FAD: A Fragmentation-Aware Data Layout Strategy for Deduplication-based SSDs

Haodong Lin*†, Wenyan You*, Zhehua Zhang*, Suzhen Wu*, Weichun Wang‡, Wei Wang‡, Yinhui Yu†, and Bo Mao*
*School of Informatics, Xiamen University, Xiamen, China
†School of Computer Science and Artificial Intelligence, Fuyao University of Science and Technology, Fuzhou, China
‡Hikvision, Hangzhou, China
Corresponding Authors: Yinhui Yu (yuyhui@fyust.edu.cn) and Bo Mao (maobo@xmu.edu.cn)

***Abstract*—Data deduplication is widely used in flash-based SSDs to improve space efficiency, but it can also introduce file fragmentation. By analyzing representative real-world deduplication workloads, we observe that deduplication can concentrate logically consecutive data blocks on the same flash chips, causing chip-level read conflicts and reducing SSD read parallelism. Existing deduplication methods do not distinguish different fragmentation patterns and often rely on coarse-grained rewriting, which can introduce excessive writes and capacity overhead.**

To address this problem, we propose FAD, a fragmentation-aware data layout strategy for deduplication-based SSDs. FAD classifies deduplication-induced fragmentation into uniformly distributed fragmentation (UDF) and non-uniformly distributed fragmentation (NUDF) according to the chip-level distribution of duplicate blocks. For UDF files, FAD places new non-duplicate blocks to improve read parallelism without additional rewriting. For NUDF files, FAD selectively rewrites high-reference-count duplicate blocks using elastic thresholds, while a write-parallelism optimization mechanism compensates for skipped chips and limits write-side overhead. Experimental results show that FAD reduces average read latency by 34.1%, 22.2%, and 15.8% compared with Deduplication, HAR, and FCRC, respectively.

***Index Terms*—Flash-based SSDs, Data Deduplication, File Fragmentation, Parallelism**

I. Introduction

The rapid growth of big data, artificial intelligence (AI), and high-performance computing (HPC) has increased the demand for storage systems with large capacity, high bandwidth, low latency, and long-term stability. Flash-based Solid-State Drives (SSDs) have therefore become the primary storage medium in many computing environments, from enterprise servers and data centers to personal devices such as notebooks [1], [2]. However, SSDs are constrained by the finite endurance of flash memory. Each flash cell can sustain only a limited number of program/erase (P/E) cycles. Once this limit is reached, cells begin to degrade, which increases error rates, reduces performance, and may eventually make the device read-only or unusable. Data deduplication has been introduced into SSDs to reduce unnecessary writes. By eliminating redundant writes and retaining only one physical copy of duplicate data, deduplication reduces write volume, shortens program/erase delays, and improves storage efficiency [3]–[7].

Modern SSDs exploit internal parallelism by distributing pages across multiple flash chips. Consecutive pages can be read in parallel when they reside on different chips, whereas requests targeting the same chip are serialized. Therefore, for SSDs, file fragmentation is not caused by mechanical seeks as in HDDs, but by an imbalanced chip-level placement that reduces read parallelism.

Although deduplication saves capacity and reduces erase operations in SSDs, it can also introduce fragmentation, creating an I/O performance bottleneck for flash-based storage [8], [9], especially in mobile devices. In HDDs, deduplication causes fragmentation by replacing identical data blocks with references, making the physical addresses of a file non-contiguous. Because HDDs rely on mechanical head movement, such fragmentation increases seek overhead and degrades read performance. SSDs do not suffer from seek latency, but deduplication can still fragment files and reduce performance. AppDedup, an application-level deduplication study, reports that deduplication can reduce sequential read performance by up to 30% [9]. Jun et al. further explain the SSD-side mechanism: when many pages of a file are served by the same flash chip, chip-level contention serializes page accesses and hurts SSD performance [1]. This contention arises because deduplication maps logical addresses of redundant data to the existing physical locations of duplicate data blocks. The remaining non-duplicate data blocks are then placed by the FTL across chips. As a result, multiple consecutive logical addresses may map to the same chip, preventing SSDs from fully exploiting internal parallelism during reads. The data must instead be accessed sequentially within one chip, reducing read parallelism and increasing latency. As files are updated, this imbalance can become more pronounced.

Several prior systems mitigate deduplication-induced fragmentation. SmartDedup [5] adopts the iDedup approach, which deduplicates only physically contiguous duplicate write sequences [10]. By filtering out short repeated sequences, it preserves data sequentiality. HAR [11] is a restore-oriented deduplication method for HDD-based backup systems; it uses container non-utilization to decide when useful chunks should be rewritten into a new container. FCRC [12] limits the overall deduplication rate, calculates the maximum number of rewrites for each data-segment write operation, and prioritizes high-reference-count data for rewriting. These approaches provide useful insights, but they are mainly designed around HDD seek

behavior or container locality [10]–[13]. They do not fully capture how deduplication changes chip-level data layout in SSDs or how that layout reduces internal parallelism.

To mitigate fragmentation, we propose FAD, a fragmentation-aware data layout strategy for deduplication-based SSDs. By analyzing how file fragmentation forms and how duplicate data blocks are distributed across chips, FAD classifies fragmentation into non-uniformly distributed fragmentation (NUDF) and uniformly distributed fragmentation (UDF). For UDF files, FAD uses a read-parallelism optimization strategy that places new non-duplicate blocks to maximize chip-level parallelism. For NUDF files, FAD uses an elastic-threshold-based duplicate block rewriting strategy. A bottom threshold identifies duplicate blocks that create chip-level conflicts, while a top threshold bounds the number of rewritten blocks. This design improves read parallelism while limiting additional write and capacity overhead. Because read-oriented placement may disturb the cyclic write order, FAD further uses a write-parallelism optimization strategy to refine write scheduling and reduce write latency.

Overall, we make the following key contributions:

(1) We investigate deduplication-induced fragmentation in SSDs through trace-driven experiments, showing that deduplication increases fragmentation compared with non-deduplicated SSDs. We also analyze the characteristics of fragmented files and classify them into NUDF and UDF.

(2) We propose FAD, which monitors fragmentation and applies targeted optimizations. For UDF files, FAD improves read parallelism through placement. For NUDF files, it uses selective duplicate block rewriting to reduce chip conflicts. FAD also introduces write-parallelism optimization to offset the write-side impact of read-oriented placement.

(3) Trace-driven experiments show that FAD reduces average read latency by 15.8%–34.1% and P99 read tail latency by 10.0%–44.6% across the compared baselines. These results show that FAD mitigates fragmentation-induced performance loss in deduplication-based SSDs.

II. Background and Motivation

A. Structure of Flash-based SSDs

Modern SSDs consist mainly of an SSD controller and an array of flash chips. Here, a *data block* denotes the deduplication granularity; in our implementation, one data block is one logical flash page (4 KB in Table I). A *flash page* denotes the NAND read/write unit, whereas a *flash block* denotes the NAND erase unit and contains tens to hundreds of flash pages. The SSD controller performs address mapping, garbage collection, wear leveling, and request scheduling. Address mapping translates the logical address of a data block to the corresponding physical flash page. Because SSDs read and write at flash-page granularity, erase at flash-block granularity, and do not support in-place page updates, they rely on out-of-place updates. When flash pages in a flash block become invalid after overwrites or reference-count drops, garbage collection migrates the remaining valid flash pages to free locations and erases the old flash block. Wear leveling balances erasures across flash blocks because flash memory supports only a limited number of erase cycles. The request scheduler determines the order of read and write operations. Flash chip arrays use a multi-channel shared-bus architecture with one Flash Controller per channel. Multiple flash chips share a channel bus; each chip contains multiple dies, each die contains multiple planes, and each plane contains registers and many flash blocks. SSDs can exploit parallelism across channels, chips, dies, and planes [3], [14]–[19].

Despite this internal parallelism, each chip can process only one request at a time. When multiple read requests access different flash pages in the same chip, the requests are serialized within that chip. We refer to this chip-level read conflict as a read collision. As more requests are serialized on the same chip, their waiting time increases, reducing the benefit of SSD parallelism and increasing access latency [1], [13], [20]. For writes, SSDs typically allocate addresses in round-robin order to maximize write parallelism [21], [22]. The Flash Translation Layer (FTL) therefore selects chips sequentially instead of concentrating writes on one chip.

