ADAPT: Dynamic Grouping and Cross-Group Aggregation for GC-Efficient Log-Structured Storage in SSD Arrays

Ruisong Zhou
Huazhong University of Science and Technology
China
ruisongzhou@hust.edu.cn

Peng Wang
Huazhong University of Science and Technology
China
wangpeng@hust.edu.cn

Ke Zhou*
Huazhong University of Science and Technology
China
zhke@hust.edu.cn

Chunhua Li
Huazhong University of Science and Technology
China
li.chunhua@hust.edu.cn

Hui Li
Jinan Inspur Data Technology Co., Ltd.
China
hui.lee@inspur.com

Abstract

Log-structured storage (LSS) has been widely adopted in SSD-based array architectures due to its high I/O performance and flash-friendly design. However, LSS requires optimized data placement to curb the write amplification (WA) resulting from its garbage collection (GC) mechanisms. We reveal that existing data placement strategies suffer from high padding overhead and redundant GC migrations under production workloads due to the granularity mismatch between log-structured storage management and array-level organization. To tackle these inefficiencies, we propose ADAPT, an access-density-aware data placement strategy to decrease padding overhead and minimize WA. ADAPT separates user-written blocks into groups with an adaptive threshold by considering workload-level access density and block-level popularity. Additionally, cross-group dynamic aggregation exploits unused space in cold groups to optimize real-time block padding for hot data. Moreover, we deploy a proactive demotion placement algorithm to coordinate GC processes and eliminate redundant migrations. Experimental evaluations demonstrate that compared with state-of-the-art approaches, ADAPT lowers WA by 21.8%-46.3% and decreases padding traffic by 40%-72.1% under production workloads, while delivering high throughput and maintaining manageable memory overhead under heavy workloads.

CCS Concepts

• **Information systems** → **Storage architectures**; *Disk arrays*; • **Computer systems organization** → *Architectures.*

Keywords

log-structured storage, SSD array, data placement, garbage collection.

ACM Reference Format:
Ruisong Zhou, Peng Wang, Ke Zhou, Chunhua Li, and Hui Li. 2025. ADAPT: Dynamic Grouping and Cross-Group Aggregation for GC-Efficient Log-Structured Storage in SSD Arrays. In *54th International Conference on Parallel Processing (ICPP '25), September 08–11, 2025, San Diego, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3754598.3754616

1 Introduction

With the rapid evolution of information technology and the exponential expansion of data generation scales, solid-state drive (SSD)-based storage arrays have gained widespread adoption in massive data storage and processing domains, due to their superior performance characteristics and continuously optimized cost-efficiency. Compared to traditional hard disk drives (HDDs), SSDs demonstrate exponential improvements in read/write bandwidth and random access performance, along with millisecond-level latency response, high energy efficiency, and enhanced data reliability. To fully exploit the hardware potential of solid-state media for high-performance storage demands, both industry [14, 29, 32] and academia [6, 7] have adopted log-structured storage systems for I/O scheduling optimization. By transforming discrete write operations into sequential append-only patterns, log-structured storage systems leverage the advantages of NAND flash-based devices, achieving lower latency and higher throughput.

Nevertheless, the high cost of garbage collection (GC) remains the primary obstacle in log-structured systems. This process induces additional write operations on user data (known as write amplification, WA), which significantly degrades the performance of log-structured storage. Mitigating WA in log-structured storage has been a well-studied topic in the literature. Prior research has predominantly focused on designing *data placement* strategies [4, 17–19, 25, 26] by grouping segments and placing data blocks in separate groups. Notable implementations include workload-aware algorithms that analyze block temperatures by access frequency [4, 19] and data block lifespan [18, 25], as well as machine learning approaches [17, 26] that use clustering techniques to classify data temperatures and allocate them into optimized groups.

*Corresponding author

ICPP '25, September 08–11, 2025, San Diego, CA, USA

ACM ISBN 979-8-4007-2074-1/25/09
https://doi.org/10.1145/3754598.3754616

However, our analysis reveals a critical limitation of these approaches in SSD array configurations: the inherent block size mismatch between log-structured storage management and array-level organization. While log-structured systems typically operate at 4KB block granularity, modern storage arrays employ substantially larger chunk sizes for efficient error correction (e.g., 16KB or larger) [14]. This architectural disparity leads to frequent zero-padding operations when the storage array receives unaligned writes, as the smaller granularity prevents full utilization of array-level chunks. The resulting overhead not only exacerbates the actual write amplification ratio but also significantly degrades overall array performance and accelerates hardware wear-out.

Although write coalescing techniques offer potential solutions to mitigate this mismatch by aggregating small requests into larger I/O operations, their practical efficacy faces two critical constraints in real-world deployments. The first is the sparsity of access patterns. Real-world workloads often exhibit substantial irregularity and spatial access sparsity, making it challenging for log-structured storage to coalesce sufficient small-grained requests within constrained time windows to fully utilize array-level chunk sizes. Meanwhile, stringent SLA requirements from storage service providers impose strict limitations on the temporal buffering window available for request aggregation. To comply with such latency SLAs (e.g., 100 µs in Ali cloud [29]), storage systems are frequently compelled to flush uncoalesced small requests directly to the storage array, making it difficult to reduce padding through I/O coalescing.

In this paper, we propose ADAPT, an access-density-based data placement strategy that addresses these challenges through three key innovations. First, the *Density-Aware Threshold Adaptation* technique dynamically identifies optimal separating thresholds between cold and hot data by adapting to workload-level access density and block-level popularity, to reduce padding traffic and WA of user-written groups. Second, to address the inefficiency of in-group write coalescing when separating user writes, ADAPT deploys the *Cross-Group Dynamic Aggregation* mechanism which enables temporary persistence of hot data in cold groups to coalesce more requests within fixed chunks. Finally, the *Proactive Demotion Placement* strategy proactively coordinates garbage collection processes to eliminate redundant migrations of cold data through predictive group downgrading.

We implement and evaluate ADAPT on a trace-driven simulator and a log-structured storage prototype. We conduct comprehensive experiments using real-world workloads for diverse production environments[13, 16, 30]. We show that ADAPT achieves minimal WA against current mainstream data placement strategies. For example, ADAPT lowers WA by 21.8%-33.1% under Alibaba Cloud workload [13] with Greedy selection policy. It also decreases padding traffic by 40%-72.1% compared to other temperature-based placement strategies. The prototype evaluation shows ADAPT can deliver high throughput while maintaining manageable memory overhead under heavy workloads.

The rest of the paper is organized as follows. Section 2 and Section 2.3 introduce the background and motivation for our research. Section 3 describes the design details. Section 4 presents the evaluation results. Section 5 discusses related work and the final section concludes the paper.

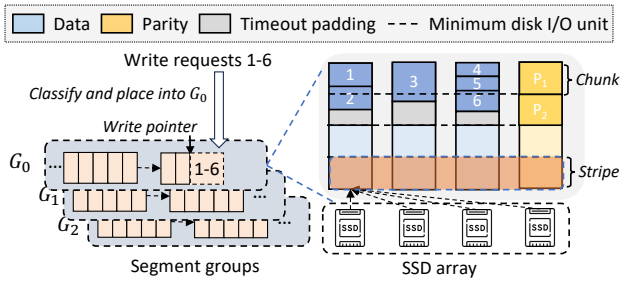


Figure 1: The architecture of the LSS deployed is based on an SSD array. Write requests append data blocks to segments in different groups according to the data placement algorithm. The minimum write unit on an SSD array is a chunk.

