ZipCache: A DRAM/SSD Cache with Built-in Transparent Compression

Rui Xie
Rensselaer Polytechnic Institute
Troy, NY, USA
xier2@rpi.edu

Linsen Ma
Rensselaer Polytechnic Institute
Troy, NY, USA
mal3@rpi.edu

Alex Zhong
Harker School
San Jose, CA, USA
25alexz@students.harker.org

Feng Chen
Louisiana State University
Baton Rouge, LA, USA
fchen@csc.lsu.edu

Tong Zhang
Rensselaer Polytechnic Institute
Troy, NY, USA
tzhang@ecse.rpi.edu

Abstract

As a core component in modern data centers, key-value cache provides high-throughput and low-latency services for high-speed data processing. The effectiveness of a key-value cache relies on its ability of accommodating the needed data. However, expanding the cache capacity is often more difficult than commonly expected because of many practical constraints, such as server costs, cooling issues, rack space, and even human resource expenses. A potential solution is *compression*, which virtually extends the cache capacity by condensing data in cache. In practice, this seemingly simple idea has not gained much traction in key-value cache system design, due to several critical issues: the compression-unfriendly index structure, severe read/write amplification, wasteful decompression operations, and heavy computing cost. This paper presents a hybrid DRAM-SSD cache design to realize a systematic integration of data compression in key-value cache. By treating compression as an essential component, we have redesigned the indexing structure, data management, and leveraged the emerging computational SSD hardware for collaborative optimizations. We have developed a prototype, called ZipCache. Our experimental results show that ZipCache can achieve up to 72.4% higher throughput and 42.4% lower latency, while reducing the write amplification by up to 26.2 times.

CCS Concepts

• **Information systems → Database design and models**.

Keywords

Key-Value Cache, Data Compression, DRAM/SSD Cache, Computational SSD

ACM Reference Format:
Rui Xie, Linsen Ma, Alex Zhong, Feng Chen, and Tong Zhang. 2024. ZipCache: A DRAM/SSD Cache with Built-in Transparent Compression. In *The International Symposium on Memory Systems (MEMSYS '24), September 30-October 3, 2024, Washington, VA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3695794.3695805

1 Introduction

Key-value cache plays a crucial role in providing high-throughput, low-latency data services. Major Internet service providers, such as Google and Meta, often deploy a fleet of cache servers as the first line of defense to handle a massive influx of requests for key-value data, a type of unstructured data organized in simple forms as *keys* and *values* (e.g., "User ID" and "User name"). Key-value caching can accelerate data retrievals and alleviate the traffic to backend databases by serving from high-speed storage medium, typically DRAM, flash memory, or a combination of both, such as Meta's CacheLib [8] and Ximalaya's xcache [3].

The effectiveness of key-value caching hinges on its ability of accommodating the requested data in cache. While a larger cache intuitively leads to improved performance, expanding the capacity of key-value caches within data centers is often not simply a matter of hardware upgrade. Many practical constraints, such as server costs, cooling expenses, rack space, real estate limitations, and even human resource expenses must be taken into consideration. Let us consider the hardware cost as an example: Microsoft Azure reports that DRAM constitutes 50% of their server costs [48], and Meta reports a similar trend (40% of the rack cost) [46]; Although the prices of high-speed NVMe SSDs are comparatively lower, they are still rather substantial [15, 49, 58]. Relying solely on hardware investment to increase cache capacity is apparently not a sustainable, cost-effective approach to keep up with the rapid growth of data. This poses an increasingly severe challenge in today's data centers.

A potential solution is *compression.* By condensing data to occupy a smaller footprint, one could *virtually* expand the cache capacity, allowing cache to hold more data, which in turn increases the cache hit ratio. Despite the adoption of data compression in computing systems in prior studies [25, 31, 52, 62], interestingly, this simple idea has not gained much traction in key-value cache design. We believe that this lack of adoption in practice is due to several unique and critical issues inherent in key-value cache systems:

MEMSYS '24, September 30-October 3, 2024, Washington, VA, USA

ACM ISBN 979-8-4007-1091-9/24/09
https://doi.org/10.1145/3695794.3695805

- **Issue #1: The commonly used hash indexing causes random, compression-unfriendly data placement**. Most key-value cache systems adopt a *hash index* based structure to manage the key-value data [6, 21, 44]. With hash indexing, keys are randomly dispersed in a flat, shallow data structure, which is advantageous for quick search in a large key space but comes with a detrimental effect for compression: Due to the nature of hash functions, the keys are evenly distributed, leaving unrelated data randomly mingled together. Such data layout is inherently difficult for effective data compression, as compression algorithms heavily rely on organizing similar data content within a close proximity.
- **Issue #2: The structure designed for managing small-size key-value items introduces a severe read/write amplification problem**. Key-value workloads are known to be dominated by small-size data items. According to a study from Meta/Facebook, the majority of key-value items are (much) smaller than 500 bytes [16]. Since compressing each individual small key-value item yields limited or no benefits in size reduction, achieving effective compression requires to pack a collection of small key-value items for a reasonable compression ratio. However, this "optimization" would result in a substantial, undesirable increase in access operations, i.e., *read/write amplification*, when reading or updating a small key-value item in a much larger compression unit.
- **Issue #3: Compression and decompression are simply treated as two opposite processes on the same unit of data**. As a common practice, a block of data is compressed and decompressed as a full, single unit. As we increase the compression granularity to reduce indexing costs and increase compression ratios, the efficiency of decompression process unfortunately diminishes. This is because the more data is compressed, the more needs to be decompressed, leading to a proportionally increased amount of data accesses and longer delays to decompress and locate the requested key-value item.
- **Issue #4: Compression imposes a heavy computing cost and interferes other data-processing tasks**. It is well-known that compression is computation intensive, essentially trading computation for storage capacity. Conducting data compression and decompression operations on general-purpose CPUs not only increases the burden on limited CPU resources but also causes disruptive effect on foreground service operations, potentially reducing the overall system throughput and increasing user-perceived delays. Considering the stringent requirement for cache latency, the additional delay is a non-trivial overhead that must be considered and mitigated.

All the above-said issues pose a critical challenge to the current key-value cache systems, calling for a full consideration of data compression as an essential component in the cache system design. In this paper, we propose a new scheme, called *ZipCache*, to realize a systematic integration of data compression in the key-value cache system design. By treating data compression as an integral component, we have redesigned the indexing structure, data management, and leverage cutting-edge hardware for collaborative optimizations. Specifically, we take several important measures to achieve systematic optimizations for data compression:

Firstly, we abandon the conventional hash indexing structure and adopt a seemingly more "costly" B+ tree based structure to manage key-value items. This enables us to preserve content similarity and retain the spatial locality. Secondly, we introduce a sparse structure, called *super-leaf*, to store key-value data for compression in a virtualized SSD storage space. This leverages the emerging commercial SSDs with built-in transparent compression to maintain low-cost indexing without wasting any physical storage space. Thirdly, we decouple the data units for compression and decompression by creating a special intra-page structure for *just-in-need decompression*. This method facilitates early termination, significantly reducing decompression time and read amplification. Lastly, we fully exploit the abilities of the emerging computational SSDs with built-in transparent compression to offload heavy-cost data compression operations from the CPU to the storage device, which alleviates the computing burden and removes potential interference. To the best of our knowledge, this is the first work introducing hardware-assisted data compression into a hybrid DRAM/SSD key-value cache system.

We have implemented a prototype of ZipCache, which is a hybrid key-value cache with two cache layers, a DRAM layer and a flash memory layer. We use ScaleFlux's CSD3000 SSD [11] with hardware acceleration for transparent on-device compression. Our experiments show promising results. In comparison to the state-of-the-art solutions, including CacheLib [19] and xcache [3] and Kangaroo [47], our evaluation results demonstrate that, with a design carefully optimized for data compression, ZipCache can achieve up to 72.4% higher throughput and 42.4% lower 90-percentile read latency, and reduces the SSD write amplification by up to 26.2×. It is our hope that this work will motivate more future research to explore the full potential of the *long-overlooked* block compression for performance-critical caching systems.

The rest of this paper is organized as follows. Section 2 introduces background. Section 3 and 4 present our design and the experimental results. Section 5 discusses the related work. The last section concludes this paper.

2 Background and Motivation

2.1 Data Compression

General-purpose block compression is realized by *deduplicating* repeated byte strings in a data block, referred to as *LZ search* [64, 65]. Although CPU-based *LZ search* suffers from low speed due to the high CPU cache miss rate, its reverse process (i.e., LZ decompression) can be much faster. The compression block size affects the trade-off between compression ratio[1] and (de)compression speed. Using the file *samba* in *Silesia* corpus [2] as test data, Fig. 1 shows the LZ4 compression ratio and (de)compression latency under different block sizes. It shows about 4× speed performance difference between compression and decompression. As the compression block size continues to increase, the compression ratio first significantly improves and then gradually saturates.

The decompression process scans through the LZ-compressed byte stream to sequentially reconstruct the original data block. In theory, this process could terminate at any byte location, leading to a *partially* reconstructed data block. This makes it possible to

[1] In this work, we define *compression ratio* as $\frac{S_{orig}}{S_{comp}} \geq 1$, where S_{orig} and S_{comp} denote the size of the original and compressed data blocks.