B. SSD Deduplication

Deduplication eliminates redundant data by dividing input data into fixed-size data blocks and generating a fingerprint for each block with a cryptographic hash function. In this work, the deduplication data-block size is equal to the flash page size used by SSDsim, i.e., 4 KB. The system compares fingerprints to identify duplicates. If a match is found, only one physical copy is stored, and duplicate logical pages are replaced with references to that physical page. This reduces storage usage and write volume. Deduplication-based SSDs can therefore reduce write amplification and extend device lifetime, especially for high-capacity SSDs where frequent writes and erasures are costly [7], [23]–[28].

SSD deduplication has been studied across multiple layers. Early systems such as CAFTL [2] and CA-SSD [3] integrate content-aware mechanisms into the Flash Translation Layer (FTL) to eliminate redundant writes and extend SSD lifetime. These studies implement content-addressable storage (CAS) at the device level instead of relying only on a host-side file system or cache. Because traditional FTL architectures can issue many redundant writes, these systems use specialized mapping mechanisms to map multiple logical addresses to a single physical address.

More recent work extends SSD deduplication in several directions. CAGC [4] addresses performance bottlenecks in ultra-low-latency SSDs with a content-aware GC mechanism. By embedding deduplication into garbage collection, CAGC uses idle GC resources to reduce interference with foreground I/O. SmartDedup [5], Nitro [29], and CacheDedup [26] extend deduplication to broader storage architectures, including SSD-based caching between DRAM-based storage and HDD-based primary storage. Remap-SSD [6] studies secure and efficient SSD address remapping for data deletion, and ARM-Dedup

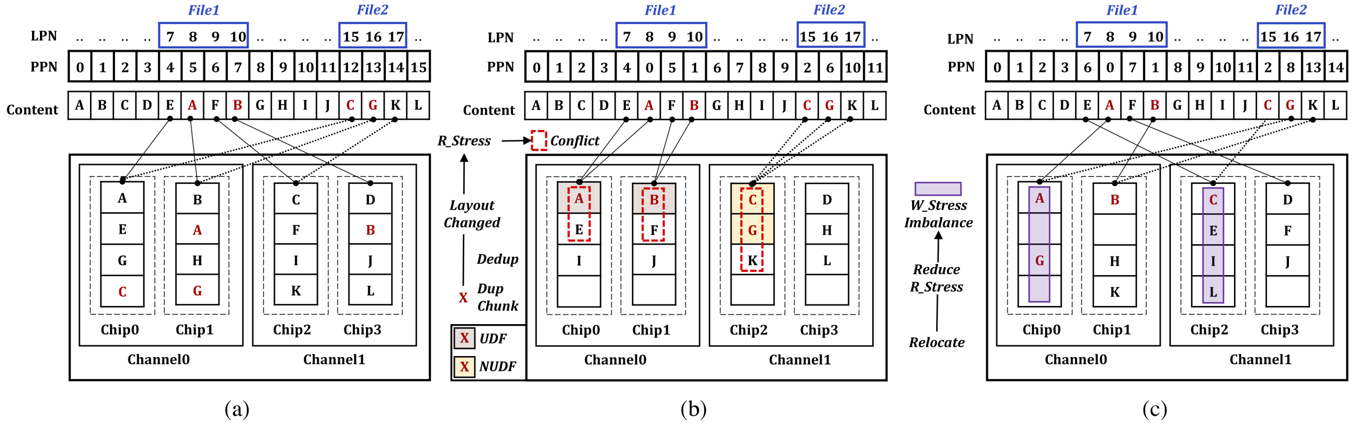

Fig. 1. (a) Data layout before deduplication. (b) Data layout and fragmentation types after deduplication. (c) Ideal data layout.

[7] offloads deduplication management from the host to SSDs to reduce CPU and memory overhead.

C. Fragmentation Problem

In traditional HDD-based storage systems, deduplication can cause file fragmentation. Data blocks are normally stored contiguously to maximize write bandwidth and minimize I/O requests. Deduplication removes redundant blocks, which can make the remaining data blocks non-contiguous. On HDDs, this increases disk-head movement during reads and reduces access speed.

Deduplication also causes fragmentation in SSDs, but the mechanism is different. Unlike HDD fragmentation, which causes mechanical seek overhead, SSD fragmentation reduces the ability to exploit parallel access across multiple chips. Modern SSDs use multiple flash chips in parallel to maximize bandwidth and throughput [21], [30], but each chip can handle only one request at a time [31]. If read requests are concentrated on one chip, they must be processed sequentially. Deduplication disrupts sequential placement by eliminating redundant writes, often concentrating both duplicate and non-duplicate data blocks on the same chip [1]. This prevents full use of SSD parallelism, forcing serialized reads and increasing wait time. As a result, SSD read latency and tail latency increase.

Figure 1 illustrates how deduplication affects SSD internal parallelism. Traditional SSDs use round-robin placement to distribute consecutive writes across different parallel units (chips), mapping each logical page number (LPN) to a physical page number (PPN). As shown in Figure 1(a), sequential writes from LPN7 to LPN10 are assigned to four chips, allowing the corresponding reads to be served in parallel. With deduplication enabled, redundant writes are eliminated, and multiple logical addresses with identical content map to the same physical page. As shown in Figure 1(b), consecutive logical addresses may then map to the same chip. For example, LPN7 and LPN9 can be processed in parallel on different chips, but LPN8 must wait for LPN7 and LPN10 must wait for LPN9. Parallelism is therefore reduced from four chips to two.

Data deduplication can increase fragmentation and degrade storage read performance. To quantify this impact, we compare a native SSD system without deduplication (Native) and an SSD system with deduplication (Deduplication) across nine workloads, using both fragmentation level and normalized average read latency. System settings and workload details are discussed in Section IV-A. Unlike HDDs, fragmentation in SSDs is closely tied to internal parallelism [8], [13]. Let $p_i(X)$ denote the number of pages of file X placed on chip i. A read of X needs

$$r(X) = \max_i p_i(X)$$

chip-level rounds, because the chip with the most pages determines the serialized part of the read. The effective average parallelism is therefore $P(X) = \mathrm{Num}(X)/r(X)$.

The ideal layout minimizes the number of chip-level rounds:

$$r^*(X) = \lceil \mathrm{Num}(X)/N_{\text{chip}} \rceil ,$$

and its ideal effective parallelism is $P^*(X) = \mathrm{Num}(X)/r^*(X)$. We define the degree of fragmentation as

$$DOF(X) = 1 - \frac{P(X)}{P^*(X)}.$$

Thus, $DOF(X) = 0$ means that the file achieves the best possible chip-level parallelism under its size and the number of chips. For example, if a five-page file is placed on four chips as $[2, 1, 1, 1]$, then $r(X) = r^*(X) = 2$ and $DOF(X) = 0$. If all five pages are placed on one chip, then $r(X) = 5$, $P(X) = 1$, $P^*(X) = 2.5$, and $DOF(X) = 0.6$. In our experiment, we use the chip as the parallel unit for fragmentation computation, though the same method can be applied to channels, dies, or planes.

As shown in Figure 2, Deduplication increases the DOF by 47.7% on average compared with Native, with the mds1 workload showing a 76% increase. This result shows that deduplication reduces storage redundancy but disrupts data

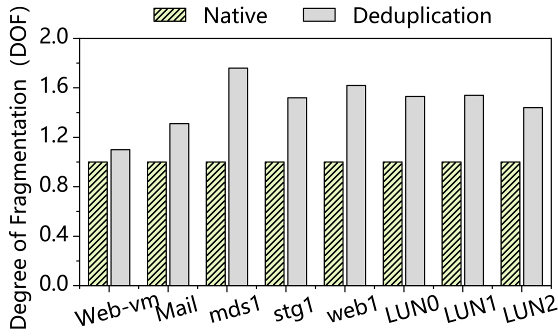

Fig. 2. Normalized fragmentation level (DOF) comparison between Native and Deduplication (normalized to Native).

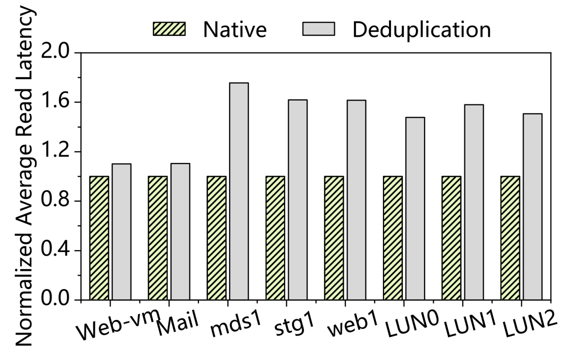

Fig. 3. Normalized average read latency comparison between Native and Deduplication (normalized to Native).

distribution by concentrating duplicate and non-duplicate data blocks on the same chip, limiting multi-chip parallelism. Figure 3 shows the corresponding effect on read performance. Compared with Native, Deduplication increases average read latency by 47%, with the largest increase, 75.6%, on mds1. These results indicate that deduplication-induced fragmentation reduces read parallelism, increases normalized DOF, and degrades SSD read performance.