2 Background and Motivation

2.1 Log-Structured Storage

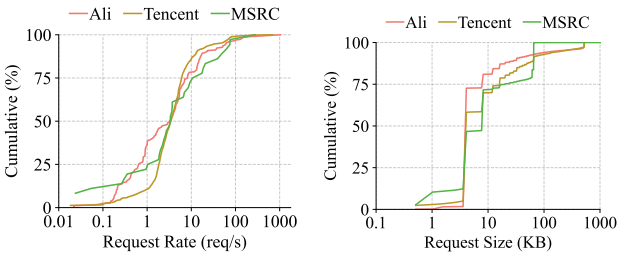
Log-Structured Storage (LSS) optimizes writes by sequentially updating data in an append-only manner, avoiding in-place modifications. It leverages the multi-stream capabilities of modern SSDs to reduce latency and increase throughput. The capacity of LSS is divided into fixed-size segments. Each segment consists of fixed-size blocks, where a block serves as the minimum unit for user requests and is identified by a logical block address (LBA). For writes or updates, LSS appends blocks to a segment that has not yet reached its maximum size. Once a segment is full, it becomes immutable, prompting LSS to open a new one.

LSS requires garbage collection (GC) operations to reclaim space. When GC is triggered, the system must execute a four-phase process: (1) selecting candidate segments via victim selection algorithms; (2) validating data block validity through segment scanning; (3) migrating valid data to new segments; and (4) erasing and reclaiming the original segment space. The additional write operations significantly degrade the performance of the storage system. Consequently, numerous data placement algorithms [4, 17, 18, 25, 26] have been proposed to reduce WA. The core idea behind these algorithms is to group data blocks with similar invalidation times, ensuring that the data blocks within a segment are invalidated around the same time, which can minimize GC overhead.

2.2 LSS on SSD Arrays

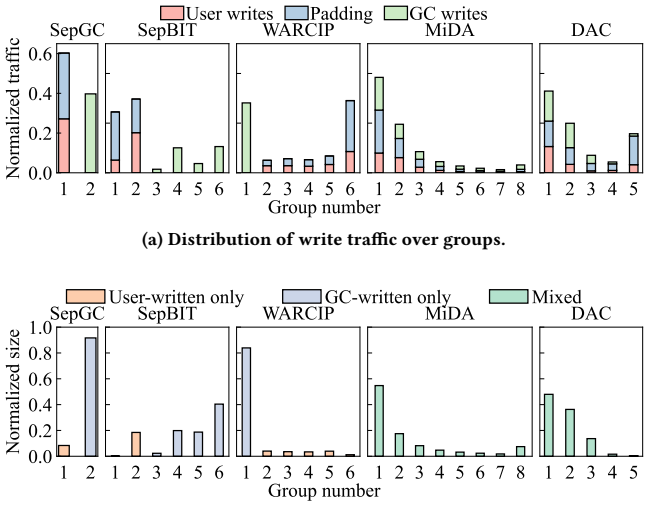
SSD arrays (also called SSD-based RAID) are a type of storage organization that combines multiple SSDs into a single logical unit to aggregate their capacity and throughput while providing fault tolerance. SSD arrays typically use chunks as logical storage units for data allocation and operations. A chunk is an abstract data unit, usually composed of multiple physical pages [2]. Combined chunks from different disks form a stripe, which includes data and parity blocks for fault tolerance and error correction.

Handling sub-chunk requests requires differentiated strategies. For reads, systems fetch entire chunks encompassing the requested data. Writes incur read-modify-write (RMW) penalties from mandatory chunk-wide reads and rewrites. Modern implementations mitigate this through zero-padding: appending dummy data to



(a) Request rate distribution (b) Request size distribution

Figure 2: The cumulative distribution of the average request rate (a) and the average write request size (b) in real-world workloads.



(a) Distribution of write traffic over groups.

(b) Distribution of group sizes, with different colors representing whether it is limited to user/GC write or not

Figure 3: Analysis of five current data placement strategies under production workload.

partial writes constructs complete chunks for direct submission, bypassing read operations while maintaining compatibility with log-structured append-only architectures. Concurrently, adopting append-oriented writes over in-place logical block updates reduces SSD internal write amplification, thereby enhancing garbage collection efficacy and wear-leveling optimization [12].

Figure 1 illustrates the architecture of LSS deployed on an SSD array and provides an example of how it handles write requests (e.g., requests 1-6). The LSS detects the hotness of write requests and places them into the corresponding segment (e.g., group G_0) based on the data placement strategy. The data blocks are distributed across different data columns in the SSD array for persistence, and parity blocks are generated in the parity column. If data is smaller than a chunk size and remains unfilled, padding ensures it meets I/O unit requirements. For instance, request blocks 1 and 2 are smaller than the chunk size and are coalesced into one data column. Request 3 is larger than a chunk size and is flushed into another data column. Similarly, requests 4-6 are coalesced together and flushed into the third data column, then the system generates a parity chunk P_1. Since there is no subsequent I/O, the unfilled chunk will be padded to the size of one chunk for flushing under SLA constraints, while parity block P_2 is generated.

2.3 Motivation

While data placement strategies can significantly reduce WA to improve end-to-end performance, we observe that they incur a high padding overhead when deployed on modern SSD arrays, particularly in scenarios with strict Service Level Agreement (SLA) requirements. To demonstrate this, we reproduce several well-known data placement strategies [4, 18, 23, 25, 26] and conduct experiments using a log-structured storage prototype. We analyze real-world production workloads [13, 16, 30] from block storage systems and replay them to evaluate write performance, yielding the following observations.

Observation 1: Many of the real-world workloads are mainly composed of small I/Os and have a low average request rate. We use block-level I/O traces collected from AliCloud and Tencent Cloud production cloud storage systems, along with MSRC traces from enterprise data centers. We analyze all volumes from these three traces collectively and present statistics on access density and average request size. Figure 2 shows the distribution of request rates and request sizes of three workloads. As illustrated in Figure 2a, the access density is very sparse for nearly all volumes. Specifically, only 1.9%-2.7% of volumes exhibit average intensities exceeding 100 req/s, while 75%-86.1% of volumes have average intensities below 10 req/s. We further examine the characteristics of write requests and find that small-sized I/Os dominate these workloads. Figure 2b presents the cumulative distributions of write request sizes across all I/O requests. Notably, 69.8%-80.9% of writes are no larger than 8 KiB in these production workloads, while only 10.8%-23.4% of writes exceed 32 KiB, confirming the prevalence of small I/Os dominating the workloads.