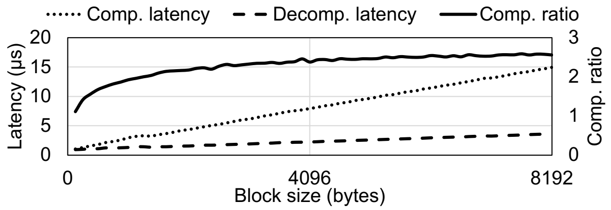


Figure 1: Comparison of compression ratio, (de)compression latency under different compression block size.

realize *early termination* of decompression: Suppose *LZ search* compresses an n-segment data block $\mathbf{D} = [b_1, b_2, \cdots, b_n]$ into $\mathbf{C}$. When decompressing $\mathbf{C}$, the original data $[b_1, \cdots, b_k]$ are successively reconstructed as k grows from 1 to n. Let τ denote the latency of decompressing $\mathbf{C}$ to reconstruct the entire $\mathbf{D}$. If the decompression process terminates once after the first m segments $[b_1, \cdots, b_m]$, where $m \leq n$) have been reconstructed, we define its early termination factor $\gamma(m, n) = m/n$, and the decompression latency is about $\gamma(m, n) \cdot \tau$. Suppose we are only interested in obtaining one cache object in the m-th segment, we could reduce the latency by $1 - m/n$ via decompression early termination. Using file *samba* in *Silesia* corpus [2] as test data, we partitioned each 4KB data block into 256B ($n = 16$) segments and measured the average LZ4 decompression latency under different early termination factor $\gamma(m, n)$ as shown in Table 1, which reveals substantial performance benefits.

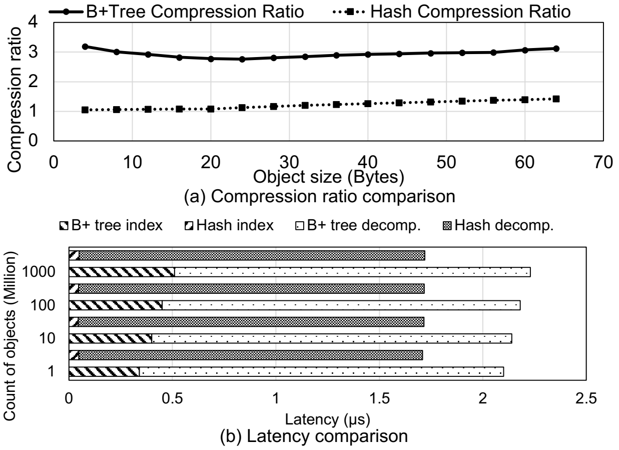
Table 1: Decompression latency under different $\gamma(m, n)$.

$\gamma(m, n)$	1/16	2/16	4/16	8/16	16/16
Latency (μs)	0.10	0.22	0.31	0.54	1.48

2.2 Cache Index Data Structure

Most in-memory data stores (e.g., Redis [6], FASTER [21], MICA [44]) use hash index to reduce the latency and simplify the implementation. In contrast, most storage-based data stores (e.g., RocksDB [7, 29], WiredTiger [12], Bw-tree [41]) employ tree index to reduce the index memory usage and embrace the storage block I/O interface. As for hybrid-DRAM/SSD caches, Meta's CacheLib [8, 19] uses hash index for the DRAM and SSD tiers, while Redis-compatible xcache [3] (developed based on Redis [6]) and Pika/RocksDB [9] employs hash index for DRAM tier and log-structured merge (LSM) tree [51] index for SSD tier.

The index structure has a substantial effect on the efficacy of data compression. Since compression ratio is proportional to the byte content similarity, a tree index that sorts all the cache objects based on their keys is clearly more beneficial, in comparison to hash index that randomly hashes cache objects into data blocks. For the purpose of demonstration, Fig. 2(a) shows the 4KB block compression ratio under hash index and B+ tree index. The key is 8B unix timestamp and object size ranging from 4B to 64B, which are both extracted from the Bitstamp Exchange Data [4]. By keeping objects with adjacent keys together, B+ tree achieves over 2× higher compression ratio than hash index. Although B+ tree index has a longer traversing latency than hash index, decompression tends to dominate the overall data access latency, which largely reduces the overall latency gap between B+ tree and hash index as shown in Fig. 2(b). A strong implication is that, although prior work on in-memory cache design widely adopted hash index, integrating block compression into in-memory cache makes tree index a favorable choice.



(a) Compression ratio comparison

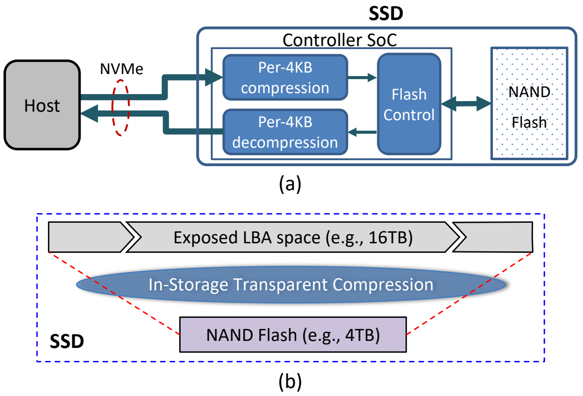
(b) Latency comparison

Figure 2: (a) Compression ratio of 4KB blocks under hash index and B+ tree index, and (b) latency of index traversing and block decompression under different total number of cache objects (hence different B+ tree depth), where key and values are obtained from Bitstamp Exchange Data [4].

Meta's CacheLib uses hash index for SSD-resident objects by directly hashing each object to a 4KB SSD LBA (logical block address) block without an in-memory hash table. This makes CacheLib subject to a high SSD write amplification: each object insert/update invokes re-writing a 4KB LBA block. As a variant of CacheLib, Kangaroo [47] applies write-ahead log (WAL) to amortize the SSD write cost by buffering multiple cache objects hashed to the same LBA. Although it can reduce the SSD write amplification, the storage and management of WAL introduce non-negligible overhead in terms of SSD capacity and CPU/memory usage.

2.3 In-Storage Transparent Compression

Data compression in a hybrid cache system imposes additional computation at both DRAM and flash cache tiers. Due to the lack of hardware support, compression over DRAM cache tier must be handled by host CPU, while the SSD-tier compression can be offloaded to the emerging computational SSDs with built-in transparent compression. Fig. 3(a) illustrates the structure of such SSDs [11], where the controller SoC (system on chip) (de)compresses each 4KB LBA block along the I/O path and manages the placement of compressed blocks on NAND flash memory. Host CPU accesses the SSD through standard I/O interface (e.g., NVMe). The dedicated hardware engines on the controller SoC implement per-4KB (de)compression at the latency of a few microseconds, which is over 10× shorter than the TLC/QLC NAND flash memory read (about 50μs and above) and write latency (about 1ms and above). Therefore, SSDs with built-in transparent compression can maintain the same IOPS (IO per second) and latency performance as ordinary SSDs. Such SSDs expose an expanded *logical* storage space that is larger (e.g., by 2× or 4×) than the physical NAND flash storage capacity, as illustrated in Fig. 3(b). This unique feature enables unique opportunities for our optimization efforts.



(a)

(b)

Figure 3: An illustration of (a) an SSD with built-in transparent compression, and (b) the expanded LBA space.

3 Design

We have designed a hybrid key-value cache solution, called *ZipCache*, with highly efficient built-in block compression. In this section, we will first introduce its basic architecture, then present a set of design techniques for improving its implementation efficiency, and finally describe its major operations.

3.1 Architecture Overview

ZipCache is a hybrid cache with two cache tiers. As illustrated in Fig. 4, ZipCache employs B+ tree index data structure to manage both its DRAM and SSD cache tiers, performing block compression on the B+ tree leaf pages. As mentioned in Section 2.2, a key benefit with B+ tree indexing is that all cache objects are *sorted* with their keys, enabling significantly higher compression ratios than its hash-based counterpart.

ZipCache is optimized for handling massive amount of small key-value items, which is not only practically important [19] but also poses significant challenges. ZipCache categorizes cache objects into three different size classes *tiny*, *medium*, and *large* by using pre-defined thresholds (e.g., 128B and 2KB). The three types of key-value items are handled differently: Tiny- and medium-size objects are stored across the DRAM and SSD tiers, while large-size objects are always SSD-resident, which is for maximizing the DRAM cache tier hit ratio. We compress in-memory tiny-size objects together in the unit of tree pages, and compress each in-memory medium-size object individually. In order to manage the different key-value items in DRAM and SSD cache tiers, ZipCache maintains three B+ trees:

(1) BT_{DRAM} for DRAM cache: This index structure entirely resides in host DRAM. Its compressed leaf pages hold tiny-size objects and pointers that point to in-memory compressed medium-size objects.
(2) BT_{SSD} for SSD cache: Its leaf pages hold tiny/medium-size objects and are resident in SSD, and all its non-leaf pages reside in host DRAM.
(3) BT_{LO} for indexing large-size objects: It entirely resides in host DRAM, and its leaf pages hold pointers that point to SSD-resident large-size objects.