Three strategies are commonly used to mitigate deduplication-induced fragmentation: rewriting, caching, and data placement. Rewriting reduces fragmentation by selectively retaining additional physical copies of duplicate data, but it consumes capacity and lowers the deduplication ratio. For example, FCRC [12] dynamically adjusts capping thresholds according to data block distribution and uses Container Reference Count (CNRC) to select blocks for rewriting. HAR [11] improves restore performance in deduplicated backup storage by classifying fragmented containers and limiting rewrite overhead. These schemes are useful baselines, but their decisions are not designed around SSD chip-level parallelism. Caching improves access efficiency by storing frequently accessed data in SSD or memory [8], but it adds storage and management cost. Data placement reduces fragmentation by optimizing physical locations, improving read performance by better exploiting SSD parallelism.

D. Observation and Motivation

Most deduplication algorithms that address file fragmentation rely on the reference count of data blocks. A higher reference count means that a physical data block is shared by more live logical pages, possibly across multiple files, so rewriting it can potentially benefit more future reads. However, reference count alone does not capture the chip locations of those shared blocks. These methods therefore have limited ability to address SSD fragmentation, because they focus on sharing degree without fully considering the post-deduplication chip-level layout.

To understand how fragmentation affects SSD reads, we analyze file-block distribution after deduplication and compare it with the ideal read layout. As shown in Figure 1(b), the write request LPN7–LPN10 corresponds to a file with two duplicate blocks, A and B. Deduplication does not write new physical copies of A and B; instead, the corresponding logical pages point to existing physical pages. When the file is read, A and B are stored on different chips and can be read in one parallel round. Blocks E and F must wait because their current placement conflicts with those chips. In the ideal layout in Figure 1(c), E and F are placed on Chip2 and Chip3, allowing all blocks to be read in parallel. For LPN15–LPN17, duplicate blocks C and G are stored on the same chip, so they require two serialized read rounds and force block K to wait. The ideal layout places G on Chip0 and K on Chip1, enabling parallel reads without waiting.

In the ideal layouts of files 1 and 2, blocks E and F of file 1 can be written while avoiding the chip addresses of duplicate blocks A and B. For file 2, block K can also be placed without conflicting with duplicate block addresses. However, block G is already stored in the SSD. Creating a better layout for the current file therefore requires additional rewriting, which introduces overhead and may affect capacity. We call a file uniformly distributed fragmented (UDF) if its existing duplicate blocks are not over-concentrated on any chip. In this case, an ideal or near-ideal chip-level layout can be reached by placing only the new non-duplicate blocks. Formally, this holds when no chip already stores more duplicate blocks of the file than the per-file bottom threshold $N_{\mathrm{f}}(X)$. As shown in Figure 1(b), file 1 is UDF because Chip0 and Chip1 each hold only one duplicate block, leaving enough room to place E and F on other chips. By contrast, non-uniformly distributed fragmentation (NUDF) occurs when at least one chip already stores too many duplicate blocks of the same file, i.e., more than $N_{\mathrm{f}}(X)$. In this case, placing only new blocks cannot remove the serialized reads caused by the existing duplicate-block concentration; selective rewriting is needed. In Figure 1(b), file 2 is NUDF because Chip2 contains two duplicate blocks, C and G, from the same file.

Based on the discussion above, we identify three challenges:

For UDF, a placement-only approach can improve layout without capacity overhead or additional rewriting. The challenge is to identify the fragmentation type of each file and track

the chip locations of its duplicate data blocks. This requires analyzing data redundancy, physical locations, and their impact on read parallelism.

For NUDF, placement alone is insufficient, so duplicate block rewriting must be combined with layout optimization. However, rewriting introduces additional physical copies compared with a deduplication-only design. The system must therefore balance read-performance benefits against capacity and write overhead.

Data layout also affects write performance. Traditional SSDs use a circular write strategy that writes data to different chips in logical write order, keeping write requests balanced across chips. Fragmentation-aware placement may skip some chips to avoid chip-level read conflicts, which can disrupt this balance and reduce write parallelism. For example, in Figure 1(c), block E would normally be written to chip 0 after block D. Because chip 0 already contains duplicate block A from the same file, block E skips chip 0 and tries chip 1. Chip 1 already stores duplicate block B, so block E skips chip 1 and is eventually written to chip 2. Such skipping reduces writes to chips 0 and 1 while increasing writes to chip 2, weakening write load balance.

III. Design of FAD

A. System Overview

FAD is designed to mitigate fragmentation caused by data deduplication in SSDs. Its goals are to improve read parallelism and reduce tail latency while preserving write performance and limiting capacity overhead. To achieve these goals, FAD combines selective duplicate block rewriting with fragmentation-aware data placement. Because read-oriented placement can disrupt the cyclic write rule and reduce write parallelism, FAD also includes a compensation mechanism for write scheduling.

As shown in Figure 4, FAD constructs a Redundant Data Localization Table (RDLT) during the write path to track the chip-level distribution of duplicate data blocks. It then classifies each file as UDF or NUDF. For UDF files, FAD applies layout optimization. For NUDF files, it further applies reference-count-aware duplicate block rewriting. FAD also uses a Write Skip Tracking List (WSTL) and a circular-pointer scheduling mechanism to compensate for chips skipped by read-oriented placement. The architecture consists of four modules: fragmentation type judgment, duplicate block rewriting, read parallelism optimization, and write parallelism optimization.

FAD write path. For clarity, we summarize the FAD write path as follows. (1) For each incoming file write, FAD computes fingerprints and identifies duplicate blocks through the deduplication mapping. (2) FAD builds the RDLT by counting the chip locations of existing duplicate blocks and computes $N_{\mathrm{f}}(X)$. (3) If all RDLT counters are no larger than $N_{\mathrm{f}}(X)$, the file is classified as UDF; otherwise, it is classified as NUDF. (4) For UDF files, FAD adjusts the placement of new non-duplicate blocks without rewriting existing duplicate blocks.

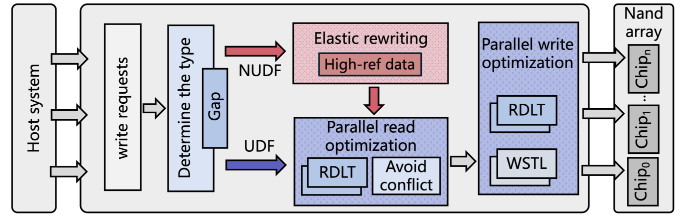

Fig. 4. System structure of FAD.

For NUDF files, FAD first selects a bounded number of duplicate blocks for rewriting and then applies the same placement rule. (5) Whenever a chip is skipped by the read-oriented placement rule, FAD records it in WSTL and prioritizes it in subsequent writes to restore round-robin write balance. The following subsections introduce these steps in order.

B. Data Structures

To support fragmentation-aware placement, FAD introduces two data structures: the Redundant Data Localization Table (RDLT) and the Write Skip Tracking List (WSTL).

RDLT is initialized at the beginning of each file write request and deallocated after address assignment finishes for that file. It is a temporary one-dimensional array with $N_{\text{chip}}+1$ entries. The first N_{chip} entries are per-chip counters for the current file, and the final entry stores the bottom threshold $N_{\mathrm{f}}(X)$. The counters are initialized from duplicate blocks found by fingerprint lookup and updated as new or rewritten blocks are assigned. Thus, RDLT is not persistent per-file metadata; it is short-lived $O(N_{\text{chip}})$ state used only while placing the current file. During duplicate block rewriting, RDLT identifies chips whose duplicate-block counters exceed the threshold. During read-parallel placement, it prevents new non-duplicate blocks from being allocated to overloaded chips, reducing chip-level read conflicts and improving read parallelism.

Each WSTL node records a skipped chip address and a pointer to the next node. When the placement rule skips a chip because writing to it would exceed $N_{\mathrm{f}}(X)$, the chip address is appended to WSTL. During subsequent writes, FAD first checks the WSTL head and attempts to backfill that chip. Backfill is allowed only when the RDLT counter after the write would not exceed $N_{\mathrm{f}}(X)$. For rewritten blocks, the new chip must also differ from the old chip. After a successful backfill, the node is removed. Thus, WSTL records temporary scheduling debt caused by read-oriented placement and helps restore round-robin write balance.

