usenix.org/system/files/vault20_slides_bjorling.pdf, 2020.

[4] "Data Sheet of Western Digital Ultrastar DC ZN540," https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/product/ultrastar-dc-zn540-ssd/data-sheet-ultrastar-dc-zn540.pdf, 2021.

[5] "The Density, Cost, and Marketing of Semiconductor Memory," https://news.skhynix.com/the-density-cost-and-marketing-of-semiconductor-memory/, 2021.

[6] "A Host-Assisted Computational Storage Architecture," https://www.snia.org/sites/default/files/cmss/2023/SNIA-CMSS23-Lu-Host-Assisted-CS-Architecture.pdf, 2023.

[7] "Btrfs compression," https://archive.kernel.org/oldwiki/btrfs.wiki.kernel.org/index.php/Compression.html, 2024.

[8] "DapuStor Roealsen6 Products," https://en.dapustor.com/, 2024.

[9] "FIO benchmark," https://fio.readthedocs.io/en/latest/, 2024.

[10] "Intel® QuickAssist Technology Plugin for RocksDB Storage Engine," https://github.com/intel/qat-plugin-rocksdb, 2024.

[11] "Intel® QuickAssist Technology (QAT) QATzip Library," https://github.com/intel/QATzip, 2024.

[12] "LZ4: Extremely fast compression," https://lz4.org/, 2024.

[13] "RocksDB: A persistent key-value store for fast storage environments," http://rocksdb.org/, 2024.

[14] "ScaleFlux CSD Products," https://scaleflux.com/products/, 2024.

[15] "Silesia Corpus Datasets," http://www.data-compression.info/Corpora/SilesiaCorpus/index.html, 2024.

[16] "SmartSSD: Intelligent self-processing for the era of big data," https://semiconductor.samsung.com/ssd/smart-ssd/, 2024.

[17] "Snappy: A fast compressor/decompressor," https://google.github.io/snappy/, 2024.

[18] "Specification of Intel® QuickAssist Adapter 8970," https://www.intel.com/content/www/us/en/products/sku/125200/intel-quickassist-adapter-8970/specifications.html, 2024.

[19] "Specification of Intel® Xeon® Platinum 8458P Processor," https://www.intel.com/content/www/us/en/products/sku/231742/intel-xeon-platinum-8458p-processor-82-5m-cache-2-70-ghz/specifications.html, 2024.

[20] "SSDs with NVMe Zoned Namespace (ZNS) Support," https://zonedstorage.io/docs/introduction/zns, 2024.

[21] “ZenFS: RocksDB Storage Backend for ZNS SSDs and SMR HDDs,” https://github.com/westerndigitalcorporation/zenfs, 2024.

[22] “Zoned Storage: Kernel Configuration,” https://zonedstorage.io/docs/linux/config, 2024.

[23] “Zstandard: Real-time data compression algorithm,” https://facebook.github.io/zstd/, 2024.

[24] M. Allison, “What’s New in NVMe® Technology: Ratified Technical Proposals to Enable the Future of Storage,” in *Flash Memory Summit*, 2022.

[25] N. Aota, “File System Native Support of Zoned Block Devices: Regular vs Append Writes,” in *Storage Developer Conference (SDC)*, 2020.

[26] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, “Workload analysis of a large-scale key-value store,” in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, 2012, pp. 53–64.

[27] H. Bae, J. Kim, M. Kwon, and M. Jung, “What you can’t forget: exploiting parallelism for zoned namespaces,” in *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, 2022, pp. 79–85.

[28] B. Berg, D. S. Berger, S. McAllister, I. Grosof, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter *et al.*, “The {CacheLib} caching engine: Design and experiences at scale,” in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 753–768.

[29] M. Bjørling, “ZNS SSDs: Achieving Large-Scale Deployment,” in *Flash memory summit*, 2023.