ZipCache deploys its SSD cache tier over SSDs with built-in transparent compression, meaning that the compression of all the SSD-resident objects is transparently handled by SSDs. To minimize the SSD write amplification and leverage the huge DRAM-SSD bit cost gap (more than 20×), ZipCache adopts the inclusive caching policy over its two tiers, meaning that a key-value item could be held in both tiers. Due to the distinct characteristics of DRAM and SSD, the two cache tiers face different issues and challenges. Hence we will present the design of DRAM and SSD cache tiers separately in the following subsections.

3.2 DRAM Cache Tier

DRAM cache tier relies on host CPUs to (de)compress each B+ tree leaf page, and a leaf page should be large enough (e.g., 4KB) to ensure high compression ratio. As shown in the previous section, since (de)compressing a 4KB data block incurs a much higher overhead than traversing in-memory B+ tree (only a few hundred nanoseconds), we introduce the following three techniques to mitigate the (de)compression-induced overheads:

Decompression early termination. Motivated by the observation that the decompression time is almost proportional to the amount of decompressed data, we introduce a hash-assisted method to realize early termination of the decompression process. Let $\mathbf{P} = [\mathbf{p}_1, \mathbf{p}_2, \cdots, \mathbf{p}_n]$ denote one original (uncompressed) B+ tree leaf page, and $\mathbf{C}$ denote the compressed version of $\mathbf{P}$. To obtain a cache object within the leaf page $\mathbf{P}$, if we *a priori* know that the cache object locates in the sub-page $\mathbf{p}_m$, we can reduce the cache read latency by roughly $1 - m/n$ via decompression early termination.

To realize decompression early termination, we must be able to know which sub-page $\mathbf{p}_i$ contains the requested cache object before performing decompression, which however is impossible if we construct B+ tree leaf pages with conventional practice [34]. To address this issue, we construct each B+ tree leaf page in a hash-based manner as illustrated in Fig. 5: Let $\mathcal{K}$ denote the cache object key space and define a hash function $f : \mathcal{K} \rightarrow [1, n]$. For cache objects that fall into one B+ tree leaf page, we use the hash function f to calculate their destined sub-pages inside the page. Therefore, to fetch a cache object from a B+ tree leaf page, we can determine its associated sub-page through a simple hashing and hence accordingly configure the decompression early termination.

A side effect is that due to the nature of hashing and varying cache object size, a sub-page may only be partially filled with cache objects. However, padding the unused space with all zeros, our page compression process can almost entirely eliminate this potential memory space waste (see Fig. 5). If a sub-page is completely filled up, the B+ tree leaf page is split into two new B+ tree leaf pages, each storing roughly half of the cache objects in the original leaf page.

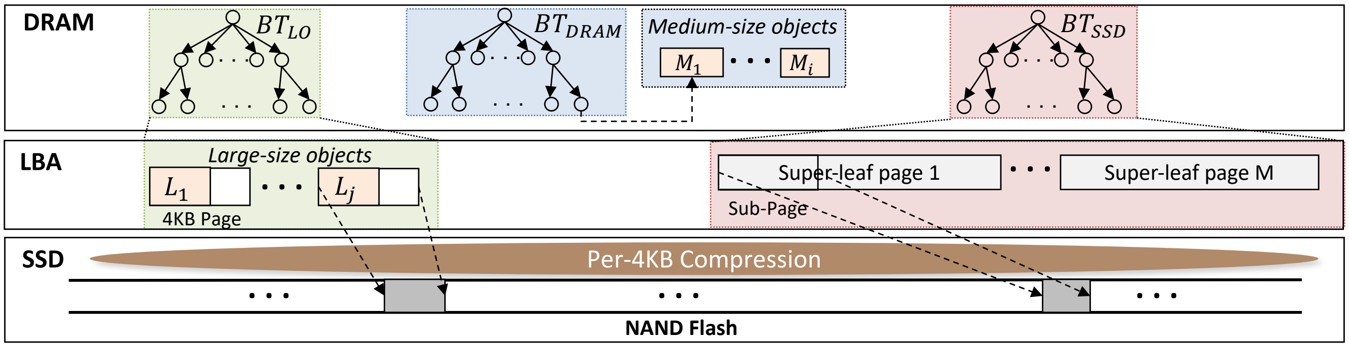


Figure 4: Overview of ZipCache architecture that employ three B+ trees to manage the DRAM cache tier, SSD cache tier, and large-size cache objects respectively.

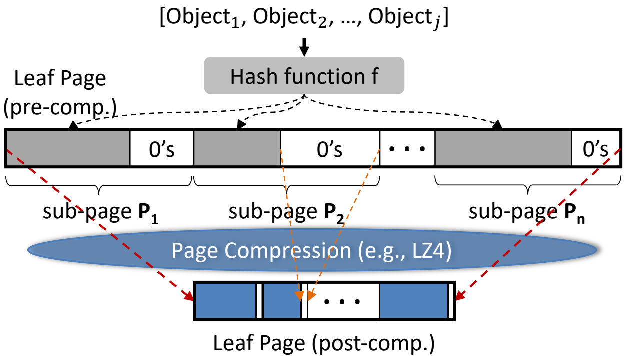


Figure 5: Illustration of hash-based mapping between cache objects and sub-pages in a leaf page of BT_{DRAM}, including the leaf page decompression early termination.

Adaptive compression bypassing. Intuitively, repeatedly compressing and decompressing *hot* B+ tree leaf pages would impose undesirable overhead, especially for workloads with highly skewed access pattern. We create two sub-tiers in the DRAM cache layer to adapt to runtime workloads: (1) An *uncompressed* sub-tier contains a small number of hot B+ tree leaf pages in their original, uncompressed form, and (2) a *compressed* sub-tier contains the rest leaf pages in the compressed form. To avoid sacrificing the DRAM cache tier hit ratio, we adaptively adjust the *uncompressed* hot-page sub-tier capacity according to the degree of workload locality. The uncompressed sub-tier could be completely eliminated in absence of sufficient locality.

The uncompressed sub-tier contains hot data. ZipCache adaptively auto-tunes its capacity as follows. Each B+ tree leaf page is associated with a per-page counter for tracking the runtime access intensity. The counters are periodically right-shifted by one bit to avoid overflow and age out obsolete accesses. Let μ_D and σ_D denote the mean and deviation of the page access intensity, the uncompressed sub-tier only contains leaf pages whose access intensity exceeds the threshold of $\mu_D + r \cdot \sigma_D$, where $r > 1$ is a design parameter. By setting r sufficiently large (e.g., 3), we can ensure only a small number of leaf pages could possibly reside in the uncompressed sub-tier. To further improve the adaption to runtime workloads, we may dynamically fine-tune the parameter r. In particular, we can vary the value of r and then monitor its effect on the overall cache performance. Optimization algorithm (e.g., simulated annealing) can be used to search the value of r that better matches the runtime workload characteristics.

Per-page write buffering. Inserting or updating a cache object in a compressed B+ tree leaf page needs to first decompress the entire page, insert/update the cache object in the page, and then re-compress the entire page. Such read/write amplification leads to a high operational overhead. We mitigate this issue by maintaining a write buffer to temporarily hold multiple insert/update to the same B+ tree leaf page and merge them together into the page through a single round of page decompression-modification-compression. This essentially trades extra memory usage for lower compression-induced implementation overhead. Once the buffer size exceeds a pre-defined threshold (e.g., 128B or 256B), the corresponding B+ tree leaf page is marked as a candidate for background compaction. To further reduce the interference with foreground operations, the page compaction operations are handled by background threads. We note that the aggregated runtime write buffer memory usage largely depends on the spatial locality of write requests. Under a workload with high spatial locality, only a small percentage of leaf pages undergo intensive updates, and write buffering can effectively remove overhead caused by unnecessary re-compression.

3.3 SSD Cache Tier

The emerging computational SSD provides built-in transparent compression [11], which brings multiple technical advantages. It not only offloads the resource-demanding (de)compression operations from the host CPUs, but also relieves the cache system from the complexities of handling the storage of variable-length post-compression data blocks.

Leveraging SSDs with built-in transparent compression, ZipCache SSD cache tier simply writes the B+ tree leaf pages in their original, uncompressed form into the underlying SSDs. Although being greatly assisted by such a new breed of SSDs, ZipCache SSD cache tier still needs to tackle two nontrivial issues: (1) How to reduce its host DRAM consumption without affecting the cache hit cost? (2) How to reduce the SSD write amplification in the presence of significant size mismatch between B+ tree pages and cache objects? In the following, we present three design techniques to address these two challenging issues.

Intra-page object hashing. In order to accelerate the SSD cache hits, we keep all the non-leaf pages of the SSD-tier B+ tree in host DRAM. It allows us to access SSD only once when serving an SSD cache read request. However, this approach incurs non-trivial spatial overhead for storing these non-leaf pages in DRAM, which reduces the DRAM capacity available for DRAM cache tier. To mitigate this issue, the only option is to increase the size of SSD cache B+ tree leaf pages. Conventional implementation of a B+ tree always reads and writes one page as a whole on storage devices. As a result, a larger leaf page size would proportionally increase the SSD read/write amplification, leading to a higher SSD cache hit cost and shorter SSD lifetime.