We then analyze current data placement technologies by replaying the Alibaba Cloud traces, adhering to the SLA settings of the Pangu storage system [29] from Alibaba. We set the chunk size to 64 KB, which is the default configuration in *Linux mdraid*. We focus on the write traffic distribution under real access density and different data placement algorithm group configurations, replaying the access interval based on the actual timestamps. Figure 3a illustrates the distribution of write volumes across groups with different data placement strategies, while Figure 3b depicts the group settings and the distribution of group size.

Observation 2: Zero padding operations occur predominantly in user-written and mixed-written groups, with minimal presence in GC-rewritten groups. This indicates that zero padding operations primarily arise during the user write phase and rarely during GC. For instance, SepGC segregates user writes from GC writes, with padding accounting for 54.9% of total writes in the user-written group (Group 0) and less than 1% in the GC-rewritten group (Group 1). MiDA utilizes the age of a block for data placement, and all groups can handle user requests, with padding traffic accounting for 33.3%-45% of the overall write volume. This is because

background I/O operations, such as GC, are typically performed in a centralized manner. These operations are often large in scale, and the system usually does not need to wait for the aggregation of requests when processing them.

Observation 3: Splitting user-written blocks across multiple groups often increases padding operations. The total padding traffic can be estimated by summing the padding across all groups. For instance, SepGC has one user-written group, whereas SepBIT has two. We estimate the padding traffic of the two methods to be 33.1% and 40.87%, respectively. Moreover, WARCIP, MiDA and DAC have five or more groups for handling user writes, resulting in 40.9%-42.7% more padding operations. This is because the increase in user-written groups results in each group receiving fewer requests and reduces write aggregation efficiency, leading to required padding for any group during sparse access requests.

Observation 4: Groups handling GC writes are generally larger than those handling user writes. As depicted in Figure 3b, SepGC, SepBIT, and WARCIP decouple user-written groups and GC-rewritten groups for WA reduction, but the GC groups occupy 83.9%–91.6% of the total capacity. In temperature-based algorithms, most of the user-written blocks are migrated to GC groups due to inaccurate identification. Although MiDA and DAC permit user and GC writes to coexist in the same group, the traffic from GC still exceeds user writes.

2.4 Summary

Our analysis reveals three critical guidelines for designing efficient data placement strategies in SSD arrays under real-world workloads. Based on Observations 1 and 2, **decoupling user writes from GC writes** is essential to minimize padding overhead during sparse access phases, as GC operations inherently exhibit lower padding requirements due to their bulk nature. Based on Observation 3, **maintaining a moderate number of temperature-based groups** is vital to balance access density and padding costs; over-partitioning groups amplifies padding inefficiencies while undermining GC effectiveness. Based on Observation 4, **precise data temperature tracking** must be prioritized to reduce redundant block migrations across groups, which otherwise worsen WA through repeated GC-induced relocations. These principles motivate the design of our novel data placement strategy.

3 Design

In this section, we present an access-density-based data placement strategy called ADAPT, which is based on the principles mentioned above, to improve the GC efficiency in LSS. We first provide a high-level overview of ADAPT and introduce the relationships among the key components. Next, we describe the design of each element in detail.

3.1 Overview of ADAPT

Figure 4 shows the overall architecture of ADAPT. ADAPT defines six classes of groups, two of which correspond to the segments of user-written blocks and the remaining four to the segments of GC-rewritten blocks. Each group is configured with one open segment and contains multiple sealed segments.

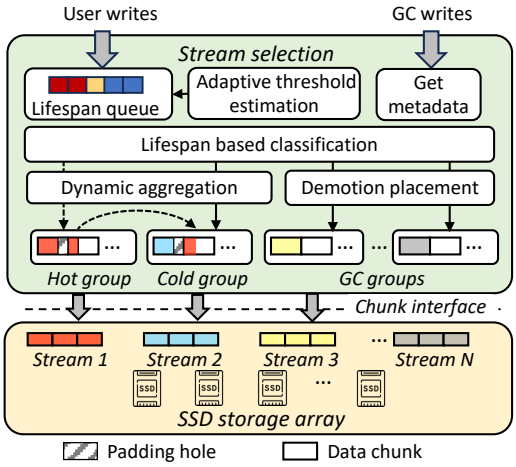


Figure 4: Overview of ADAPT.

ADAPT separates user-written blocks and GC-rewritten blocks and places them into different groups. Like SepBIT, ADAPT employs a lifespan-based policy to assess the hotness of user-written blocks and estimates the residual lifespan of GC-rewritten blocks using an age-based approach. Since access density also affects the GC of user-written groups, we dynamically adjust the threshold (seen in §3.2) for separating data blocks by considering block-level popularity and workload-level access density. User-written blocks are assigned to either the hot or cold groups based on whether their lifespan exceeds the threshold. For intensive access workloads, where data chunks can accumulate sufficient data quickly, ADAPT flushes them directly to the SSD array. For sparse access workloads, where the data chunks require padding to meet size requirements, ADAPT delays the persistence of blocks in the hot group through cross-group dynamic aggregation (seen in §3.3). Additionally, ADAPT employs proactive demotion placement design by initially assigning user-written data to GC-rewritten groups (seen in §3.4), which effectively reduces data migration caused by GC.

ADAPT serves as an intermediate layer between log-structured storage and SSD array systems. Since ADAPT does not modify the SSD array, it maintains strong generalization ability. It can also leverage SSDs' multi-stream capability to reduce in-device WA by mapping groups to streams one-to-one.

3.2 Density-Aware Threshold Adaptation

Previous data placement algorithms only separate user-written data blocks based on block popularity, without considering access density, leading to suboptimal performance. ADAPT minimizes WA from user-written data blocks by adjusting the separation threshold based on block-level popularity and workload-level access density. We first illustrate how ADAPT estimates the hot-cold threshold, and then explain the details of the internal algorithm.

We expect the data to be grouped into user-written groups with minimal GC overhead and padding traffic. ADAPT adjusts the threshold by simulating the WA performance under different

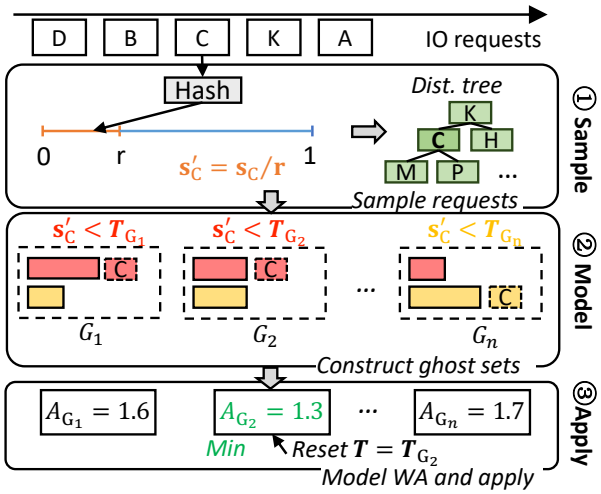


Figure 5: The scheme of threshold adaptation is based on block-level popularity and workload-level access density.