C. Fragmentation Type Identification

Let $d_i(X)$ denote the number of duplicate blocks of file X that are already stored on chip i when the file write is processed. FAD classifies the file by the maximum duplicate-block concentration across chips. If $\max_i d_i(X) \leq N_{\mathrm{f}}(X)$, the file is UDF: existing duplicate blocks are not over-concentrated on any chip, so placing only the new non-duplicate blocks can reach or approach the ideal chip-level layout. If $\max_i d_i(X) > N_{\mathrm{f}}(X)$, the file is NUDF: at least one chip already contains

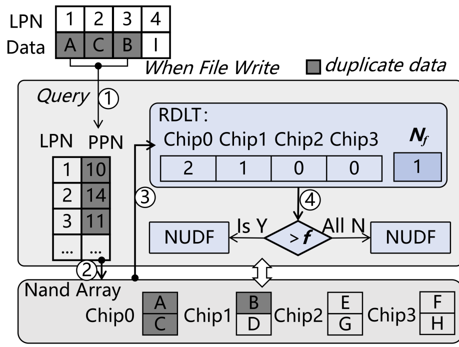

Fig. 5. Fragmentation classification.

too many duplicate blocks of the same file, so new-block placement alone cannot remove serialized reads and selective rewriting is needed.

To determine the fragmentation type, FAD creates an RDLT for each file write request, identifies duplicate data blocks that will be replaced by references, and records the chip addresses of their existing physical pages. The RDLT counters are then compared with $N_{\mathrm{f}}(X)$. This judgment guides subsequent data layout optimization and selective rewriting.

Figure 5 shows an example of fragmentation classification. The current file contains data blocks with LPNs from LPN1 to LPN4 and is being written to the SSD. The data blocks shown in the flash array are existing blocks, not blocks newly written for the current file. Data block A for LPN1, data block C for LPN2, and data block B for LPN3 are duplicate blocks. The classification process has three steps: (1) Obtain the chip locations of duplicate blocks and update the RDLT. FAD traverses the blocks of the current file and queries the address mapping table for each block. If a physical address is found, the block is identified as a duplicate block and its chip location is recorded in the RDLT. For the example in Figure 5, blocks A, C, and B are duplicate blocks located on Chip 0 and Chip 1. Accordingly, the corresponding entries in the RDLT are updated to reflect that Chip 0 stores two duplicate blocks and Chip 1 stores one duplicate block. If no matching physical address is found, the block is treated as a non-duplicate block (e.g., block I).

(2) Calculate the bottom threshold $N_{\mathrm{f}}(X)$: This threshold represents the ideal number of blocks from the same file that should be stored per chip. It is calculated as follows:

$$N_{\mathrm{f}}(X) = \left\lceil \frac{\mathrm{Num}(X)}{\min(\mathrm{Num}(X), N_{\mathrm{chip}})} \right\rceil \tag{1}$$

where $Num(X)$ is the total number of data blocks in file X, and N_{chip} is the total number of chips in the SSD. The ceiling makes the threshold an integer capacity per chip. For example, if $Num(X) = 5$ and $N_{\mathrm{chip}} = 4$, the ideal layout needs two read rounds and allows at most $N_{\mathrm{f}}(X) = 2$ blocks on any chip. If the number of duplicate blocks already stored on one chip exceeds $N_{\mathrm{f}}(X)$, the file may still suffer from read delays even when its new blocks are ideally placed. This happens because duplicate data blocks on the same chip must be accessed sequentially, increasing that chip's service time relative to other chips. As a result, SSD parallelism is constrained and read performance degrades.

(3) Determine the fragmentation type. At this stage, all chip counters in the RDLT are compared to the bottom threshold. If any counter exceeds the threshold, the file is classified as NUDF. Otherwise, it is classified as UDF.

D. *Elastic-Threshold-based Duplicate Block Rewriting*

As discussed in II-D, handling NUDF is important for SSD read parallelism. When a file read request arrives, many data blocks may need to be processed serially on the same chip. This increases I/O latency and reduces system throughput. To address this issue, we propose an elastic-threshold-based duplicate-block rewriting strategy. FAD dynamically computes a bottom threshold according to the number of data blocks in each file. During file writing, if the number of duplicate data blocks on a chip exceeds this threshold, those blocks become rewriting candidates. To balance deduplication and performance, FAD also sets a top threshold. This threshold limits how many data blocks can be rewritten for one file, bounding additional writes and capacity use while reducing fragmentation.

The formula for the bottom threshold is given in Eq. (1). When the number of duplicate blocks on a chip exceeds this threshold, the duplicate blocks on that chip are listed as candidates for rewriting. For each candidate block b, FAD uses its reference-count-based heat value $H(b)$. This value is the reference count maintained by the deduplication mapping, i.e., the number of live logical pages, possibly from multiple files, that currently reference the same physical page. The count increases when a new duplicate reference is created and decreases when a file is overwritten or deleted. We use the reference count as a compact sharing indicator rather than a direct measurement of temporal read frequency. Heat is therefore a proxy for how many future file reads may benefit from rewriting a physical block; it does not assume that every referenced file is read with the same frequency. Candidate blocks are sorted in descending order of $H(b)$. As shown in Figure 5, the file contains four data blocks (LPN1–LPN4). Using Eq. (1), the bottom threshold $N_{\mathrm{f}}(X)$ is 1, meaning that ideally each chip should store one block of the file. Chip 0 stores two duplicate data blocks, which exceeds the threshold, so blocks on Chip 0 become rewriting candidates.

The policy also needs to bound the number of blocks rewritten in each operation. FAD uses the top threshold $N_{\mathrm{top}}(X)$ for this purpose:

$$N_{\mathrm{top}}(X) = \lfloor \rho \times \mathrm{Num}(X) \rfloor \tag{2}$$

where ρ is the allowable rewrite ratio, i.e., the proportion of additional physical writes that the system is willing to accept for a file. We set ρ to 30%. $Num(X)$ is the total number of blocks to be written in file X. In the example shown in Fig. 5, if the file contains four data blocks, the top threshold $N_{\mathrm{top}}(X)$ is 1, meaning that at most one data block can be

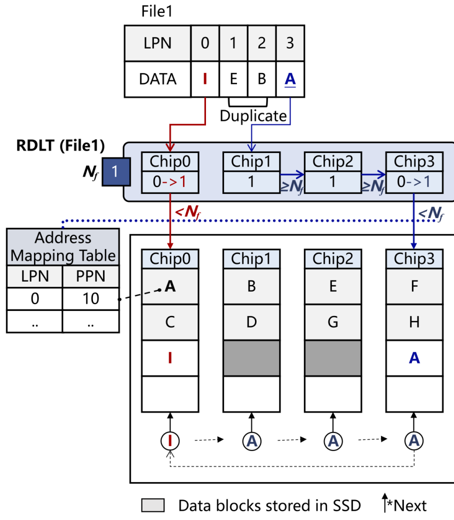

Fig. 6. Read parallelism optimization strategy.

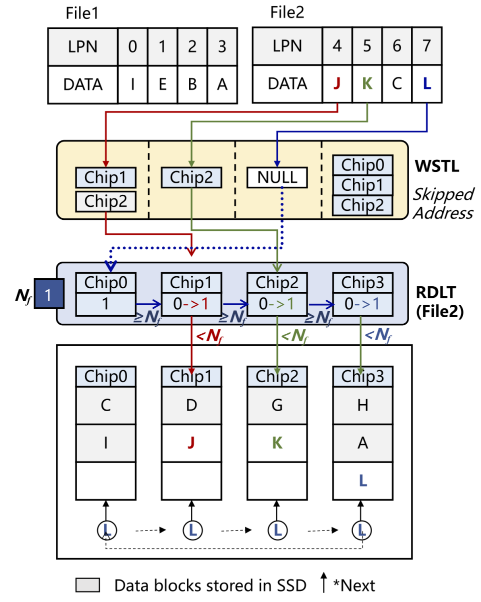

Fig. 7. Write parallelism optimization strategy.

rewritten. Therefore, on Chip 0, one candidate block is selected for rewriting. The candidate with the higher reference-count-based heat is rewritten first because it can reduce chip conflicts for more logical references. In this example, data block A is selected because it has the higher heat value. The RDLT is then updated accordingly.