We address the above-said challenge by decoupling the B+ tree leaf page size from SSD read/write unit. As illustrated in Fig. 6, we construct each SSD cache B+ tree leaf page in a hash-based manner. Since the LBA I/O size is by default 4KB, we set SSD cache's B+ tree leaf page size as a multiple of 4KB (i.e., 4m KB, where m is a positive integer). Each leaf page $\mathbf{Q}$ is partitioned into m 4KB sub-pages $[\mathbf{q}_1, \mathbf{q}_2, \cdots, \mathbf{q}_m]$. Let $\mathcal{K}$ denote the cache object key space and define a hash function $g : \mathcal{K} \rightarrow [1, m]$. For cache objects that fall into one B+ super-leaf page, we could use the hash function g to calculate their destined 4KB sub-pages. As a result, regardless of the B+ tree page size (e.g., 16KB or 64KB), we only need to fetch/write one 4KB from/to SSD to serve a read/write request. Such leaf pages are referred to as *super-leaf* pages.

Although such hash-based page construction likely leaves empty space in 4KB sub-page, we could obviate the waste of physical flash memory storage space by filling up the empty space in each 4KB sub-page with all zeros. Intra-SSD compression could seamlessly compress away the all-zero segments. It is worth noting that although both DRAM-tier and SSD-tier B+ trees adopt a similar hash-based leaf-page construction, they serve for completely different purposes: The former is for enabling decompression early termination to accelerate cache hits, while the latter is for mitigating the SSD cache read/write amplification problem by decoupling the read/write units from the B+ tree leaf pages.

Page-based DRAM-to-SSD eviction. Write amplification has a strong negative impact on SSDs in terms of both performance and lifetime. This and next techniques aim to reduce the write amplification caused by DRAM-to-SSD cache objects eviction. Because of intra-SSD transparent compression, we can calculate the overall write amplification as follows:

Let V_{obj} denote the total data volume of all the cache objects being evicted from DRAM to SSD, V_{host} denote the total amount of data written by host to SSD through the I/O interface (e.g., NVMe), and V_{NAND} denote the total amount of data written into the NAND flash memory. We define (i) *host-side write amplification* $WA_{host} = V_{host}/V_{obj} \geq 1$ to represent the write amplification induced by host-side software; (ii) *intra-SSD write reduction* $WR_{NAND} = V_{host}/V_{NAND} \geq 1$ to quantify the effect of intra-SSD compression. Hence, we can express the overall write amplification $WA = WA_{host}/WR_{NAND}$. To reduce the damage on NAND flash memory, we must reduce the host-side write amplification WA_{host} and/or increase the intra-SSD write reduction WR_{NAND}.

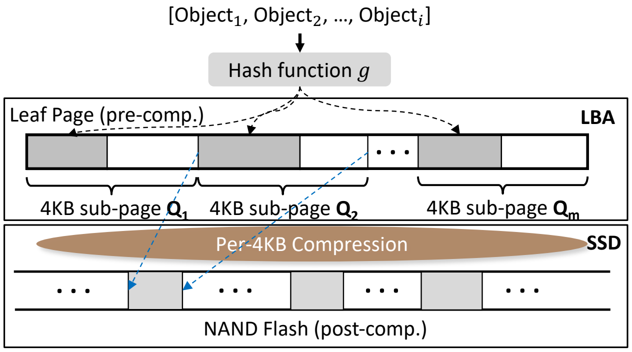


Figure 6: Illustration of hash-based mapping between cache objects and 4KB sub-pages in a leaf page of BT_{SSD}, which decouples the the SSD read/write amplification from B+ tree leaf page size.

Because both DRAM and SSD cache tiers use B+ tree index, we propose to apply page-oriented DRAM-to-SSD cache objects eviction to reduce the host-side write amplification WA_{host}. We choose the classical second-chance eviction policy for the purpose of implementation simplicity. Since B+ trees sort all the DRAM/SSD-resident cache objects based on their keys, the key range of a DRAM cache B+ tree leaf page may overlap with only one or few SSD cache B+ tree leaf pages. Unlike evicting small key-value items from random locations in cache, by evicting cold cache data in the unit of leaf pages, it only incurs the read-modify-write operations over a small number of SSD LBAs, leading to a small host-side write amplification WA_{host}. Moreover, thanks to the abundant spatial locality in real-world workloads, such page-based eviction is highly efficient, compared to object-based eviction as in CacheLib [19]. As demonstrated in prior work [47] and our experiments in Section 4, object-based eviction suffers from very high SSD write amplification.

Sub-page under-filling. The objective of this technique is to reduce the SSD write reduction WR_{NAND} by increasing the content compressibility of each 4KB SSD LBA block. As discussed above, within one SSD cache B+ tree super-leaf page, each cache object is hashed into one of multiple 4KB sub-pages, where each sub-page associates with one 4KB SSD LBA. Let $\beta_{fill} \leq 1$ denote the 4KB sub-page fill-factor (i.e., the percentage of 4KB space that is occupied by cache objects), and the rest $1 - \beta_{fill}$ portion of the sub-page is filled with all zeros. Evidently, the compression ratio of each 4KB sub-page is inversely proportional to its fill-factor β_{fill}. The lower the sub-page fill-factor is, the more the sub-page can be compressed inside the SSD. We set a threshold T on the permissible sub-page fill-factor. As we evict pages from DRAM to SSD, once the fill-factor of any sub-page exceeds T, the entire page is split to ensure none of sub-pages have a fill-factor larger than the specified threshold. By setting T well below 1 (e.g., 0.75), we can improve the sub-page compression ratio and hence increase the intra-SSD write reduction. Meanwhile, T cannot be too small due to the B+ tree page split overhead.

3.4 Major Operations

ZipCache supports GET, SCAN, PUT, and DELETE requests. The operation flows are summarized as follows:

To serve a GET request, we search all the three B+ trees in the order of $BT_{DRAM} \rightarrow BT_{LO} \rightarrow BT_{SSD}$. Since both BT_{DRAM} and BT_{LO} entirely reside in host DRAM, we search the SSD cache tier B+ tree BT_{SSD} in the last to ensure SSD is accessed no more than once when serving a GET request. If a GET request hits the SSD cache tier, the obtained tiny/medium-size cache object will be inserted into the DRAM cache tier. To serve a SCAN request, we must carry out range scans over all the three B+ trees and accordingly merge the results together as the output.

To serve a PUT request, if the cache object is a tiny/medium-size object, we insert it into the DRAM cache tier, and meanwhile search the large-size object index B+ tree BT_{LO} for possible cache object deletion, ensuring that any existing large-size object with the same key is removed. If the cache object is a large-size object, we write it to SSD in the 4KB-aligned manner and insert its pointer into BT_{LO}, and meanwhile insert a *tombstone* object with the same key into the DRAM cache tier to perform possible cache object deletion. We note that a *tombstone* inserted into the DRAM cache tier will not disappear until it reaches the SSD cache tier. To serve a DELETE request, we insert one *tombstone* object into the DRAM cache tier, and search BT_{LO} for possible cache object deletion.

Besides normal B+ tree management operations such as page split, ZipCache carries out two additional major background operations: (1) Leaf page re-compression in DRAM cache B+ tree: As DRAM cache B+ tree uses per-page write buffering to amortize the leaf page re-compression cost, once the size of one per-page write buffer reaches the pre-specified threshold, ZipCache performs background decompression-modify-compression over the leaf page to merge the buffered objects into the compressed leaf page. (2) DRAM-to-SSD page eviction: ZipCache keeps track of the hotness of each DRAM cache B+ tree leaf page. When the DRAM cache tier runs out of memory space, ZipCache evicts cold DRAM-resident leaf pages into the SSD cache tier. All the in-memory medium-size objects associated with to-be-evicted pages are first decompressed and then moved to the SSD cache tier together with other tiny-size objects.

4 Evaluation

We have implemented the ZipCache prototype in C++ and carried out experiments on a server with two Intel Xeon Gold 6134 CPUs, 384GB DRAM, and one 3.84TB ScaleFlux CSD 3000 drive with built-in transparent compression. The system OS is Ubuntu Linux release 22.04. Being fully compliant with the NVMe protocol, the CSD 3000 drive realizes hardware-based zlib (de)compression on each 4KB LBA data block along the I/O path. It can achieve a compression ratio similar to that of the software zlib library at the level 6, and its (de)compression latency is sub-5μs, at least one order of magnitude shorter than TLC/QLC NAND flash memory read/write latency. Meta's Cachebench [19] is used to generate realistic cache object access workloads with the configurable access locality. By configuring the percentage of cache objects that collectively serve 80% cache access requests in Cachebench, we carried out experiments based on four different categories of workload locality as listed in Table 2, where 80%→20% means that 80% cache access requests hit 20% of all the cache objects. In the case of *zero locality*, we configure Cachebench to randomly generate requests over all the cache objects, representing the worst-case scenario.

Table 2: Four categories of workload locality.

Strong	Moderate	Weak	Zero
80%→8%	80%→20%	80%→64%	*Random*

To study the impact of cache object content compressibility, we modified Cachebench to generate the cache object content as follows. Given a parameter $\eta \in [0\%, 100\%)$, we fill $1-\eta$ of each cache object with incompressible random content and set the remaining η portion as all zeros. Hence, the cache object content compressibility improves as η increases. The same method has been used by the popular I/O test tool FIO (flexible I/O tester) [5] to generate I/O data with configurable compressibility.