[30] M. Bjørling, A. Aghayev, H. Holmberg, A. Ramesh, D. Le Moal, G. R. Ganger, and G. Amvrosiadis, “ZNS: Avoiding the Block Interface Tax for Flash-based SSD,” in *Proceedings of the USENIX Annual Technical Conference (ATC’21)*, 2021, pp. 689–703.

[31] M. Bjørling, J. Gonzalez, and P. Bonnet, “Lightnvm: The linux open-channel ssd subsystem,” in *15th USENIX Conference on File and Storage Technologies (FAST 17)*, 2017, pp. 359–374.

[32] 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 *18th USENIX conference on file and storage technologies (FAST 20)*, 2020, pp. 29–41.

[33] J. Chen, M. Daverveldt, and Z. Al-Ars, “Fpga acceleration of zstd compression algorithm,” in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2021, pp. 188–191.

[34] X. Chen, T. Lu, J. Wang, Y. Zhong, G. Xie, X. Cao, Y. Ma, B. Si, F. Ding, Y. Yang *et al.*, “Ha-csd: Host and ssd coordinated compression for capacity and performance,” in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2024, pp. 825–838.

[35] X. Chen, Y. Li, and T. Zhang, “Reducing flash memory write traffic by exploiting a few mbs of capacitor-powered write buffer inside solid-state drives (ssds),” *IEEE Transactions on Computers*, vol. 68, no. 3, pp. 426–439, 2019.

[36] X. Chen, N. Zheng, S. Xu, Y. Qiao, Y. Liu, J. Li, and T. Zhang, “Kallaxdb: A table-less hash-based key-value store on storage hardware with built-in transparent compression,” in *Proceedings of the 17th International Workshop on Data Management on New Hardware*, 2021, pp. 1–10.

[37] C. Choi and Y. Kang, “Green Computing with Computational Storage Devices,” in *Storage Developer Conference*, 2022.

[38] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with ycsb,” in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[39] N. Dayan, P. Bonnet, and S. Idreos, “Geckoftl: Scalable flash translation techniques for very large flash devices,” in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 327–342.

[40] P. Deutsch, “Rfc1951: Deflate compressed data format specification version 1.3,” 1996.

[41] J. Fowers, J.-Y. Kim, D. Burger, and S. Hauck, “A scalable high-bandwidth architecture for lossless compression on fpgas,” in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2015, pp. 52–59.

[42] R. Gao, Z. Li, G. Tan, and X. Li, “Beezip: Towards an organized and scalable architecture for data compression,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 133–148.

[43] D. Gouk, S. Lee, M. Kwon, and M. Jung, “Direct access,{High-Performance} memory disaggregation with {DirectCXL},” in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 287–294.

[44] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, “Efficient memory disaggregation with infiniswap,” in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 649–667.

[45] A. Gupta, Y. Kim, and B. Urgaonkar, “Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings,” *Acm Sigplan Notices*, vol. 44, no. 3, pp. 229–240, 2009.

[46] K. Han, H. Gwak, D. Shin, and J. Hwang, “ZNS+: Advanced zoned namespace interface for supporting in-storage zone compaction,” in *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI’21)*, 2021, pp. 147–162.

[47] D. Harnik, R. Kat, D. Sotnikov, A. Traeger, and O. Margalit, “To zip or not to zip: Effective resource usage for {Real-Time} compression,” in *11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013, pp. 229–241.

[48] H. Holmberg, “Accelerating RocksDB with NVMe Zoned SSDs,” in *Storage Developer Conference (SDC)*, 2019.

[49] X. Hu, F. Wang, W. Li, J. Li, and H. Guan, “Qzfs: Qat accelerated compression in file system for application agnostic and cost efficient data storage,” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 163–176.

[50] D. Huang, D. Feng, Q. Liu, B. Ding, W. Zhao, X. Wei, and W. Tong, “Splitzns: Towards an efficient lsm-tree on zoned namespace ssds,” *ACM Transactions on Architecture and Code Optimization*, vol. 20, no. 3, pp. 1–26, 2023.