It is important to note that rewriting a duplicate block in FAD does not migrate the original physical page or remap other files that already reference it. Instead, FAD materializes an additional physical copy for the current file and updates only the current file's logical-to-physical mapping. Existing files continue to reference the old physical page. Therefore, rewriting cannot directly worsen the layout of previously written files. Its side effects are limited to additional physical writes, extra capacity use, and possible future GC pressure, all bounded by the top threshold $N_{\text{top}}(X)$.

E. Read Parallelism Optimization Strategy

For the fragmented files identified in Section II-D, FAD improves read parallelism by controlling the chip selected for each new or rewritten block. Traditional round-robin placement writes data blocks to chips in a fixed cyclic order. This preserves write balance but does not consider where duplicate blocks of the same file already reside. FAD keeps the cyclic pointer, but it consults the RDLT before accepting the pointed chip. This prevents new blocks from further concentrating a file on a small number of chips.

The placement rule first queries the RDLT for the chip pointed to by the *next* pointer. If writing the current block would keep the chip counter at or below $N_{\text{f}}(X)$, the chip is accepted. Otherwise, the chip is skipped and recorded in WSTL. For a rewritten block, FAD also verifies that the new physical address is on a different chip from the old physical address; otherwise, rewriting would not reduce the read conflict. Once a suitable chip is found, the data block is written, the RDLT counter is updated, and the *next* pointer advances.

In Figure 6, the file contains four data blocks (LPN0 to LPN3), corresponding to blocks I, E, B, and A on physical chips. Blocks E, B, and A are duplicates, and block A requires rewriting. Initially, the temporary RDLT records the number of duplicate data blocks stored on each chip. Because the bottom threshold is 1, each chip should store at most one block of the file. The system uses the *next* pointer to determine the write order and assign blocks to appropriate chips.

First, data block I is written. FAD queries the RDLT and finds that chip 0 has zero data blocks, which is below the bottom threshold. Data block I is therefore written to chip 0. After the write, the counter for chip 0 is incremented by 1, and the *next* pointer moves to chip 1. FAD then processes data blocks E and B. Because both are duplicate blocks rather than rewritten blocks, the logical mapping table is updated to reference their existing physical pages. Finally, FAD processes data block A, which is rewritten and requires careful placement. The *next* pointer initially points to chip 1. Since chip 1 has already reached the bottom threshold, FAD skips chip 1 and moves the pointer to chip 2. Chip 2 also has a counter value of 1, so it is skipped as well. The pointer then advances to chip 3, whose counter is 0. Before writing A, FAD checks A's old address, which is on chip 0. Because the new chip differs from the old one, A is written to chip 3. The RDLT counter for chip 3 is then incremented, and the *next* pointer advances.

F. Write Parallelism Optimization Strategy

As discussed in II-D, data layout optimization can improve read parallelism but may affect writes. In the traditional cyclic write strategy, data blocks are distributed evenly across chips to maintain high write parallelism. Read-oriented placement

changes this order and may cause some chips to receive multiple blocks in quick succession while others remain idle. This imbalance can reduce write parallelism and degrade write performance.

To address this issue, FAD uses WSTL to dynamically adjust the write order and rebalance data distribution. First, the system checks WSTL. If the list is not empty, the chip address at the head of the list is prioritized for writing. If the list is empty, writes follow the default order. Next, the system consults the temporary RDLT to assess the chip before making a write decision. If the RDLT counter of the chip is at or below the bottom threshold, the data block is written, and the WSTL is updated accordingly. If the RDLT counter would exceed the threshold, the chip is skipped, and its address is added to the end of the list so it can be reconsidered for future writes.

Figure 7 illustrates the WSTL list, which tracks chips skipped by data layout optimization. Suppose chips 1 and 2 were previously skipped, and the temporary RDLT indicates that they currently store zero data blocks.

When writing data block J, FAD first queries WSTL and retrieves the next skipped chip, chip 1. It then checks the RDLT and finds that chip 1 currently stores zero data blocks, so J is written to chip 1. After the write, FAD increments the counter for chip 1 and removes the WSTL head node. The same process is then applied to data block K: WSTL returns chip 2, the RDLT check succeeds, K is written to chip 2, and the corresponding WSTL node is removed. When writing data block L, WSTL is empty, so FAD follows the default write order and checks chip 0 first. Because the RDLT counter for chip 0 exceeds the threshold, FAD skips it to avoid further chip-level concentration for the current file and appends its address to WSTL. Chips 1 and 2 are also skipped and recorded for the same reason. Finally, chip 3 has a counter value of zero, so L is written to chip 3 and the RDLT is updated. After all data blocks have been written, the temporary RDLT is cleared.

G. Garbage Collection

Garbage Collection (GC) in SSDs reclaims space by selecting victim blocks, migrating their valid pages, and erasing blocks with invalid pages. This prevents performance degradation and helps extend device lifetime. If all chips perform GC simultaneously, foreground I/O requests may be delayed. FAD reuses WSTL as a lightweight hint for GC chip selection, while victim-block selection follows the same greedy policy as the baseline. When valid data pages are migrated during GC, they are written back within the same chip after erasure, so GC does not perform an additional cross-chip layout optimization. GC is triggered when the number of free flash blocks falls below a threshold. The process first scans WSTL to prefer chips with more skipped entries when possible, using the list only as a scheduling hint. A greedy algorithm then selects flash blocks with more invalid data, freeing space while reducing valid-page migrations. After erasure, the remaining valid pages are migrated within the same chip. Finally, WSTL is updated for subsequent GC decisions.

TABLE I
SSDSIM CONFIGURATION PARAMETERS.

Parameters	Value	Parameters	Value
Total Capacity	80 GB	Reserved Space Ratio	20%
Channels	8	Chips per Channel	2
Dies per Chip	1	Planes per Die	10
Blocks per Plane	2048	Pages per Block	64
Page Size	4 KB	Erase Latency	1.5 ms
Read Latency	20 μs	Write Latency	200 μs

TABLE II
TRACE WORKLOAD CHARACTERISTICS.

Trace	Read Ratio	Avg. Read Size	Total I/Os (K)	Avg. Req. Size	Dedup. Rate
Homes	19.5%	24.5 KB	3268	21.4 KB	33.3%
Mail	21.5%	30.5 KB	1478	52 KB	91.0%
Web-vm	30.2%	55.4 KB	1778	32.1 KB	47.3%
mds1	92.9%	63.5 KB	1548	60.1 KB	50.0%
stg1	53.8%	91.3 KB	1698	52.8 KB	50.0%
web1	48.3%	61.5 KB	135	34.7 KB	50.0%
LUN0	75.6%	35.4 KB	2317	33.8 KB	50.0%
LUN1	81.1%	46.1 KB	1647	42.6 KB	50.0%
LUN2	69.0%	25.6 KB	2745	22.6 KB	50.0%

IV. PERFORMANCE EVALUATION

A. Experimental Setup

We evaluate FAD by integrating it into SSDsim, an open-source event-driven SSD simulator developed by Microsoft Research. SSDsim models key SSD mechanisms, including reads, writes, garbage collection, and wear leveling. Table I lists the simulated SSD parameters. These parameters follow the multi-chip SSDsim configuration used in prior SSD parallelism studies [14], [31]. They provide a representative multi-chip SSD configuration rather than a model of a specific commercial device. Absolute latency values may vary with device parameters such as chip count and page latency. However, all compared systems use the same SSD configuration. In addition, FAD makes placement decisions based on relative chip-level distribution and recomputes $N_{\mathrm{f}}(X)$ from N_{chip}. Changing the number of chips therefore mainly changes the absolute degree of available parallelism, while comparisons among schemes remain controlled under the same device configuration.

We evaluate FAD from two perspectives: end-to-end comparison against representative baselines and ablation analysis of its internal components. For end-to-end comparison, we consider five systems: Native, Deduplication, HAR, FCRC, and FAD. For ablation analysis, we further evaluate three FAD variants: FAD-RPO, FAD-RWPO, and FAD-RWPO-Rewrite. Native is the original SSD without deduplication. It serves as the default SSDsim design and is mainly used for the motivation experiments. Deduplication extends the original SSDsim design with a hash-based deduplication engine similar to CAFTL [2]. The engine determines whether to write data by computing the fingerprint of each logical data page, with a computational overhead of 32 μs [2], [6]. HAR [11] is

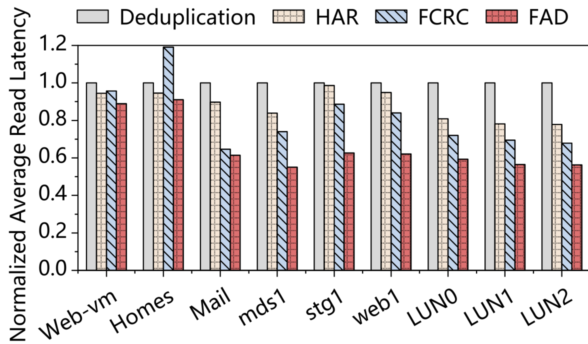

Fig. 8. Normalized average read latency for different systems.