4.1 Overall Cache Performance

We first evaluate and compare the speed performance of CacheLib [8], Kangaroo [47] (a variant of CacheLib for reducing SSD write amplification), xcache [3], and ZipCache. To cover a wide range of spectrum, we considered the scenarios when the total active working set size is either larger or smaller than the DRAM/SSD cache capacity. For both scenarios, we set the key size as 16B and cache object size as 64B. The cache object content is generated with the compressibility parameter $\eta = 50\%$. The workload consists of 16 user threads issuing GET and PUT requests at the ratio of 1:1. Since adaptive compression bypassing is effective only under workloads with very strong locality, we disable this feature here an will study its effect later in Section 4.4. The parameters of CacheLib and xcache are set as their default values, and we turned on the LZ4 compression of xcache's SSD cache tier. For ZipCache, the leaf page size of DRAM tier B+ tree and SSD tier B+ tree is set to 4KB and 64KB, respectively, and the DRAM tier per-page write buffer size limit is set to 256B.

Fig. 7 shows the throughput and overall cache hit ratio when the active working set is much larger than the DRAM/SSD cache capacity. We fixed the total active working set size C_{WS} as 6TB. Let C_{DRAM} and C_{SSD} denote the DRAM and SSD cache tier capacity, we considered four different settings of $\{C_{DRAM}, C_{SSD}\}$: {64GB, 0.5TB}, {64GB, 1TB}, {128GB, 0.5TB}, and {128GB, 1TB}. In case of the SSD cache tier miss due to the larger-than-cache active working set, we configured the backend data access latency as 1ms. Since xcache/ZipCache both apply compression over the SSD cache tier, they have similar overall cache hit ratio that is higher than CacheLib. As a result, xcache and ZipCache have higher throughput than CacheLib and Kangaroo, as shown in Fig. 7. Meanwhile, since ZipCache applies compression over its DRAM cache tier, it has a higher DRAM cache tier hit ratio (hence higher throughput) than xcache. For instance, under moderate workload locality with $\{C_{DRAM}, C_{SSD}\}$ of {128GB, 1TB}, the overall cache hit ratio of ZipCache is 69.7%, 2.6 percentage points (p.p.), 29.2 p.p. and 30.9 p.p. higher than xcache, CacheLib, and Kangaroo. The throughput of ZipCache is about 6.1%, 75.0% and 76.1% than that of xcache,

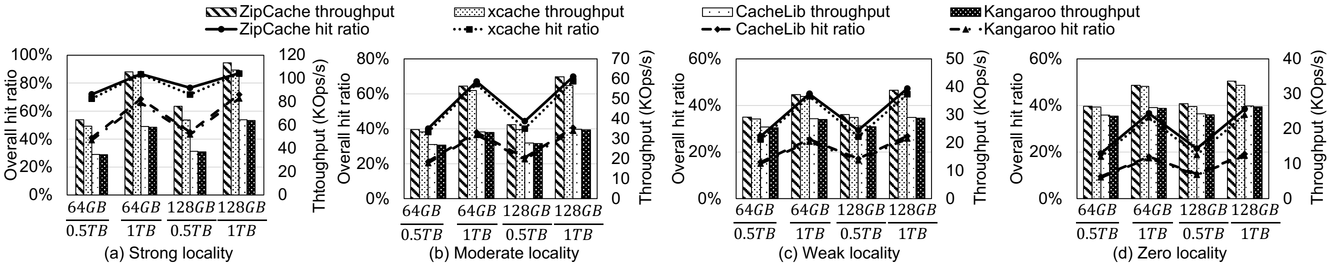


Figure 7: Comparison of throughput and overall DRAM/SSD cache hit ratio when the active working set is much larger than the DRAM/SSD cache capacity. We fixed the total active working set size C_{WS} as 6TB and, under each category of workload locality, considered four different settings of {C_{DRAM}, C_{SSD}}: {64GB, 0.5TB}, {64GB, 1TB}, {128GB, 0.5TB}, and {128GB, 1TB}.

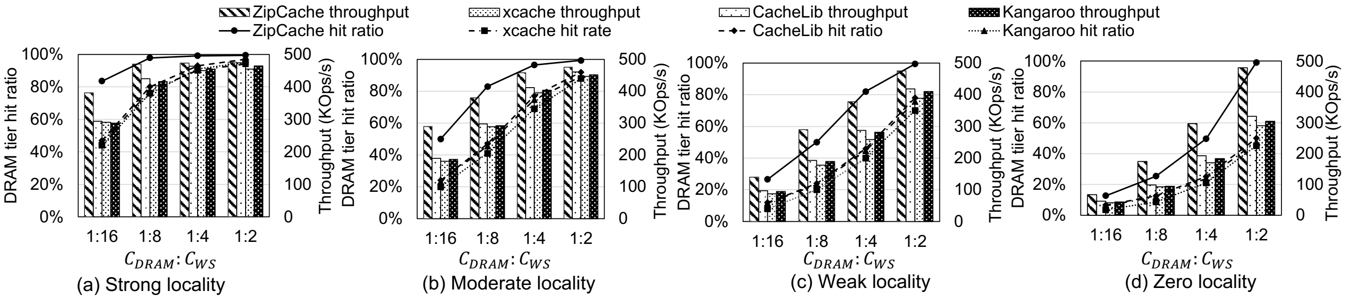


Figure 8: Comparison of throughput and DRAM cache hit ratio when the active working set fits into the DRAM/SSD cache (hence the SSD cache tier hit ratio is 100%). We fixed the DRAM cache tier capacity C_{DRAM} as 64GB and considered C_{DRAM} : C_{WS} ratio of 1:16, 1:8, 1:4, and 1:2.

CacheLib and Kangaroo, respectively. As shown in Fig. 7, as the workload locality weakens, the performance difference among the three caches becomes smaller since the backend access latency becomes more dominant.

Fig. 8 shows the throughput and DRAM cache tier hit ratio when the active working set fits completely in the DRAM/SSD cache (hence the SSD cache tier hit ratio is 100%). We fixed the DRAM cache tier capacity C_{DRAM} as 64GB and considered the C_{DRAM} : C_{WS} ratio of 1:16, 1:8, 1:4, and 1:2. Because of the build-in block compression over its DRAM cache tier, ZipCache consistently achieves higher DRAM tier hit rate and higher throughput than CacheLib, Kangaroo and xcache, with throughput improvements of up to 72.4%. Compared with xcache, CacheLib achieves slightly higher DRAM cache hit ratio and hence throughput than xcache. Under high workload locality (e.g., strong/moderate locality), the advantage of ZipCahe over CacheLib/Kangaroo/xcache gradually diminish as the DRAM capacity increases. For example, under workloads with *strong locality*, when a small cache capacity (C_{DRAM} : C_{WS} of 1:16), ZipCache achieves about 36.3 p.p. higher DRAM tier hit rate and 30.4% higher throughput than CacheLib/Kangaroo/xcache; when C_{DRAM} : C_{WS} increases to 1:2, their DRAM hit rate and throughput become almost the same. This is because, under high workload locality, increasing the DRAM capacity alone (without data compression) can be sufficient to quickly raise the DRAM tier hit rate over 90%. In comparison, under relatively low workload locality (e.g., weak/zero locality), increasing the DRAM capacity alone is much less effective on improving the DRAM tier hit ratio. As a result, the value of data compression becomes more evident. For example, with *zero locality*, ZipCache achieves a DRAM hit rate that is 6.4 to 49.2 p.p. higher than CacheLib, when C_{DRAM} : C_{WS} increases from 1:16 to 1:2.

4.2 DRAM Cache Tier Compression

Fig. 9 further shows the GET latency when active working set fits in DRAM/SSD cache, where we measured the 50-percentile (p50) latency, 90-percentile (p90 latency), and 99-percentile (p99) latency. Compared to CacheLib, Kangaroo, and xcache, ZipCache consistently achieves shorter latency due to its higher DRAM cache tier hit ratio, with latency reductions of up to 42.4%. Moreover, the latency reduces as the DRAM cache tier capacity increases, e.g., the p90 latency of ZipCache reduces from 28.9μs to 5.2μs when C_{DRAM} : C_{WS} increases from 1:16 to 1:2. All the three have similar p99 latency since it is largely determined by the SSD tier read latency.

In addition to workloads with GET:PUT ratio of 1:1, we further compared the performance under PUT-only and GET-only Cachebench workloads with moderate locality. As shown in Fig. 10, under C_{DRAM} : C_{WS} of 1:16 and PUT-only workload, ZipCache achieves 78.1% and 145.7% higher throughput than CacheLib/Kangaroo

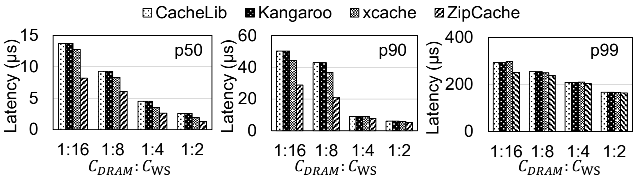


Figure 9: GET latency under workloads with moderate locality and different $C_{DRAM} : C_{WS}$.

and xcache, because its DRAM tier compression could help reducing the SSD tier write amplification due to DRAM-to-SSD eviction. Under $C_{DRAM} : C_{WS}$ of 1:16 and GET-only workload, ZipCache achieves 56.3% and 71.6% higher throughput than CacheLib/Kangaroo and xcache, because its DRAM tier compression helps to increase the DRAM tier hit ratio. The performance difference gradually shrinks as the DRAM cache tier capacity increases.