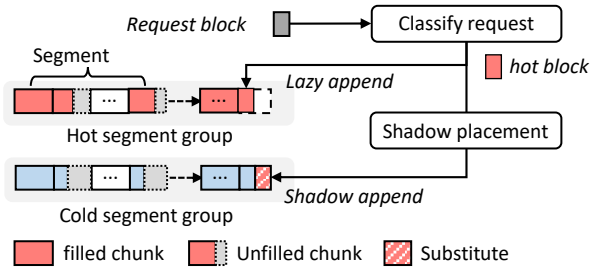


Figure 6: The aggregation of hot user-written data. The hot data blocks can generate substitutes and persist them in cold segment groups, which are then lazily flushed into the original hot segment groups.

threshold grouping scenarios. Figure 5 demonstrates the architecture of threshold adaptation and gives an instance. The incoming stream of request blocks, DBCKA, is first sent to the sampling module with sampling rate r. If a block (e.g., block C) is sampled, its last access interval $S_c^{'}$ is computed and assigned to a series of ghost sets M with incremental hot-cold thresholds T_M. For each ghost set M_i, the block is placed in the hot segment group if $S_c^{'}$ is less than the threshold T_{M_i}; otherwise, it is placed in the cold segment group. Ghost sets also require GC when the garbage ratio meets the condition. The WA of ghost sets will gradually stabilize after multiple GCs. ADAPT then compares the WA of each ghost set and updates the actual threshold based on the configuration of the ghost set with the smallest WA.

Tracking workload characteristics. Since full workload analysis would result in unacceptable resource consumption, ADAPT employs spatial sampling [24] to reduce memory overhead while preserving hot and cold characteristics. Specifically, request blocks are uniformly sampled via a hash algorithm. A distance tree is used to record the access sequence of sampled request blocks. ADAPT calculates the access interval for each sampled block as the number of other intervening unique request blocks referenced since the last access to that block, and scales the access interval of sampled blocks by the actual sampling rate to obtain the real access interval. ADAPT then constructs the ghost sets based on the sampled request blocks and their access intervals.

Ghost set simulation. The ghost sets simulate the GC for user-written groups by only tracking the logical block addresses (LBA) of request blocks, and each set uses a different hot-cold threshold. If the access interval is less than the threshold, the block is placed into the hot group; otherwise, it is put it in the cold group. The thresholds change in granularity based on segment size. To determine the appropriate value, they are initialized using an exponentially growing sliding window, and then switched to the linear growth mode after the first adjustment. For example, if T_{G_k} is the first selected threshold, the range for the second iteration spans from $T_{G_{k-1}}$ to $T_{G_{k+1}}$. The unit of threshold adjustment is the segment size in the ghost set. If the WA simulation exhibits monotonicity with the threshold, the exponentially growing sliding window is reused to adapt to changes in workload characteristics.

Unlike the original LSS, the capacity of segments in ghost sets is proportionally scaled by the sampling rate. Meanwhile, the chunk aggregation time is proportionally increased. The ghost set's GC trigger condition matches the original system, activating when the garbage ratio exceeds the limit. But unlike the standard GC process, ghost sets discard valid blocks without rewriting them. This is because the data placement algorithm migrates valid blocks from user-written groups to GC-rewritten groups before segment reclamation, where ghost sets only preserve user-written groups. For each GC, the ghost set tracks the number of valid blocks discarded in the GC process, so we can calculate WA by determining the ratio of discarded blocks to written blocks.

Updating threshold configuration. During the cold start of the ghost set simulation, we configure the initial threshold via the lifespan of segments in the hot group, which is defined as the number of unique user-written bytes in the workload since the segment is created until GC reclaims it. Once the ghost set stabilizes with minimal WA, the configuration updates from the best simulation results. ADAPT monitors write volume and GC conditions to determine whether ghost sets have an authentic simulation. Specifically, when the write volume exceeds 10% of storage capacity or the WA is steady, ADAPT scales the threshold to real segment size and applies the configuration from the set with minimal WA.

3.3 Cross-Group Dynamic Aggregation

As described in Section 2.3, setting multiple groups for user-written blocks may introduce additional padding traffic in sparse workloads due to the inefficient write aggregation mechanism. To address this, we propose cross-group aggregation. Our key insight is to utilize redundant blocks in unfilled chunks of cold groups to temporarily ensure the persistence of hot data blocks to meet SLA requirements. This is because access density and data hotness are not strictly orthogonal in practice, hot data blocks often have higher access intensity and the potential to aggregate into a complete chunk within a short time.

Figure 6 illustrates the aggregation process involving a hot group and a cold group. We aggregate individual data blocks rather than entire unfilled chunks, as finer granularity enables more flexible aggregation criteria. Hot data blocks are consolidated into a chunk within the hot group. If this chunk is predicted to remain unfilled after in-group aggregation, we utilize an unfilled chunk from another user-written group to construct a filled chunk, completing the persistence of these hot blocks. This cross-group persistence is termed **shadow append**, while the original data block writing is referred to as **lazy append**. Subsequently, the shadow-appended blocks are flushed to disk for reliability. The original chunk in the hot group can then accommodate new incoming data blocks. Note that the chunk retains the original data blocks; only its aggregation timer is reset.

Aggregation condition. Cross-group dynamic aggregation performs effectively under sparse workload access density. ADAPT monitors the write rate for each user-written group and triggers cross-group aggregation when a group cannot accumulate sufficient data to avoid padding. Specifically, for group i, ADAPT tracks the number of data blocks V_i forming k segments and the padding count P_i, then computes the average accumulated size of unfilled chunks C_i as:

$$C_i = \frac{V_i - S_{ck} * (k * N_{ck} - P_i)}{P_i} \quad (1)$$

where S_{ck} is the chunk size and N_{ck} is the chunk number of a segment. Then ADAPT determines the aggregation condition in two steps. First, it predicts whether a chunk requires dynamic aggregation based on previous access intervals, as access density is continuous and evolves. Thus, if a chunk remains unfilled within a short interval, the next one is also likely to be unfilled. Second, when a chunk already contains lazily appended data blocks, ADAPT decides whether to continue aggregating subsequent requests. Specifically, if the aggregated block bytes in the segment to be written exceed the average padding size of the group, ADAPT stops aggregating new data blocks across groups.

Group selection for shadow append. ADAPT always selects the colder segment group as the shadow group for the hot segment group for two reasons. First, cold segment groups inherently exhibit lower write rates and longer chunk accumulation intervals, leaving more stable unused space in their unfilled chunks. This allows hot groups with high write rates to efficiently utilize these padding spaces for shadow append. Second, using hot segment groups as shadow groups may lead to unnecessary data migration. Substitute blocks expire once the original blocks are persisted via lazy append, causing shadow-appended segments to have a shorter lifespan than the group average. Consequently, these segments are prioritized for GC. Since block lifespans in hot segments are shorter than in cold segments, early GC in hot segments undermines cold-hot separation, resulting in higher write amplification.