[51] K. Huang, Z. Shen, Z. Shao, T. Zhang, and F. Chen, “Breathing new life into an old tree: Resolving logging dilemma of b+-tree on modern computational storage drives,” *Proceedings of the VLDB Endowment*, vol. 17, no. 2, pp. 134–147, 2023.

[52] Y. Huang, A. Song, C. Guo, and Y. Yang, “Asic design of lz77 compressor for computational storage drives,” *Electronics Letters*, vol. 59, no. 22, p. e13000, 2023.

[53] J.-Y. Hwang, S. Kim, D. Park, Y.-G. Song, J. Han, S. Choi, S. Cho, and Y. Won, “{ZMS}: Zone abstraction for mobile flash storage,” in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024, pp. 173–189.

[54] J. Im, J. Bae, C. Chung, S. Lee *et al.*, “{PinK}: High-speed in-storage key-value store with bounded tails,” in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 173–187.

[55] G. Jeong, B. Sharma, N. Terrell, A. Dhanotia, Z. Zhao, N. Agarwal, A. Kejariwal, and T. Krishna, “Characterization of data compression in datacenters,” in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2023, pp. 1–12.

[56] C. Ji, L.-P. Chang, L. Shi, C. Gao, C. Wu, Y. Wang, and C. J. Xue, “Lightweight data compression for mobile flash storage,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, pp. 1–18, 2017.

[57] 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.

[58] Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson, “Kaml: A flexible, high-performance key-value ssd,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 373–384.

[59] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho, “The multi-streamed {Solid-State} drive,” in *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, 2014.

[60] M. Kang, W. Lee, J. Kim, and S. Kim, “Pr-ssd: Maximizing partial read potential by exploiting compression and channel-level parallelism,” *IEEE Transactions on Computers*, vol. 72, no. 3, pp. 772–785, 2022.

[61] S. Kannan, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Y. Wang, J. Xu, and G. Palani, “Designing a true {Direct-Access} file system with {DevFS},” in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018, pp. 241–256.

[62] S. Karandikar, A. N. Udipi, J. Choi, J. Whangbo, J. Zhao, S. Kanev, E. Lim, J. Alakuijala, V. Madduri, Y. S. Shao *et al.*, “Cdpu: Co-designing compression and decompression processing units for hy-

perscale systems," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–17.

[63] J. Koo, J. Im, J. Song, J. Park, E. Lee, B. S. Kim, and S. Lee, "Modernizing file system through in-storage indexing," in *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021, pp. 75–92.

[64] N. Lazarev, V. Gohil, J. Tsai, A. Anderson, B. Chitlur, Z. Zhang, and C. Delimitrou, "Sabre:{Hardware-Accelerated} snapshot compression for serverless {MicroVMs}," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 1–18.

[65] S. Lee, J. Park, K. Fleming, J. Kim *et al.*, "Improving performance and lifetime of solid-state drives using hardware-accelerated compression," *IEEE Transactions on consumer electronics*, vol. 57, no. 4, pp. 1732–1739, 2011.

[66] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal *et al.*, "Pond: Cxl-based memory pooling systems for cloud platforms," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 574–587.

[67] H. Li, M. Hao, M. H. Tong, S. Sundararaman, M. Bjørling, and H. S. Gunawi, "The CASE of FEMU: Cheap, accurate, scalable and extensible flash emulator," in *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*, 2018, pp. 83–90.

[68] J. Li, K. Zhao, X. Zhang, J. Ma, M. Zhao, and T. Zhang, "How much can data compressibility help to improve {NAND} flash memory lifetime?" in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015, pp. 227–240.

[69] Q. Li, Q. Xiang, Y. Wang, H. Song, R. Wen, W. Yao, Y. Dong, S. Zhao, S. Huang, Z. Zhu *et al.*, "More than capacity: performance-oriented evolution of pangu in alibaba," in *21st USENIX Conference on File and Storage Technologies (FAST 23)*, 2023, pp. 331–346.