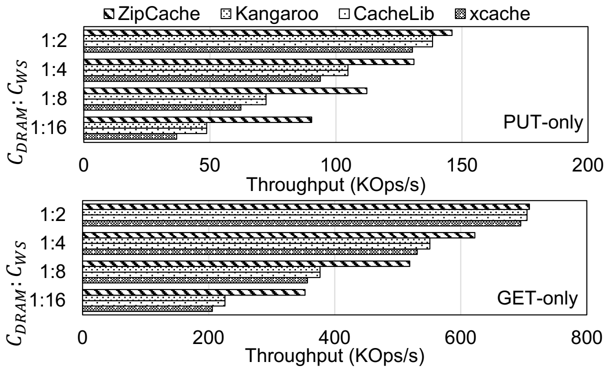


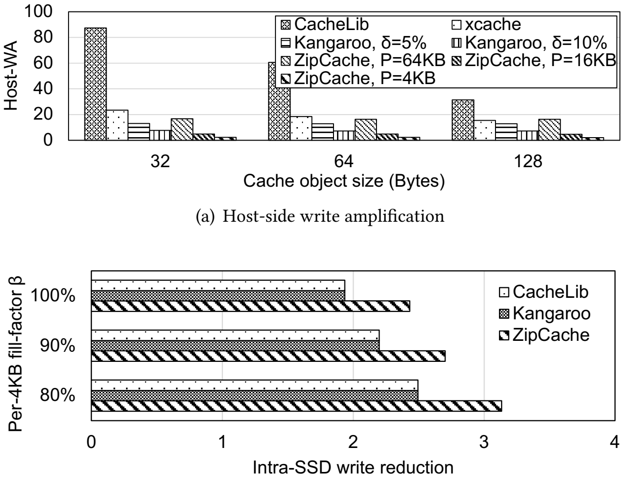
Figure 10: Average throughput of PUT-only and GET-only workload with moderate locality and different $C_{DRAM} : C_{WS}$.

The above results well demonstrate the effectiveness of incorporating in-memory data compression to improve the cache speed performance. This essentially attributes to the substantially increased DRAM cache tier hit ratio enabled by in-memory compression and the significant data access latency gap between DRAM and SSD.

4.3 SSD Cache Tier Write Amplification

We further compared the SSD cache tier write amplification among ZipCache, CacheLib [8], Kangaroo [47], and xcache [3]. As discussed above in Section 3.3, once we deploy a hybrid-DRAM/SSD cache on SSDs with built-in compression, we can express the overall SSD write amplification as $WA = WA_{host}/WR_{NAND}$, where the *host-side write amplification* $WA_{host} \geq 1$ represents the write amplification induced by host-side cache software and *intra-SSD write reduction* $WR_{NAND} \geq 1$ quantifies the effect of intra-SSD compression (i.e., compression ratio achieved by SSD).

Fig. 11(a) shows the host-side write amplification WA_{host} under the Cachebench workload with *moderate locality*. The key size is 16B and cache object size is 32B, 64B, and 128B. By directly hashing each cache object into one SSD 4KB LBA, CacheLib experiences very high WA_{host} that is inversely proportional to the cache object size. For example, its WA_{host} increases from 31.5 to 60.7 when the cache object size reduces from 128B to 64B. Regarding xcache, its DRAM cache uses hash index and SSD cache uses LSM-tree index. Hence, xcache has lower SSD write amplification than CacheLib. By complementing CacheLib with a WAL to accumulate multiple cache objects hashed to the same SSD 4KB LBA, Kangaroo [47] reduces WA_{host} at the cost of SSD storage capacity. Let $\delta < 1$ denote the ratio of WAL size and SSD cache size, we considered two values of δ: 5% and 10%. Fig. 11(a) shows the effectiveness of WAL-assisted write amplification and the trade-off between the WAL-induced storage overhead δ and host-side write amplification WA_{host}. For ZipCache, we considered three leaf page size of SSD tier B+ tree BT_{SSD}, including 4KB, 16KB, and 64KB. As discussed above, to minimize SSD tier cache hit time, BT_{SSD} keeps all its non-leaf pages in DRAM and leaves leaf pages on SSD. We define the BT_{SSD} memory overhead (denoted as ζ) as the ratio between the total size of its in-memory non-leaf pages and total size of its on-SSD leaf pages. The BT_{SSD} leaf page size affects the trade-off between host-side write amplification WA_{host} and BT_{SSD} memory overhead ζ, which can be observed from Fig. 11(a) and Table 3. The results show that, dependent upon their different configurations, Kangaroo and ZipCache have comparable host-side write amplification, which is slightly better than xcache and significantly better than that of CacheLib, especially under small cache object size.



(a) Host-side write amplification

(b) Intra-SSD write reduction

Figure 11: (a) Host-side write amplification WA_{host} comparison among CacheLib, Kangaroo and ZipCache under different cache object size, and (b) intra-SSD write reduction under different per-4KB fill-factor (xcache is not included since its intra-SSD write reduction remains as 1).

Table 3: BT_{SSD} memory overhead ζ.

Leaf page size	64KB	16KB	4KB
Memory overhead ζ	0.6%	2.1%	7.5%

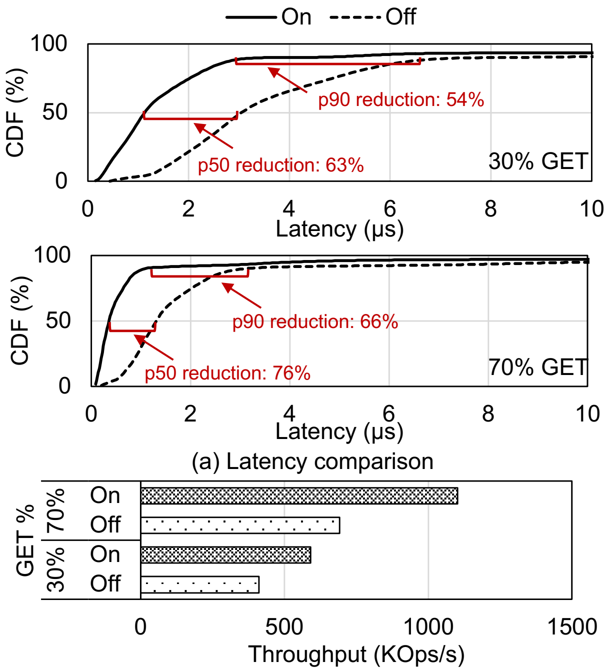
Fig. 11(b) shows the measured intra-SSD write reduction WR_{NAND} when running CacheLib, Kangaroo, and ZipCache on SSD with built-in transparent compression. Since xcache's LSM-tree-based SSD cache tier applies block compression, it does not benefit from intra-SSD transparent compression and hence always has the intra-SSD write reduction of 1. We generate the content of each cache object with the compressibility parameter $\eta = 50\%$. As discussed above, the per-4KB under-filling technique can be equally applied to CacheLib and Kangaroo. Hence, we measured the intra-SSD write reduction of all three caches under different per-4KB fill-factor β_{fill} including 100%, 90%, and 80%. Since Kangaroo and CacheLib employ the same hash-based SSD tier cache structure, they have the same intra-SSD write reduction WR_{NAND}. As discussed above in Section 2.2, by sorting all the cache objects based on their keys, B+ tree leaf pages have a higher compressibility due to the stronger content correlation across adjacent keys. Hence, ZipCache can achieve larger intra-SSD write reduction. For example, under the fill-factor of 80%, CacheLib/Kangaroo have the WR_{NAND} of 2.49 while ZipCache has WR_{NAND} of 3.13.

By combining the above results of host-side write amplification and intra-SSD write reduction, we can observe that, in addition to its higher DRAM tier hit ratio and hence higher speed performance, ZipCache achieves significantly lower SSD write amplification compared to CacheLib and xcache, with reductions of up to 26.2×. For example, with the object size of 64B and per-4KB fill-factor of 90%, the overall SSD write amplification of ZipCache (leaf page size of 16KB) is 1.8, which is only 3.7% and 9.8% of CacheLib and xcache, respectively. Even compared with Kangaroo, the CacheLib variant that is solely optimized for reducing the SSD write amplification, ZipCache has comparable or lower SSD write amplification. For example, with the object size of 64B and per-4KB fill-factor of 90%, the overall SSD write amplification of ZipCache (leaf page size of 16KB) is 53.5% of Kangaroo (with storage space overhead δ of 10%).

4.4 Adaptive Compression Bypassing

For workloads with strong localities, we could noticeably improve the cache performance by adaptively bypassing the compression over very hot leaf pages of the DRAM-tier cache's B+ tree BT_{DRAM}. In this section we will study the effect of such adaptive compression bypassing. Fig. 12 shows the CDF (cumulative distribution function) of the measured GET latency and average throughput under the Cachebench workload with *strong locality*. We configure a write-intensive and a read-intensive workload with the GET:PUT ratio set to 30%:70% and 70%:30%, respectively. The results show that the proposed *adaptive compression bypassing* can significantly improve the throughput and reduce the perceived latency. For example, by turning on *adaptive compression bypassing*, we could reduce the p50 GET latency by 63% for the write-intensive workload and 76% for the read-intensive workload, correspondingly improving the average throughput by 44% and 59%, respectively. The results clearly show the benefits of obviating CPU-intensive (de)compression operations over hot B+ tree leaf pages in the DRAM cache tier.