3.4 Proactive Demotion Placement

Efficient data placement is crucial for reducing GC overhead. After decoupling user-written groups and GC-rewritten groups, we find that most data blocks fall into the coldest group under the age-based placement policy, especially in highly skewed production workloads. This is because most of the data blocks are rarely

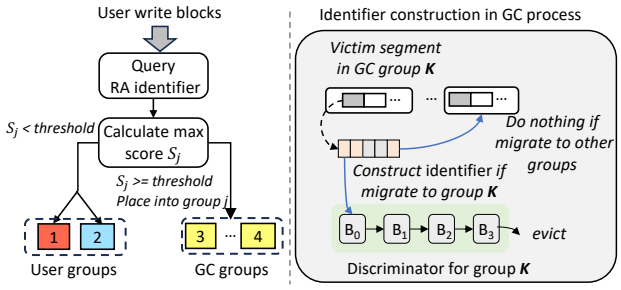


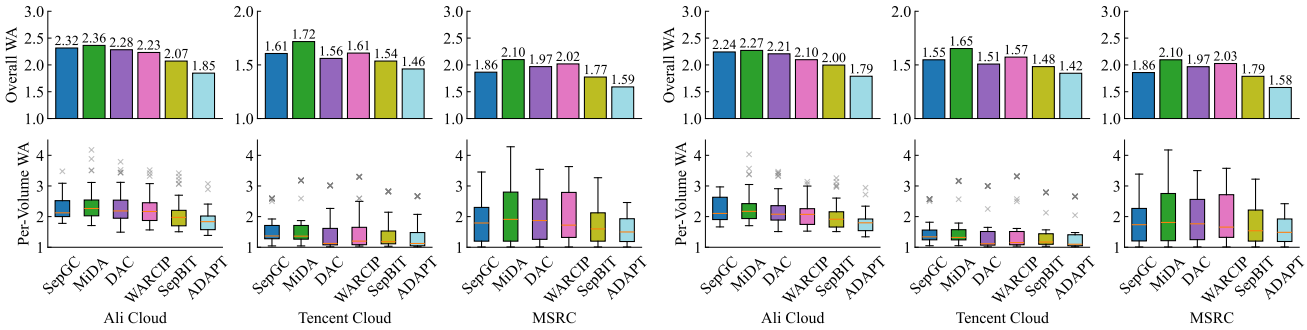
Figure 7: The left figure illustrates the proactive demotion placement process, while the right figure shows the construction of the identifier.

updated under Zipf's law. Consequently, the dominant rewrite traffic stems from repeatedly migrating blocks through progressively colder groups before they reach the coldest group.

To optimize traffic reduction for GC-rewritten blocks, our strategy prioritizes accurate identification and placement of long-lived cold blocks. The proactive demotion placement mechanism utilizes the established correlation between a block's prior lifespan and its projected lifespan [25] and assigns blocks to groups aligned with their expected lifespans. As illustrated in Figure 7, our design process incorporates a re-access (RA) identifier to detect potential long-lived data blocks. During block request processing, the RA identifier computes placement scores across GC-rewritten groups using a specialized metric. The group with the highest score is considered a candidate. If the score exceeds the pre-defined threshold, ADAPT then allocates the block to the respective group; otherwise, ADAPT uses the standard hotness-based placement algorithm to assign it to user-written groups. The score is defined as the number of times the data block is rewritten to each GC-rewritten group. Because the lifetimes of segments in the same group are similar, the block that is often migrated to the same group has a long lifespan in that group.

The RA identifier is constructed in the GC process. ADAPT uses the latest data considered valid blocks in GC to build the identifier. Specifically, when the GC process selects a victim segment in the GC-rewritten group K for cleaning, ADAPT chooses the valid blocks that are migrated back to the origin group K for identifier construction. ADAPT inserts LBAs of corresponding blocks into the discriminator in group K.

Since the identifier's lookup operation resides on the critical path of I/O processing, it is necessary to have a lightweight design to achieve high performance while consuming minimal computational resources. To achieve this, the cascading discriminator consists of several bloom filters, each of which can track the inserted data by assessing whether any bits have been modified. The lookup operation only needs some hash calculations, with an overhead of nanoseconds. The cascaded discriminator restricts the number of elements and evicts the oldest bloom filter in a FIFO manner to reduce memory overhead. When a block checks the identifier, the score for a group is the number of bloom filters with that LBA in that discriminator.



(a) Greedy selection (b) Cost-Benefit selection

Figure 8: GC efficiency of two selection policies. For each policy, the top three bar charts display the overall WA across three real-world workloads, while the bottom three box plots illustrate the per-volume WA distribution.

4 Evaluation

4.1 Experimental Setup

Baselines. We evaluate ADAPT against five existing data placement schemes: SepGC [23], Dynamic Data Clustering (DAC) [4], WARCIP [26], MiDA [18], and SepBIT [25]. SepGC serves as the baseline approach, separating user-written blocks and GC-rewritten blocks into two distinct groups—a strategy widely adopted in key-value stores [3]. Other designs employ one or multiple metrics to measure block temperatures, such as access counts (DAC), age (MiDA), update intervals (WARCIP), or block lifespan estimation (SepBIT). We maintain the default group configurations specified in the original papers for these schemes. Specifically, WARCIP uses five groups for user-written blocks and one for GC-rewritten blocks, while SepBIT uses two groups for user-written blocks and four for GC-rewritten blocks. For DAC and MiDA, which do not distinguish between user-written and GC-rewritten blocks, we configure five and eight groups, respectively, for all written blocks.

Workloads. We evaluate our approach using block I/O traces collected from diverse environments. For real-world workloads, we select 50 volumes from two cloud block I/O traces: Alibaba [13] and Tencent [30], along with MSRC traces [16] from enterprise data centers. For qualitative performance analysis, we generate the YCSB-A workload using the YCSB [5] cloud benchmark to examine the impact of varying access densities and Zipfian ratios.

We employ trace-driven simulations to efficiently assess key performance metrics of garbage collection techniques. Additionally, we implement ADAPT in a log-structured storage prototype to measure I/O performance and overheads. Our experimental setup consists of an SSD array with four Samsung MZQL2960HCJR-00A07 SSDs, an Intel Xeon Silver 4314 CPU, and 128GB of memory. Both the simulator and prototype use a 4KB block size for storage organization and a default 64KB chunk size in the underlying SSD array. We set the data chunk coalescing time to 100 µs, consistent with production environments in cloud block storage [29].

4.2 General Result

GC efficiency. We first evaluate GC efficiency by comparing the WA of various techniques. All experiments are conducted using a trace-driven simulator. We use Greedy and Cost-Benefit [19] policies for victim segment selection in GC. Figure 8 shows the overall WA of these two policies. As we can see, ADAPT achieves the lowest overall WA compared to all existing data placement schemes. Specifically, for the Ali Cloud workload, ADAPT reduces WA by 30.8%, 32.5%, 33.1%, 30.8%, and 21.8% on average with the Greedy selection policy, compared to SepGC, MiDA, DAC, WARCIP, and SepBIT, respectively. For the Tencent Cloud workload, where data access is more skewed, the average WA of each algorithm is lower than the Alibaba workload. ADAPT achieves a 14.8% -36.1% reduction in WA compared to other placement schemes. Notably, in the read-intensive MSRC workload, ADAPT reduces WA by 31.4%, 46.3%, 39.1%, 41.5%, and 23.3%, respectively. We also observe that some data placement schemes exhibit higher WA than SepGC, which simply separates user-written blocks and GC-rewritten blocks. This occurs primarily because these schemes manage numerous groups for user-written blocks and fail to effectively fill full data chunks.