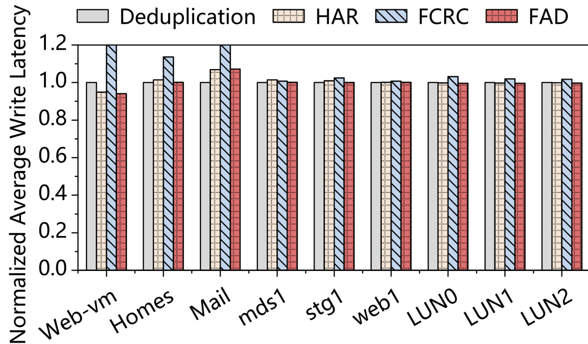

Fig. 9. Normalized average write latency for different systems.

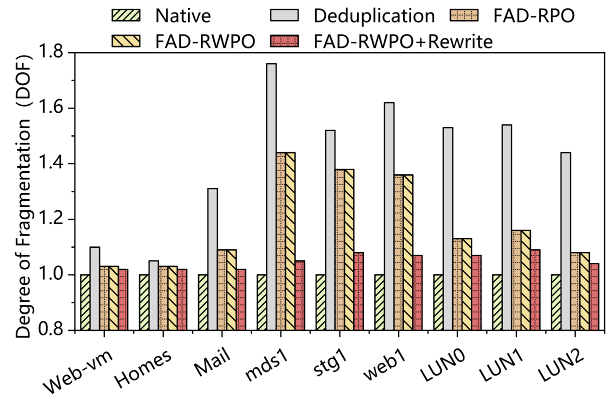

Fig. 10. Normalized fragmentation level (DOF) for FAD variants.

an HDD-oriented deduplication defragmentation scheme that uses container non-utilization as its rewrite trigger. We include HAR not as an SSD-optimized design, but as a representative rewrite-based deduplication defragmentation baseline to test whether generic rewrite policies are sufficient for SSD chip-level conflicts. In SSDsim, we treat a page group as the counterpart of a HAR container: container non-utilization is mapped to the fraction of pages in that group that are not used by the current read stream. We set the non-utilization threshold to 50%, so valid data blocks are rewritten when a read would otherwise fetch too much unused data. FCRC [12] represents a limited-deduplication/rewrite tradeoff baseline: it caps the overall deduplication rate (set to 30%), prioritizes high-reference-count data for rewriting, and helps test whether chip-aware rewriting provides benefits beyond a generic deduplication-limit policy. FAD is the proposed fragmentation-aware data layout strategy. For ablation, we evaluate three progressive variants: FAD-RPO adds read parallelism optimization to measure the benefit of layout-aware placement; FAD-RWPO adds write parallelism optimization to compensate for write-performance loss; and FAD-RWPO-Rewrite adds duplicate block rewriting to handle non-uniform fragmentation.

We use nine traces from three representative sources: Homes, Mail, and Web-vm from the FIU trace collection [25]; mds1, stg1, and web1 from the MSR block I/O dataset [8]; and LUN0, LUN1, and LUN2 from the Enterprise Virtual Desktop Infrastructure (VDI) dataset [32]. The LUN traces correspond to additive-03-2016021612-LUN0, LUN1, and LUN2. Except for Homes, Mail, and Web-vm, which contain actual data, the traces provide only request arrival times, sizes, and locations. We simulate logical data page content using the Zipf distribution to model content popularity [3], [6]. The distribution is defined as $P(t_i) = C/t_i^a$, where $C = 1/\sum_{i=1}^{N} t_i^{-a}$, N is the number of unique contents, and a is the Zipf parameter representing popularity skewness. For traces without actual content, we set a to 0.2 and generate contents with a 50% data deduplication rate (where N is 50% of the total logical pages). Because SSDsim is deterministic under a fixed trace, configuration, and content-generation seed, repeated replays produce identical latency and write-volume results. We therefore omit confidence intervals and report trace-level results for all compared schemes. Detailed trace specifications are listed in Table II.

B. *Performance Analysis*

We first compare FAD with Deduplication, HAR, and FCRC, and then evaluate the contribution of each FAD component using three variants: FAD-RPO, FAD-RWPO, and FAD-RWPO-Rewrite.

Figure 8 shows the normalized average read latency of different schemes. Compared with Deduplication, HAR, and FCRC, FAD reduces the average read latency by 34.1%, 22.2%, and 15.8%, respectively. This improvement comes from FAD's ability to track the chip locations of duplicate data blocks and control the placement of newly written blocks according to the chip-level distribution of each file. In addition, for NUDF files, FAD selectively rewrites high-reference-count duplicate blocks to reduce chip conflicts and improve read parallelism. The improvement is particularly evident for workloads with a high read ratio (e.g., mds1) or a high deduplication rate (e.g., Mail).

Figure 9 shows the normalized average write latency of different schemes. Compared with Deduplication, FAD increases the average write latency by only 0.1%, indicating that its additional write-side overhead is small. Compared with HAR and FCRC, FAD reduces the average write latency by 0.6% and 93.4%, respectively. These results show that FAD

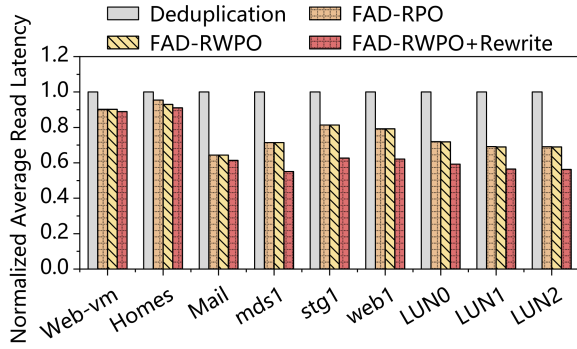

Fig. 11. Normalized read latency for different strategies of FAD.

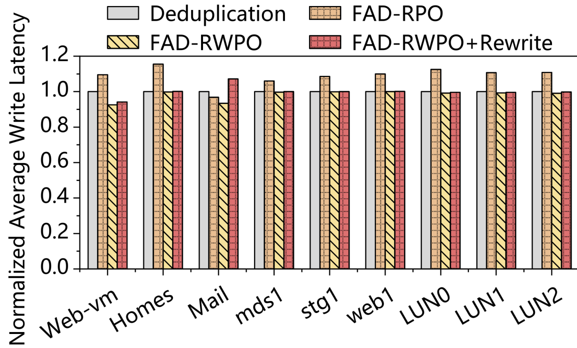

Fig. 12. Normalized write latency for different strategies of FAD.

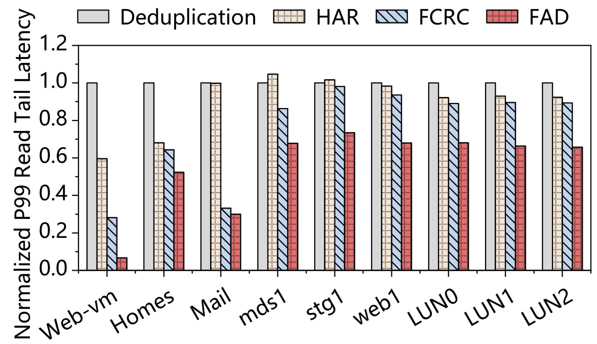

Fig. 13. P99 read tail latency.

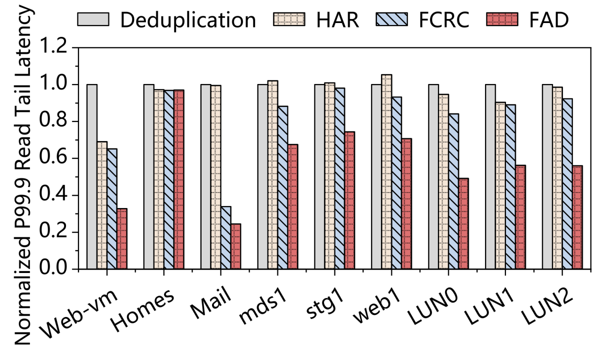

Fig. 14. P99.9 read tail latency.

preserves write parallelism through write-parallelism optimization while avoiding the excessive rewrite overhead of rewrite-intensive schemes. Homes and Web-vm cover relatively write-heavy and lower-deduplication settings in our workload set. When NUDF opportunities are limited, FAD mainly falls back to lightweight placement and WSTL scheduling rather than aggressive rewriting. This explains why Homes and Web-vm show smaller read gains, while the average write-latency increase over Deduplication remains only 0.1%.