In above experiments, the workload *hot region* remains stationary and hence has been fully captured by ZipCache. We further carry out experiments to study the responsiveness of compression bypassing to runtime workload variations. Using the same Cachebench workload with *strong locality*, we arbitrarily shift the position of the *hot region* to cover a non-overlapping set of in-memory cache objects. Fig. 13 shows the measured average GET latency before and after this sudden hot region shift. In the figure, we can see that the average GET latency rapidly increases from 1.6μs to 3.4μs due to the working-set change. As compression bypassing responds to the workload change by re-compressing the pages of the previous working set and decompressing the pages of the new hot region, the average latency gradually returns back to 1.6μs after serving about 500K operations. Given the average throughput of over 400K operations per second under strong locality (as shown above in Fig. 8), we can estimate the transition period of no more than few seconds.



(a) Latency comparison

(b) Throughput comparison

Figure 12: (a) GET latency and (b) average throughput comparison under Cachebench workload with *strong locality*, where GET:PUT ratio is 30%:70% or 70%:30%.

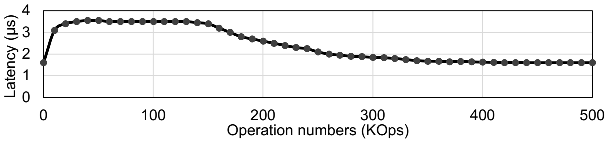
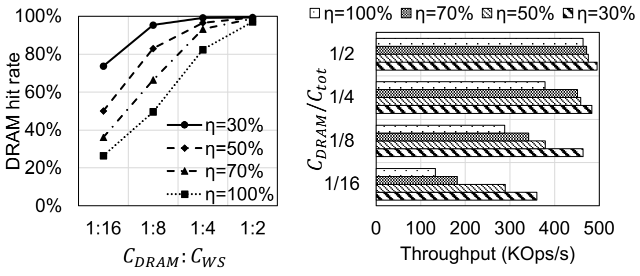


Figure 13: Change of the average GET latency after the sudden *hot region* position shift.

4.5 Sensitivity Study

Compressibility. ZipCache has several configurable parameters. In the above experiments, we fixed the cache object compressibility parameter η as 50%. Fig. 14 shows the measured DRAM cache tier hit ratio and throughput under different settings of η. We use the Cachebench workload with *moderate locality* and set the key size to 16B and cache object size to 64B. The results show the effect of data content's compressibility on the cache performance. With the DRAM cache capacity vs. active working set size ratio $C_{DRAM} : C_{WS}$ of 1:16, as the compressibility increases from incompressible (η = 100%) to highly compressible (η = 30%), the DRAM tier cache hit ratio increases from 26.4% to 73.6%. The results clearly show the impact of data compressibility on the performance of ZipCache.



(a) DRAM tier hit ratio (b) Throughput

Figure 14: (a) DRAM tier cache hit ratio and (b) overall throughput under different compressibility parameter η.

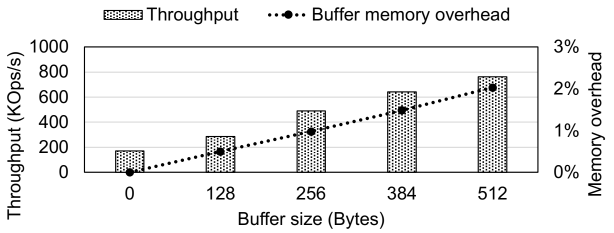


Figure 15: Throughput and memory usage overhead under different per-page write buffer size, where PUT-only Cachebench workload with *strong locality* is used.

Write Buffer Size. The per-page write buffer size could also noticeably impact the DRAM tier cache performance under write-intensive workloads. As we reduce the per-page write buffer size to save DRAM space usage, the DRAM-tier cache's B+ tree would experience more frequent page re-compression, leading to degraded performance. Fig. 15 shows the ZipCache throughput and DRAM usage overhead under different per-page write buffer sizes using the PUT-only Cachebench workload with *strong locality*. The overhead of DRAM usage is defined as the ratio between the aggregated per-page write buffer size and total DRAM tier cache capacity. The results clearly show the trade-off between the cache performance and the overhead of DRAM usage. As we increase the per-page write buffer size from 0 to 256B, the cache performance improves by 2.9× at the cost of 1% DRAM usage overhead.

Cache Object Size. Given the same total cache capacity, as cache object size decreases, the number of cache objects increases, leading to more cache index implementation complexity and more significant compression-induced overhead. Hence, to most heavily stress the cache, the experiments above focus on workloads with only tiny-size cache objects. To show the effect of cache object size, we have performed experiments with Cachebench workloads that contain tiny-size (64B), medium-size (256B), and large-size (2KB) cache objects. We ensure that the three categories of cache objects consume the same amount of cache capacity. Fig. 16 shows the GET latency for the three different sizes of cache objects. Requests over medium-size objects experienced longer latency than that over tiny-sized objects. For instance, under $C_{DRAM} : C_{WS}$ of 1:16, the p50 latency for medium-size objects is 26% more than that for tiny objects. This increased latency is due to the extra step of decompressing these medium-size objects. Large-size objects, stored on SSDs, have significantly higher GET latency compared to both tiny and medium-size objects.

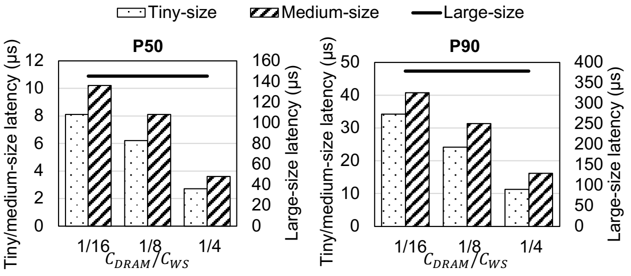


Figure 16: GET latency for various sizes objects under the Cachebench workload with *modest locality*.

5 Related Work

Key-Value Stores: The design and implementation of in-memory and SSD-based key-value stores have been extensively studied by the research community. Most in-memory KV stores chose to employ hash-based index (e.g., Memcached [1], Redis [6], MemC3 [32], MICA [44], Mega-KV [61], RAMCloud [50], FASTER [21], HotRing [23]). Only few implementations, such as MassTree [45], use tree-based index. SSD-based key-value stores typically employ tree-based index and support block data compression. Because of its low write amplification, log-structured merge (LSM) tree [51] index data structure has received most recent attentions in research community for building SSD-based key-value stores [7, 17, 22, 27, 36, 54].

Prior work also studied the design of hybrid key-value stores over different memory technologies (e.g., DRAM and SSD), aiming to strike a better balance between the speed and cost. CacheLib [19] and xcache [3] are two representative hybrid-DRAM/SSD key-value stores, where CacheLib uses hash-based index for both the DRAM and SSD tiers and xcache uses hash-based index for DRAM tier and LSM-tree index for SSD tier. Motivated by intensive research on NVM (non-volatile memory) technologies over the past decade, recent work has developed NVM-enhanced hybrid key-value stores [28, 30, 59, 60]. Though there is prior research on flash cache compression [35, 42, 43, 56], little work studied the potential of improving cache performance via in memory block data compression.

Memory Compression: The computer architecture research community has widely studied the implementation of hardware-based main memory compression [25, 31, 52, 62]). Aiming at better serving general-purpose computing systems, hardware-based main memory compression focuses on fine-grained per-cacheline (e.g., 64B) compression. The Linux kernel feature Zswap [13] compresses to-be-swapped 4KB pages and keeps them in DRAM, which has been used by Google [39] and Meta [57] to increase effective DRAM capacity. Contemporary data analytics systems like SAP HANA [33], Oracle [40], and Snowflake [26] apply block compression to their in-memory column-stores to reduce their memory consumption. In-memory time series databases [14, 53] also widely use compression to exploit the inherently high compressibility of time series data.

Computational SSD: Computational storage has attracted significant recent interest [18, 20, 38, 55], and commercial products are emerging on the commercial market (e.g., Samsung's SmartSSD [10] and ScaleFlux's CSD [11]). Recent research [24, 37, 63] has studied how database management systems could take advantage of computational SSDs with built-in transparent compression.

6 Conclusion

This paper presents a hybrid cache design called ZipCache, which integrates block data compression to improve the cache performance. To maximize the block compression ratio and hence cache hit ratio, in contrast to most existing in-memory cache design, ZipCache employs the classic B+ tree indexes to manage both the DRAM and SSD cache tiers. We developed several design techniques to reduce compression-induced DRAM tier cache hit cost overhead. Built upon emerging SSDs with transparent compression, ZipCache SSD tier cache incorporates several design techniques to reduce the its B+ tree index memory consumption and reduce the SSD write amplification, especially for workloads dominated by tiny-size cache objects. Extensive experiments demonstrated its effectiveness and studied the involved design trade-offs.

Acknowledgments

This work was supported in part by the National Science Foundation (NSF) under grants CNS-2006617, CCF-1910958, CCF-2210754, CCF-2210755, CCF-2312508, and CCF-2312509.

References

[1] 2018. *Memcached.* https://memcached.org/

[2] 2018. Silesia Corpus. https://github.com/MiloszKrajewski/SilesiaCorpus.