Under the Cost-Benefit selection policy, all data placement schemes exhibit lower overall WA than with the Greedy policy in both Ali Cloud and Tencent Cloud workloads. ADAPT maintains its advantage even with this improved GC selection strategy, reducing WA by 21%–38.2% and 12.5%–35.4% compared to other strategies in these workloads. Notably, WARCIP and SepBIT show less WA reduction with the Cost-Benefit policy than with the Greedy policy, while SepGC, MiDA, and DAC show no improvement. In contrast, ADAPT demonstrates better universality, achieving a 1.7% lower WA than the Greedy policy.

Furthermore, Figure 8 presents boxplots of per-volume WA. ADAPT achieves the lowest median and 25th/75th percentile WA across all data placement schemes, regardless of the GC policy. In most cases, Cost-Benefit is more effective than Greedy in mitigating WA. Compared to SepBIT, ADAPT reduces the 75th percentile WA by 9.4–16.9% with Greedy and 9.9–24% with Cost-Benefit. Additionally, ADAPT exhibits the fewest WA outliers across both cloud block storage workloads, demonstrating its effectiveness in WA mitigation for volumes with diverse access patterns and densities.

Coalescing efficiency. Next, we analyze the coalescing efficiency of these data placement schemes and their relationship with

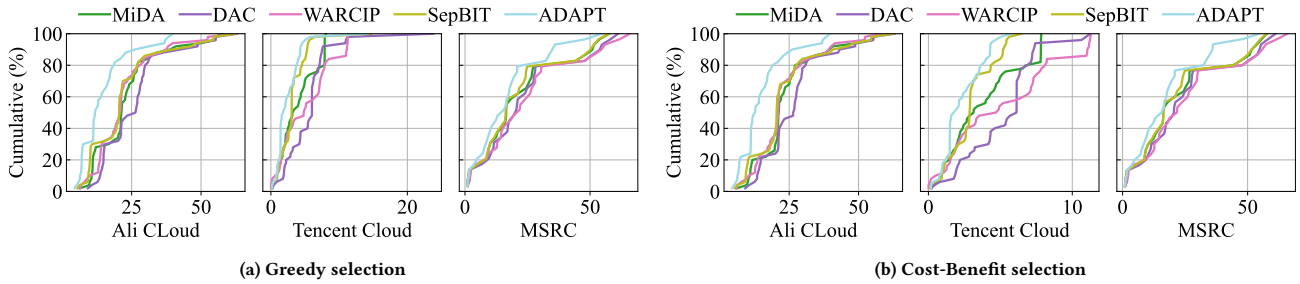


Figure 9: The cumulative distribution of padding volume ratio under various production workloads. The x-axis of the subgraphs represents the Padding Traffic Ratio(%).

WA reduction. Figure 9 shows the reduction in zero-padding volume achieved by ADAPT compared to other temperature-based placement schemes.

Overall, ADAPT reduces padding volumes more effectively than other temperature-based algorithms across all three workloads, regardless of the GC selection policy. Specifically, in Ali Cloud workloads, ADAPT ensures that over 88% of volumes have a padding traffic ratio below 25%, whereas SepBIT achieves this for only 70% of volumes. Other placement algorithms perform worse due to excessive group separation.

In Tencent workloads, all volumes achieve under 6.7% padding traffic with ADAPT or SepBIT, compared to 7.8% with MiDA and 11.2% with DAC or WARCIP. Similar results hold for the Cost-Benefit policy, where padding traffic decreases significantly across all placement strategies. This reduction correlates with the observed WA improvement under this policy.

To further investigate the relationship between padding traffic and WA, we analyze the per-volume WA reduction relative to the padding traffic reduction achieved by ADAPT compared to other placement strategies. Using the Greedy selection algorithm, we compare ADAPT with MiDA and SepBIT, both of which also infer temperature based on lifespan. Figure 10 presents the results for the Ali Cloud workload, where each point represents an individual volume. The x-axis indicates the percentage reduction in padding write traffic achieved by ADAPT compared to the given algorithm, while the y-axis shows the corresponding WA reduction. The results demonstrate that the reduction of WA is strongly correlated with the reduction of padding traffic. For example, among volumes with padding write traffic exceeding 40%, ADAPT achieves a WA reduction of at least 21.0%, with a maximum reduction of 72.1% compared to MiDA.

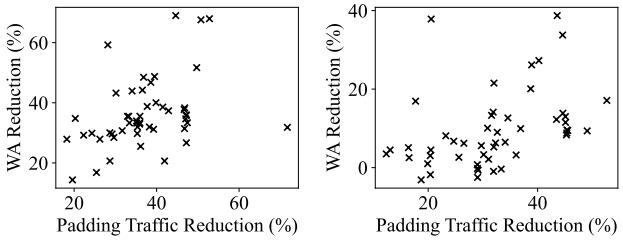


Figure 10: The correlation between padding traffic reduction and WA reduction.

4.3 Sensitivity Analysis

We next analyze the workload sensitivity of ADAPT and other data placement schemes. We generate update-heavy workloads with diverse traffic patterns and skewness using the YCSB benchmark, testing all data placement schemes with the Greedy selection policy. Each experiment fills 1M data blocks and measures WA after 10M writes. The default block size is 4KB, matching the common page size in storage systems.

Impact of access density. The left line chart in Figure 11 shows the WA factor under workloads with varying traffic intensities. The time interval between all write requests exceeds 100 µs in light traffic workloads and falls below 100 µs in medium and heavy traffic workloads. The key difference is that requests become sufficiently dense to eliminate zero-padding operations across all data placement schemes. As shown, ADAPT achieves the best performance under light traffic workloads, reducing GC writes by 21.2%–53.5% compared to other schemes. This improvement stems from ADAPT's effective cross-group aggregation, which minimizes zero-padding operations. Notably, MiDA and WARCIP consistently exhibit higher WA than SepGC, while SepGC performs second only to ADAPT under light loads. This suggests that simply separating user writes from GC writes is more effective than dividing user writes into multiple groups for light workloads.

As workload intensity increases, the WA of all data placement schemes decreases. Under heavy traffic, ADAPT maintains an advantage, reducing GC write volume by 5.2%–22.4% compared to other schemes.

Impact of workload skewness. Since YCSB-A follows a Zipfian distribution, we simulate varying access locality by generating workloads with increasing α. The right line chart in Figure 11 illustrates the sensitivity of each scheme to access locality. When $\alpha = 0$ (uniform distribution), all schemes exhibit similar WA because data block temperatures are nearly identical. As locality increases ($\alpha > 0$), WA declines across all schemes. ADAPT achieves the lowest WA at $\alpha = 0.9$, where the workload exhibits strong locality

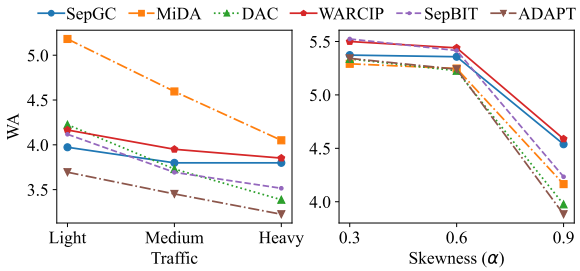


Figure 11: The impact of access density (left) and workload skewness (right).