Figure 10 reports the normalized DOF reduction from FAD's internal modules. Compared with Deduplication, FAD-RPO reduces normalized DOF by 23% on average and by up to 38%, showing that placement alone mitigates many chip-level conflicts. FAD-RWPO-Rewrite further reduces normalized DOF by 30% on average and by up to 48%, because NUDF files benefit from selective rewriting of high-reference-count duplicate blocks. The trace-level normalized DOF and ablation trends indicate that workloads with higher deduplication or read intensity, such as Mail and mds1, contain more high-conflict duplicate layouts and therefore benefit more from rewriting, whereas Homes and Web-vm contain fewer such cases.

To measure the contribution of each optimization strategy, we evaluate read latency under each FAD variant. The results are shown in Fig. 11. Compared with Deduplication, FAD-RPO reduces read latency by 23.1% on average across the nine traces, with the largest reduction on Mail (35%). FAD-RWPO-Rewrite further reduces read latency by 34.1% on average, with the largest reduction on mds1 (44.9%). Compared with FAD-RPO, FAD-RWPO does not provide additional read-performance improvement, because it is primarily designed to restore write parallelism rather than further optimize read latency. We further examine the impact of FAD modules on write performance, as shown in Fig. 12. FAD-RPO improves read performance by dynamically adjusting data placement. However, this read-oriented placement reduces write parallelism, increasing write latency by 8.9% on average. FAD-RWPO uses WSTL to recover chips skipped by read-oriented placement. During writes, it considers the skipped-chip state, chip-level conflicts for the current file, and whether the block is rewritten. By adjusting the write order, FAD-RWPO distributes write requests more evenly across chips and mitigates the write-parallelism loss introduced by FAD-RPO. Compared with FAD-RPO, FAD-RWPO reduces write latency by 11% on average.

C. Tail Latency Analysis

Tail latency captures the slowest request responses in a storage system. Even when most requests complete quickly, high tail latency can degrade quality of service. Therefore, both low average latency and low tail latency are important for evaluating storage systems.

Figures 13 and 14 compare the P99 and P99.9 read tail latency of different schemes. Compared with Deduplication, HAR, and FCRC, FAD reduces the P99 read tail latency by 10.0%, 25.4%, and 44.6%, respectively. Similarly, for P99.9

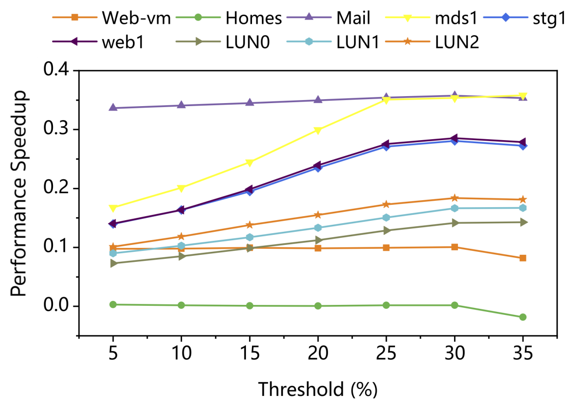

Fig. 15. Speedup of FAD under different duplicate-block rewrite thresholds.

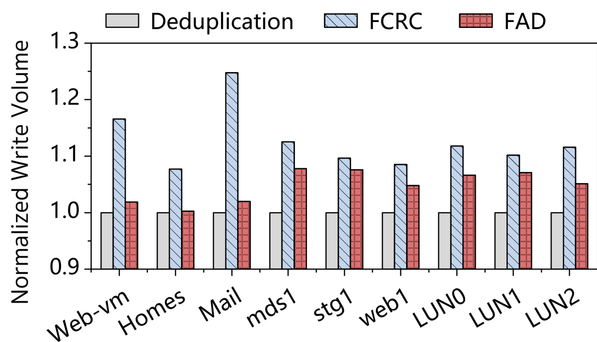

Fig. 16. Normalized write volume comparison.

read tail latency, FAD achieves average reductions of 41.3%, 36.0%, and 23.6%, respectively. These results show that FAD improves both average and tail read latency by mitigating chip-level conflicts through layout optimization and selective rewriting.

D. Sensitivity Analysis

FAD performance depends on several factors, among which the duplicate-block rewrite threshold is particularly important. Different threshold settings determine how often duplicate blocks are rewritten, directly affecting conflict mitigation and read/write parallelism.

Figure 15 shows the speedup of FAD across nine workloads under duplicate-block rewrite thresholds from 5% to 35%, compared with the baseline Deduplication system. The low-threshold results show that read and write parallelism optimization can still improve performance when rewriting is limited. As the threshold increases from 5% to 35%, the average speedups across the nine workloads are 12.8%, 14.2%, 16.0%, 18.1%, 20.1%, 20.8%, and 20.2%, respectively. Performance generally improves as the threshold increases from 5% to 30%. At lower thresholds, too few blocks are rewritten to sufficiently reduce fragmentation. As the threshold increases, FAD rewrites more high-reference-count and highly conflicted data blocks, improving the layout and read parallelism. However, the marginal gain decreases after 25%, and some workloads degrade at 35%. An excessively high threshold introduces more physical writes, consumes write bandwidth, creates additional invalid pages, and can increase future GC pressure. We therefore use 30% as the default threshold because it provides the best compromise across most workloads while bounding write overhead. The write-volume impact of the default 30% threshold is quantified in Fig. 16. Under this threshold, FAD increases total written data by only 4.7% on average compared with Deduplication.

E. Cost Analysis

In the elastic-threshold-based duplicate block rewriting strategy, FAD selectively rewrites duplicate blocks in NUDF files, which introduces additional physical writes. We quantify both the additional write volume caused by selective rewriting and the metadata/processing overhead introduced by RDLT and WSTL.

We analyze the additional data written by FAD due to rewriting and evaluate total write volume across nine workloads under Deduplication, FCRC, and FAD. FCRC is chosen as the benchmark for this write-volume comparison because it provides a threshold-based rewrite mechanism comparable to FAD. HAR is included in the latency experiments, but its container non-utilization trigger is not parameterized by the same per-file rewrite threshold. We therefore omit HAR from this threshold-specific write-volume plot for comparability, not because of its absolute performance. Figure 16 presents the normalized write-volume comparison. Compared with Deduplication, FAD increases total written data by only 4.7% on average, with workload-specific increases ranging from 0.3% to 7.7%. This overhead comes from selective duplicate-block rewriting and is bounded by $N_{\text{top}}(X)$. Compared with FCRC, FAD further reduces write overhead by 7.8%, indicating that chip-aware rewriting avoids unnecessary rewritten copies.

FAD does not maintain persistent per-file layout metadata. RDLT is temporary per-active-write state: it is allocated while processing the current file write and released after address assignment. On a 16-chip SSD, a single RDLT occupies approximately 64 bytes. Its memory usage is therefore $O(N_{\text{chip}})$ per active write and scales with concurrent writes rather than with the total number of stored files. The aggregate RDLT memory overhead remains limited because each RDLT is temporary and scales only with N_{chip}. For workloads with many small files, file size affects the number of fingerprint lookups and placement decisions, but not the RDLT size. WSTL records temporary skipped-chip scheduling debt caused by read-oriented placement. Each WSTL node occupies 12 bytes. The total WSTL overhead is bounded by the number of skipped-chip entries maintained by the scheduler and remains small in our experiments. Entries are consumed by subsequent writes and removed after successful backfill. If a strict memory limit is required, the system can safely drop old WSTL entries and fall back to default round-robin placement. This may reduce optimization opportunities but does not affect correctness. The processing cost is also bounded: for a file with

$Num(X)$ data blocks, FAD performs one fingerprint lookup pass, $O(N_{\text{chip}})$ per-file classification over RDLT counters, candidate sorting only among duplicate blocks on overloaded chips, and in-memory placement checks using the circular pointer and WSTL. Therefore, its additional CPU work is small compared with hash computation and flash access latency.