[3] 2022. xcache. https://github.com/XimalayaCloud/xcache.

[4] 2023. *Bitstamp Exchange Data.* https://www.cryptodatadownload.com/data/bitstamp/.

[5] 2023. *Flexible I/O Tester.* https://github.com/axboe/fio.

[6] 2023. *Redis.* https://redis.io

[7] 2023. *RocksDB.* https://rocksdb.org

[8] 2024. *CacheLib.* https://github.com/facebook/CacheLib

[9] 2024. Pika. https://github.com/OpenAtomFoundation/pika.

[10] 2024. *Samsung SmartSSD.* https://semiconductor.samsung.com/ssd/smart-ssd/

[11] 2024. *ScaleFlux Computational Storage.* http://scaleflux.com

[12] 2024. *WiredTiger.* https://github.com/wiredtiger/

[13] 2024. *Zswp.* https://wiki.archlinux.org/title/Zswap.

[14] Colin Adams, Luis Alonso, Benjamin Atkin, John Banning, Sumeer Bhola, Rick Buskens, Ming Chen, Xi Chen, Yoo Chung, Qin Jia, et al. 2020. Monarch: Google's planet-scale in-memory time series database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3181–3194.

[15] Bryan Ao. 2023. *NAND Flash Prices Expected to Stabilize and Rebound in Q4, Projected to Remain Steady or Increase 0-5%, Says TrendForce.* https://www.trendforce.com/presscenter/news/20230911-11839.html.

[16] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-scale Key-Value Store. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS).* 53–64.

[17] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *Proceedings of USENIX Annual Technical Conference (ATC).* 363–375.

[18] Antonio Barbalace and Jaeyoung Do. 2021. Computational Storage: Where Are We Today?. In *Proc. of Annual Conference on Innovative Data Systems Research (CIDR).*

[19] Ben Berg, Daniel Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. 2020. The CacheLib caching engine: Design and experiences at scale. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[20] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, Feng Zhu, and Tong Zhang. 2020. POLARDB meets computational storage: Efficiently support analytical workloads in cloud-native relational database. In *USENIX Conference on File and Storage Technologies (FAST).* 29–41.

[21] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. Faster: A concurrent key-value store with in-place updates. In *Proceedings of the International Conference on Management of Data (SIGMOD).* 275–290. https://doi.org/10.1145/3183713.3196898

[22] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A fast, cost-effective LSM-tree based KV store on hybrid storage. In *USENIX Conference on File and Storage Technologies (FAST).* 17–32.

[23] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. 2020. HotRing: A Hotspot-Aware In-Memory Key-Value Store. In *USENIX Conference on File and Storage Technologies (FAST).* 239–252.

[24] Xubin Chen, Ning Zheng, Shukun Xu, Yifan Qiao, Yang Liu, Jiangpeng Li, and Tong Zhang. 2021. KallaxDB: A Table-less Hash-based Key-Value Store on Storage Hardware with Built-in Transparent Compression. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN).* 1–10.

[25] Esha Choukse, Mattan Erez, and Alaa R Alameldeen. 2018. Compresso: Pragmatic main memory compression. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* IEEE, 546–558.

[26] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the International Conference on Management of Data (SIGMOD).* 215–226.

[27] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the International Conference on Management of Data (SIGMOD).* ACM, 505–520.

[28] Chen Ding, Ting Yao, Hong Jiang, Qiu Cui, Liu Tang, Yiwen Zhang, Jiguang Wan, and Zhihu Tan. 2022. TriangleKV: Reducing write stalls and write amplification in LSM-tree based KV stores with triangle container in NVM. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 4339–4352.

[29] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Transactions on Storage (TOS)* 17, 4 (2021), 1–32.

[30] Zhuohui Duan, Jiabo Yao, Haikun Liu, Xiaofei Liao, Hai Jin, and Yu Zhang. 2023. Revisiting Log-Structured Merging for KV Stores in Hybrid Memory Systems. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).* 674–687.

[31] Magnus Ekman and Per Stenstrom. 2005. A robust main-memory compression scheme. In *32nd International Symposium on Computer Architecture (ISCA).* IEEE, 74–85.

[32] Bin Fan, David G Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI).* 371–384.

[33] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2012. SAP HANA database: data management for modern business applications. *ACM Sigmod Record* 40, 4 (2012), 45–51.

[34] Goetz Graefe et al. 2011. Modern B-tree techniques. *Foundations and Trends® in Databases* 3, 4 (2011), 203–402.

[35] Jingpeng Hao, Xubin Chen, Yifan Qiao, Yuyang Zhang, and Tong Zhang. [n. d.]. Implementing Flash-Cached Storage Systems Using Computational Storage Drive with Built-in Transparent Compression. In *2021 IEEE International Conference on*

Networking, Architecture and Storage (NAS). 1–8.

[36] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An optimized storage engine for large-scale E-commerce transaction processing. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 651–665.

[37] Kecheng Huang, Zhaoyan Shen, Zili Shao, Tong Zhang, and Feng Chen. 2023. Breathing New Life into an Old Tree: Resolving Logging Dilemma of B+-tree on Modern Computational Storage Drives. *Proceedings of the VLDB Endowment* 17, 2 (2023), 134–147.

[38] Dongup Kwon, Dongryeong Kim, Junehyuk Boo, Wonsik Lee, and Jangwoo Kim. 2021. A fast and flexible hardware-based virtualization mechanism for computational storage devices. In *USENIX Annual Technical Conference (ATC)*. 729–743.

[39] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, et al. 2019. Software-defined far memory in warehouse-scale computers. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 317–330.

[40] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, et al. 2015. Oracle database in-memory: A dual format in-memory database. In *IEEE International Conference on Data Engineering (ICDE)*. 1253–1258.

[41] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *IEEE International Conference on Data Engineering (ICDE)*. 302–313.

[42] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. [n. d.]. Nitro: A Capacity-Optimized SSD Cache for Primary Storage. In *2014 USENIX Annual Technical Conference (USENIX ATC)*. 501–512.

[43] Wenji Li, Gregory Jean-Baptise, Juan Riveros, Giri Narasimhan, Tony Zhang, and Ming Zhao. 2016. CacheDedup: In-line Deduplication for Flash Caching. In *14th USENIX Conference on File and Storage Technologies (FAST)*. 301–314.

[44] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 429–444.

[45] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the european conference on Computer Systems*. 183–196.

[46] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS)*. https://doi.org/10.1145/3582016.3582063

[47] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S Berger, Nathan Beckmann, and Gregory R Ganger. 2021. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 243–262.

[48] Timothy Prickett Morgan. 2020. *CXL and Gen-Z Iron Out A Coherent Interconnect Strategy.* https://www.nextplatform.com/2020/04/03/cxl-and-gen-z-iron-out-a-coherent-interconnect-strategy/.

[49] Jan Olšan. 2023. *The days of SSDs getting cheaper are over. Prices are starting to rise.* https://www.hwcooling.net/en/the-days-of-ssds-getting-cheaper-are-over-prices-will-rise/.

[50] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. 2015. The RAMCloud storage system. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 1–55.

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

[52] Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. 2013. Linearly compressed pages: A low-complexity, low-latency main memory compression framework. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*. 172–184.

[53] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1816–1827.

[54] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. 497–514.

[55] Tobias Vinçon, Christian Knödler, Leonardo Solis-Vasquez, Arthur Bernhardt, Sajjad Tamimi, Lukas Weber, Florian Stock, Andreas Koch, and Ilia Petrov. 2022. Near-data processing in database systems on native computational storage under HTAP workloads. *Proceedings of the VLDB Endowment* 15, 10 (2022), 1991–2004.

[56] Qiuping Wang, Jinhong Li, Wen Xia, Erik Kruus, Biplob Debnath, and Patrick PC Lee. [n. d.]. Austere Flash Caching with Deduplication and Compression. In *2020 USENIX Annual Technical Conference (USENIX ATC)*. 713–726.

[57] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, et al. 2022. Tmo: transparent memory offloading in datacenters. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 609–621.

[58] Monica J. White. 2023. *The era of cheap SSDs is about to end.* https://www.digitaltrends.com/computing/samsung-flash-nand-chips-price-increase/.

[59] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *USENIX Annual Technical Conference (ATC)*. 17–31.

[60] Ling Zhan, Kai Lu, Zhilong Cheng, and Jiguang Wan. 2020. RangeKV: An efficient key-value store based on hybrid DRAM-NVM-SSD storage structure. *IEEE Access* 8 (2020), 154518–154529.

[61] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. 2015. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1226–1237.

[62] Jishen Zhao, Sheng Li, Jichuan Chang, John L Byrne, Laura L Ramirez, Kevin Lim, Yuan Xie, and Paolo Faraboschi. 2015. Buri: Scaling big-memory computing with hardware-based memory expansion. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 3 (2015), 1–24.

[63] Ning Zheng, Xubin Chen, Jiangpeng Li, Qi Wu, Yang Liu, Yong Peng, Fei Sun, Hao Zhong, and Tong Zhang. 2020. 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)*.

[64] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on information theory* 23, 3 (1977), 337–343. https://doi.org/10.1109/TIT.1977.1055714

[65] Jacob Ziv and Abraham Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory* 24, 5 (1978), 530–536. https://doi.org/10.1109/TIT.1978.1055934