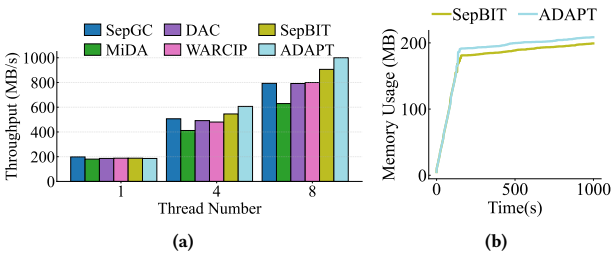


Figure 12: Throughput results (a) and memory usage (b).

(approximately 80% of traffic targets the top 20% of blocks). This demonstrates ADAPT's ability to reduce GC overhead by dynamically adjusting cold and hot thresholds.

4.4 Experiments on Prototype

In this section, we evaluate throughput performance and memory overhead, which cannot be measured using the trace-driven simulator. While ADAPT is optimized for sparse writing scenarios, it also maintains high performance under heavy workloads.

We implement the ADAPT prototype application on the SSD array and carry out a set of experiments. We use mdraid [1] to configure a RAID-5 system with 4 SSDs. The prototype incorporates a user-space aggregation mechanism to support zero-padding operations and persist data at the chunk level. For throughput experiments, we employ the default YCSB-A workloads described in §4.3. To maximize device utilization, we set the I/O depth to 8 and write chunks asynchronously. The number of background GC threads matches the number of client threads to ensure efficiency.

Figure 12a shows the throughput results for one, four, and eight clients. Specifically, for one client scenario, all data placement schemes yield similar results because the disk throughput has not yet been fully utilized; SepGC has the highest throughput due to its simple strategy. In contrast, ADAPT achieves 1.11-1.47× higher throughput with 4 clients and 1.1-1.58× higher with 8 clients compared to other data placement schemes. This is because as the number of threads increases, disk bandwidth gradually becomes the bottleneck constraining performance. By reducing GC writes, ADAPT effectively increases available user bandwidth, yielding significant performance gains.

We evaluate the memory overhead of ADAPT and compare it with SepBIT, as both have the same number of groups and use a lifespan-based policy, and have similar overhead for hotness modeling. Figure 12b demonstrates the result. Specifically, ADAPT incurs a memory overhead of 4.56% higher than SepBIT, which is a reasonable overhead with the WA reduction of 20%-25%. The additional memory usage in ADAPT stems from two components. First, the sampling module requires approximately 44 bytes per block to record sampled blocks and their time intervals. At a 0.001 sample rate, this amounts to roughly 11 MB of memory. The second part is the ghost set simulation, whose memory usage varies with threshold adjustments. The main memory usage for the ghost set is the record of LBAs and the index of data blocks. ADAPT needs about 20 bytes to simulate data placement for each data block. Note that ADAPT can reduce memory usage by reducing the sampling rate and the volume of simulation, which provides users with flexible configuration options.

5 Related Work

GC optimization in log-structured storage. Many studies aim to optimize GC overhead by focusing on two aspects: victim segment selection and data placement. For segment selection, Greedy and Cost-Benefit policies are widely adopted due to their broad applicability. Additionally, several studies propose variants of the Greedy policy, such as d-choice [22], Windowed Greedy [8], and Random Greedy [15]. For data placement, beyond the strategies evaluated in Section 4, other approaches employ machine learning-based policies [17] or access prediction using program contexts [11] on SSD platforms. Other work optimizes GC by integrating application-specific techniques, such as key-value storage[20, 31, 33], which—like LSS—organizes data in an append-only manner. For instance, NovKV [20] eliminates GC-induced queries and insertions, while DumpKV [33] reduces write amplification through lifetime-aware garbage collection. ADAPT focuses on data placement optimization for sparse I/O scenarios and can be extended to other placement algorithms.

Write optimization in SSD arrays. Traditional RAID architectures provide fault tolerance, while modern disk arrays adopt a log-structured design to generate sequential writes better suited for flash-based SSDs. Some studies improve GC performance by mitigating interference [10] or exploring new interface designs [21, 28]. Others optimize partial or sparse writes using techniques like two-phase write [9, 27]. In contrast, ADAPT maintains compatibility with the standard chunk interface and remains orthogonal to these approaches.

6 Conclusion

In this paper, we present ADAPT, a novel data placement scheme designed to mitigate WA in log-structured storage deployed on SSD arrays. To reduce zero-padding operations in sparse access scenarios, ADAPT dynamically adjusts the hot-cold boundary and coalesces write requests using a cross-group approach. Additionally, ADAPT proactively coordinates garbage collection processes to eliminate redundant cold data migrations through predictive group downgrading. Evaluations on production workloads demonstrate that ADAPT achieves the lowest WA compared with five state-of-the-art data placement schemes.

Acknowledgments

We thank the anonymous reviewers for all their helpful comments and suggestions. This work was supported by the National Key Research and Development Program (Grant No.2023YFB4502701), the National Natural Science Foundation of China (Grant No.62232007) and the China Postdoctoral Science Foundation (Grant No.2025M771551).

References

[1] 2025. Linux mdraid layer. https://github.com/torvalds/linux/tree/master/drivers/md

[2] Neil Brown. 2001. Software RAID in 2.4. In *Proceedings of linux. conf. au, Sydney, Australia*.

[3] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 1007–1019. https://www.usenix.org/conference/atc18/presentation/chan

[4] Mei-Ling Chiang, Paul CH Lee, and Ruei-Chuan Chang. 1999. Using data clustering to improve cleaning performance for flash memory. *Software: Practice and Experience* 29, 3 (1999), 267–290.

[5] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[6] 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.

[7] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. 2021. ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 147–162. https://www.usenix.org/conference/osdi21/presentation/han

[8] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. 2009. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. 1–9.

[9] Tianyang Jiang, Guangyan Zhang, Zican Huang, Xiaosong Ma, Junyu Wei, Zhiyue Li, and Weimin Zheng. 2021. FusionRAID: Achieving Consistent Low Latency for Commodity SSD Arrays. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 355–370. https://www.usenix.org/conference/fast21/presentation/jiang

[10] Jaeho Kim, Kwanghyun Lim, Youngdon Jung, Sungjin Lee, Changwoo Min, and Sam H. Noh. 2019. Alleviating Garbage Collection Interference Through Spatial Separation in All Flash Arrays. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 799–812. https://www.usenix.org/conference/atc19/presentation/kim-jaeho

[11] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. 2019. Fully automatic stream management for {Multi-Streamed}{SSDs} using program contexts. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 295–308.

[12] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. {F2FS}: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. 273–286.