[70] Y. Li, A. Kashyap, Y. Guo, and X. Lu, "Characterizing lossy and lossless compression on emerging bluefield dpu architectures," in *2023 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE, 2023, pp. 33–40.

[71] C.-Y. Liu, J. Kotra, M. Jung, and M. Kandemir, "{PEN}: Design and evaluation of {Partial-Erase} for 3d {NAND-Based} high density {SSDs}," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018, pp. 67–82.

[72] V. S. Mailthody, Z. Qureshi, W. Liang, Z. Feng, S. G. De Gonzalo, Y. Li, H. Franke, J. Xiong, J. Huang, and W.-m. Hwu, "Deepstore: In-storage acceleration for intelligent queries," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 224–238.

[73] J. J. McAuley and J. Leskovec, "From amateurs to connoisseurs: modeling the evolution of user expertise through online reviews," in *Proceedings of the 22nd international conference on World Wide Web*, 2013, pp. 897–908.

[74] J. Min, C. Zhao, M. Liu, and A. Krishnamurthy, "{eZNS}: An elastic zoned namespace for commodity {ZNS}{SSDs}," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 461–477.

[75] A. Ozsoy, M. Swany, and A. Chauhan, "Pipelined parallel lzss for streaming data compression on gpgpus," in *2012 IEEE 18th International Conference on Parallel and Distributed Systems*. IEEE, 2012, pp. 37–44.

[76] J. Park, M. Kim, M. Chun, L. Orosa, J. Kim, and O. Mutlu, "Reducing solid-state drive read latency by optimizing read-retry," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 702–716.

[77] Y. Park and J.-S. Kim, "zftl: Power-efficient data compression support for nand flash-based consumer electronics devices," *IEEE transactions on consumer electronics*, vol. 57, no. 3, pp. 1148–1156, 2011.

[78] L. Peng, Y. An, Y. Zhou, C. Wang, Q. Li, C. Cheng, and J. Zhang, "{ScalaCache}: Scalable {User-Space} page cache management with {Software-Hardware} coordination," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024, pp. 1185–1202.

[79] W. Qiao, J. Du, Z. Fang, M. Lo, M.-C. F. Chang, and J. Cong, "High-throughput lossless compression on tightly coupled cpu-fpga platforms," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018, pp. 37–44.

[80] Y. Qiao, X. Chen, J. Hao, J. Li, Q. Wu, J. Wang, Y. Liu, and T. Zhang, "Improving relational database upon the arrival of storage hardware with built-in transparent compression," in *2021 IEEE International Conference on Networking, Architecture and Storage (NAS)*. IEEE, 2021, pp. 1–9.

[81] Y. Qiao, X. Chen, N. Zheng, J. Li, Y. Liu, and T. Zhang, "Closing the b+-tree vs.{LSM-tree} write amplification gap on modern storage hardware with built-in transparent compression," in *20th USENIX Conference on File and Storage Technologies (FAST 22)*, 2022, pp. 69–82.

[82] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "Pebblesdb: Building key-value stores using fragmented log-structured merge trees," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 497–514.

[83] Y. Ren, C. Min, and S. Kannan, "{CrossFS}: A cross-layered {Direct-Access} file system," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 137–154.

[84] C. Sabol, R. Stenfort, and M. Allison, "Flexible Data Placement: State of the Union," in *Flash Memory Summit*, 2023.

[85] E. Sitaridi, R. Mueller, T. Kaldewey, G. Lohman, and K. A. Ross, "Massively-parallel lossless data decompression," in *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE, 2016, pp. 242–247.

[86] S. Somasundaram, "Towards Copy-Offload in Linux NVMe," in *Storage Developer Conference (SDC)*, 2021.

[87] Y. Song, Y. Huang, Y. Lv, Y. Zhang, and L. Shi, "When f2fs meets compression-based ssd!" in *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, 2023, pp. 87–92.