[13] Jinhong Li, Qiuping Wang, Patrick P. C. Lee, and Chao Shi. 2023. An In-depth Comparative Analysis of Cloud Block Storage Workloads: Findings and Implications. *ACM Trans. Storage* 19, 2, Article 16 (March 2023), 32 pages. https://doi.org/10.1145/3572779

[14] Qiang Li, Qiao Xiang, Yuxin Wang, Haohao Song, Ridi Wen, Wenhui Yao, Yuanyuan Dong, Shuqi Zhao, Shuo Huang, Zhaosheng Zhu, Huayong Wang, Shanyang Liu, Lulu Chen, Zhiwu Wu, Haonan Qiu, Derui Liu, Gexiao Tian, Chao Han, Shaozong Liu, Yaohui Wu, Zicheng Luo, Yuchao Shao, Junping Wu, Zheng Cao, Zhongjie Wu, Jiaji Zhu, Jinbo Wu, Jiwu Shu, and Jiesheng Wu. 2023. More Than Capacity: Performance-oriented Evolution of Pangu in Alibaba. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 331–346. https://www.usenix.org/conference/fast23/presentation/li-qiang-deployed

[15] Yongkun Li, Patrick PC Lee, and John CS Lui. 2013. Stochastic modeling of large-scale solid-state storage systems: Analysis, design tradeoffs and optimization. In *Proceedings of the ACM SIGMETRICS/international conference on Measurement and modeling of computer systems*. 179–190.

[16] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. 2008. Write off-loading: Practical power management for enterprise storage. *ACM Trans. Storage* 4, 3, Article 10 (Nov. 2008), 23 pages. https://doi.org/10.1145/1416944.1416949

[17] Seonggyun Oh, Jeeyun Kim, Soyoung Han, Jaeho Kim, Sungjin Lee, and Sam H. Noh. 2024. {MIDAS}: Minimizing Write Amplification in {Log-Structured} Systems through Adaptive Group Number and Size Configuration. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. 259–275.

[18] Hyunseung Park, Eunjae Lee, Jaeho Kim, and Sam H. Noh. 2021. Lightweight data lifetime classification using migration counts to improve performance and lifetime of flash-based SSDs. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems* (Hong Kong, China) *(APSys '21)*. Association for Computing Machinery, New York, NY, USA, 25–33. https://doi.org/10.1145/3476886.3477520

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

[20] Chen Shen, Youyou Lu, Fei Li, Weidong Liu, and Jiwu Shu. 2020. Novkv: efficient garbage collection for key-value separated lsm-stores. In *36th International Conference on Massive Storage Systems and Technology (MSST 20)*.

[21] Zhenhua Tan, Linbo Long, Jingcheng Shen, Renping Liu, Congming Gao, Kan Zhong, and Yi Jiang. 2024. Optimizing Garbage Collection for ZNS SSDs via In-storage Data Migration and Address Remapping. 21, 4, Article 77 (Nov. 2024), 25 pages. https://doi.org/10.1145/3689336

[22] Benny Van Houdt. 2013. A mean field model for a class of garbage collection algorithms in flash-based solid state drives. *SIGMETRICS Perform. Eval. Rev.* 41, 1 (June 2013), 191–202. https://doi.org/10.1145/2494232.2465543

[23] B. Van Houdt. 2014. On the necessity of hot and cold data identification to reduce the write amplification in flash-based SSDs. *Performance Evaluation* 82 (2014), 1–14. https://doi.org/10.1016/j.peva.2014.08.003

[24] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. 2015. Efficient MRC construction with SHARDS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (Santa Clara, CA) *(FAST'15)*. USENIX Association, USA, 95–110.

[25] Qiuping Wang, Jinhong Li, Patrick P. C. Lee, Tao Ouyang, Chao Shi, and Lilong Huang. 2022. Separating Data via Block Invalidation Time Inference for Write Amplification Reduction in Log-Structured Storage. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 429–444. https://www.usenix.org/conference/fast22/presentation/wang

[26] Jing Yang, Shuyi Pei, and Qing Yang. 2019. WARCIP: write amplification reduction by clustering I/O pages. In *Proceedings of the 12th ACM International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR '19)*. Association for Computing Machinery, New York, NY, USA, 155–166. https://doi.org/10.1145/3319647.3325840

[27] Shushu Yi, Xiurui Pan, Qiao Li, Qiang Li, Chenxi Wang, Bo Mao, Myoungsoo Jung, and Jie Zhang. 2024. ScalaAFA: Constructing User-Space All-Flash Array Engine with Holistic Designs. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 141–156. https://www.usenix.org/conference/atc24/presentation/yi-shushu

[28] Shushu Yi, Shaocong Sun, Li Peng, Yingbo Sun, Ming-Chang Yang, Zhichao Cao, Qiao Li, Myoungsoo Jung, Ke Zhou, and Jie Zhang. 2024. BIZA: Design of Self-Governing Block-Interface ZNS AFA for Endurance and Performance. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 313–329. https://doi.org/10.1145/3694715.3695953

[29] Weidong Zhang, Erci Xu, Qiuping Wang, Xiaolu Zhang, Yuesheng Gu, Zhenwei Lu, Tao Ouyang, Guanqun Dai, Wenwen Peng, Zhe Xu, Shuo Zhang, Dong Wu, Yilei Peng, Tianyun Wang, Haoran Zhang, Jiasheng Wang, Wenyuan Yan, Yuanyuan Dong, Wenhui Yao, Zhongjie Wu, Lingjun Zhu, Chao Shi, Yinhu Wang, Rong Liu, Junping Wu, Jiaji Zhu, and Jiesheng Wu. 2024. What's the story in EBS glory: evolutions and lessons in building cloud block store. In *Proceedings of the 22nd USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) *(FAST '24)*. USENIX Association, USA, Article 17, 16 pages.

[30] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. 2020. OSCA: An Online-Model Based Cache Allocation Scheme in Cloud Block Storage Systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 785–798. https://www.usenix.org/conference/atc20/presentation/zhang-yu

[31] Ruisong Zhou, Yuzhan Zhang, Chunhua Li, Ke Zhou, Peng Wang, Gong Zhang, Ji Zhang, and Guangyu Zhang. 2024. HyperDB: a Novel Key Value Store for Reducing Background Traffic in Heterogeneous SSD Storage. In *Proceedings of the 53rd International Conference on Parallel Processing* (Gotland, Sweden) *(ICPP '24)*. Association for Computing Machinery, New York, NY, USA, 453–463. https://doi.org/10.1145/3673038.3673153

[32] Yanbo Zhou, Erci Xu, Li Zhang, Kapil Karkra, Mariusz Barczak, Wayne Gao, Wojciech Malikowski, Mateusz Kozlowski, Łukasz Łasek, Ruiming Lu, Feng Yang, Lilong Huang, Xiaolu Zhang, Keqiang Niu, Jiaji Zhu, and Jiesheng Wu. 2024. CSAL: the Next-Gen Local Disks for the Cloud. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 608–623. https://doi.org/10.1145/3627703.3629566

[33] Zhutao Zhuang, Xinqi Zeng, and Zhiguang Chen. 2024. DumpKV: Learning based lifetime aware garbage collection for key value separation in LSM-tree. arXiv:2406.01250 [cs.DB] https://arxiv.org/abs/2406.01250