[88] R. Stoica, R. Pletka, N. Ioannou, N. Papandreou, S. Tomic, and H. Pozidis, "Understanding the design trade-offs of hybrid flash controllers," in *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2019, pp. 152–164.

[89] J. Sun, S. Li, Y. Sun, C. Sun, D. Vucinic, and J. Huang, "Leaftl: A learning-based flash translation layer for solid-state drives," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 442–456.

[90] B. Tian, H. Liu, Z. Duan, X. Liao, H. Jin, and Y. Zhang, "Scalable billion-point approximate nearest neighbor search using {SmartSSDs}," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024, pp. 1135–1150.

[91] Y. Wang, L. Y. Chow, X. Nie, Y. Liang, and M.-C. Yang, "Znh2: Augmenting zns-based storage system with host-managed heterogeneous zones," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, 2024.

[92] Y. Wang, Y. Zhou, F. Wu, J. Zhang, and M.-C. Yang, "Land of oz: Resolving orderless writes in zoned namespace ssds," *IEEE Transactions on Computers*, 2024.

[93] Y. Wang, Z. Sun, Y. Zhou, T. Lu, C. Xie, and F. Wu, "Balloon-zns: Constructing high-capacity and low-cost zns ssds with built-in compression," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–6.

[94] Y. Wang, Y. Zhou, Z. Lu, X. Zhang, K. Wang, F. Zhu, S. Li, C. Xie, and F. Wu, "Flexzns: Building high-performance zns ssds with size-flexible and parity-protected zones," in *Proceedings of the 41st IEEE International Conference on Computer Design (ICCD'23)*, 2023.

[95] Y. Wen, X. Zhao, Y. Zhou, T. Zhang, S. Yang, C. Xie, and F. Wu, "Eliminating storage management overhead of deduplication over ssd arrays through a hardware/software co-design," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 320–335.

[96] J. Yang, Y. Yue, and K. Rashmi, "A large scale analysis of hundreds of in-memory cache clusters at twitter," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 191–208.

[97] X. Yao, Q. Li, K. Lin, X. Gan, J. Zhang, C. Gao, Z. Shen, Q. Xu, C. Yang, and J. Xue, "Extremely-compressed ssds with i/o behavior prediction," *ACM Transactions on Storage*, 2024.

[98] Y. Yuan, R. Wang, N. Ranganathan, N. Rao, S. Kumar, P. Lantz, V. Sanjeepan, J. Cabrera, A. Kwatra, R. Sankaran *et al.*, "Intel accelerators ecosystem: An soc-oriented perspective: Industry product,"

in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 848–862.

[99] J. Zhang, Y. Ren, and S. Kannan, "{FusionFS}: Fusing {I/O} operations using {CISCOps} in firmware file systems," in *20th USENIX Conference on File and Storage Technologies (FAST 22)*, 2022, pp. 297–312.

[100] T. Zhang, "The True Value of Storage Drives with Built-in Transparent Compression: Far Beyond Lower Storage Cost," in *Storage Developer Conference*, 2020.

[101] X. Zhang, J. Li, H. Wang, K. Zhao, and T. Zhang, "Reducing {Solid-State} storage device write stress through opportunistic in-place delta compression," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016, pp. 111–124.

[102] B. Zhou, H. Jin, and R. Zheng, "A high speed lossless compression algorithm based on cpu and gpu hybrid platform," in *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2014, pp. 693–698.

[103] Y. Zhou, E. Xu, L. Zhang, K. Karkra, M. Barczak, W. Gao, W. Malikowski, M. Kozlowski, Ł. Łasek, R. Lu *et al.*, "Csal: the next-gen local disks for the cloud," in *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024, pp. 608–623.

[104] A. Zuck, S. Toledo, D. Sotnikov, and D. Harnik, "Compression and SSDs: Where and how?" in *Proceedings of the 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW'14)*, 2014.