V. Related Work

Fragmentation in deduplication systems has been widely studied. Deduplication replaces repeated data with references to existing physical copies, saving capacity but potentially breaking the physical locality of a file. In HDD-based backup systems, the main symptom is additional seeks and fragmented container reads. In SSDs, the symptom is different: a file may lose chip-level read parallelism when too many logical pages map to physical pages on the same chip. Existing studies address deduplication-induced fragmentation mainly through rewriting, caching, and SSD-internal deduplication or layout techniques.

Foundational deduplication systems also exploit stream and chunk locality to scale duplicate detection. Sparse indexing uses sampling and locality to reduce the chunk-lookup bottleneck in large-scale inline deduplication, and recent fine-grained deduplication frameworks further improve deduplication ratio while highlighting locality and backup/restore performance tradeoffs [33], [34]. These studies motivate locality-aware deduplication design, whereas FAD focuses on SSD chip-level layout and read parallelism.

Rewrite schemes write selected replicas according to fragmentation level to preserve data locality. iDedup [10] performs selective deduplication in primary storage, using a threshold so that only block sequences with sufficient deduplication potential are deduplicated. Kaczmarczyk et al. [35] rewrite highly fragmented duplicate blocks by comparing their stream context and disk context. Fu et al. [11] proposed the history-aware rewrite algorithm (HAR), which identifies and rewrites sparse containers based on historical backup information. Lillibridge et al. [36] introduced container capping, which uses fixed-size segments to identify blocks that should be rewritten. Because each container read introduces overhead, Tan et al. [37] proposed FGDefrag, a fine-grained defragmentation method that uses variable-size chunks to identify fragmented data more accurately. Zou et al. [27] proposed MFDedup, which performs duplicate detection on previous backups, classifies data, and uses an offline iterative algorithm to rearrange blocks into a compact sequential layout. These methods mainly optimize stream or container locality. In contrast, FAD uses chip-level duplicate-block distribution to decide whether placement alone is sufficient or selective rewriting is required.

Caching schemes use memory or SSDs to cache frequently referenced fragments or data blocks. This can improve restore speed but increases hardware and metadata cost. To improve cache hit rate, Kaczmarczyk et al. [35], Nam et al. [38], and Park et al. [39] use container-based caching, while other studies cache data blocks directly [40]. Lu et al. [8] proposed an elastic data cache (EDC) in primary storage to improve the I/O performance of deduplicated SSDs by identifying and caching frequently accessed data in DRAM. Caching reduces the number of physical reads, whereas FAD reduces chip-level conflicts for reads that still reach the SSD.

SSD deduplication has also been studied within the flash translation layer and SSD controller. CAFTL [2] and CA-SSD [3] integrate content-aware deduplication into the FTL to remove redundant writes and extend SSD lifetime. Remap-SSD [6] exploits SSD address remapping to eliminate duplicate writes safely and efficiently, and ARM-Dedup [7] offloads deduplication management from the host to SSD arrays. App-Dedup [9] and DedupHR [24] show that deduplication affects application and flash-storage performance. These works reduce write traffic or metadata overhead, whereas FAD focuses on the read-side fragmentation caused by the resulting physical layout and explicitly optimizes SSD chip-level parallelism.

VI. Conclusion

We presented FAD, a fragmentation-aware data layout strategy for deduplication-based SSDs. FAD classifies deduplication-induced fragmentation into UDF and NUDF and applies targeted optimizations to each type. For UDF files, FAD improves read parallelism through layout-aware placement of non-duplicate blocks. For NUDF files, it reduces chip-level read conflicts through selective rewriting of high-reference-count duplicate blocks. FAD also uses write-parallelism optimization to preserve write performance while improving read efficiency.

Experimental results show that FAD mitigates the performance degradation caused by deduplication-induced fragmentation. Compared with the evaluated deduplication baselines, FAD reduces both average read latency and tail latency while introducing only small write overhead. These results indicate that fragmentation-aware layout optimization is a practical approach for improving deduplication-based SSDs.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback. This work was supported by the National Key R&D Program of China No. 2023YFB4502703 and the National Natural Science Foundation of China under Grant No. U22A2027.

References

[1] Y. Jun, S. Park, J.-U. Kang, S.-H. Kim, and E. Seo, "We ain't afraid of no file fragmentation: Causes and prevention of its performance impact on modern flash SSDs," in *Proceedings of the 22nd USENIX Conference on File and Storage Technologies (FAST'24)*, 2024, pp. 193–208.

[2] F. Chen, T. Luo, and X. Zhang, "CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives," in *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*, 2011, pp. 77–90.

[3] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging value locality in optimizing NAND flash-based SSDs," in *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*, 2011, pp. 91–103.

[4] S. Wu, C. Du, H. Li, H. Jiang, Z. Shen, and B. Mao, "CAGC: A content-aware garbage collection scheme for ultra-low latency flash-based SSDs," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'21)*, 2021, pp. 162–171.

[5] Q. Yang, R. Jin, and M. Zhao, "SmartDedup: Optimizing deduplication for resource-constrained devices," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'19)*, 2019, pp. 633–646.

[6] Y. Zhou, Q. Wu, F. Wu, H. Jiang, J. Zhou, and C. Xie, "Remap-SSD: Safely and efficiently exploiting SSD address remapping to eliminate duplicate writes," in *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21)*, 2021, pp. 187–202.

[7] Y. Wen, X. Zhao, Y. Zhou, T. Zhang, S. Yang, C. Xie, and F. Wu, "Eliminating storage management overhead of deduplication over SSD arrays through a hardware/software co-design," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24)*, 2024, pp. 320–335.

[8] M. Lu, F. Wang, Z. Li, and W. He, "EDC: An elastic data cache to optimizing the I/O performance in deduplicated SSDs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 7, pp. 2250–2262, 2022.

[9] B. Mao, S. Wu, H. Jiang, X. Chen, and W. Yang, "Content-aware trace collection and I/O deduplication for smartphones," in *Proceedings of the 33rd IEEE International Conference on Massive Storage Systems and Technology (MSST'17)*, 2017, pp. 1–8.

[10] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti, "iDedup: Latency-aware, inline data deduplication for primary storage," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, 2012, pp. 1–14.

[11] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, J. Liu, W. Xia, F. Huang, and Q. Liu, "Reducing fragmentation for in-line deduplication backup storage via exploiting backup history and cache knowledge," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 3, pp. 855–868, 2016.

[12] Z. Cao, S. Liu, F. Wu, G. Wang, B. Li, and D. H. Du, "Sliding look-back window assisted data chunk rewriting for improving deduplication restore performance," in *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*, 2019, pp. 129–142.

[13] S. S. Hahn, S. Lee, C. Ji, L.-P. Chang, I. Yee, L. Shi, C. J. Xue, and J. Kim, "Improving file system performance of mobile storage systems using a decoupled defragmenter," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'17)*, 2017, pp. 759–771.

[14] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and C. Ren, "Exploring and exploiting the multilevel parallelism inside SSDs for improved performance and endurance," *IEEE Transactions on Computers*, vol. 62, no. 6, pp. 1141–1155, 2013.

[15] S. Li, F. Tu, L. Liu, J. Lin, Z. Wang, Y. Kang, Y. Ding, and Y. Xie, "ECSSD: Hardware/Data layout co-designed in-storage-computing architecture for extreme classification," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA'23)*, 2023, Art. no. 58, pp. 1–14.

[16] G. Yadgar, M. Gabel, S. Jaffer, and B. Schroeder, "SSD-based workload characteristics and their performance implications," *ACM Transactions on Storage*, vol. 17, no. 1, Jan. 2021, art. no. 8.

[17] W. Zhu and K. R. Butler, "NASA: NVM-assisted secure deletion for flash memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 3779–3790, 2022.

[18] Y.-H. Chen, T.-E. Liao, and L.-P. Chang, "iFKVS: Lightweight key-value store for flash-based intermittently computing devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 11, pp. 3564–3575, 2024.

[19] J. Jeon, J. Kim, S. H. Noh, and E. Seo, "AWUPF Rediscovered: Atomic writes to unleash pivotal fault-tolerance in SSDs," in *Proceedings of the 23rd USENIX Conference on File and Storage Technologies (FAST'25)*, 2025, pp. 441–448.

[20] B. S. Kim, H. S. Yang, and S. L. Min, "AutoSSD: An autonomic SSD architecture," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'18)*, 2018, pp. 677–690.

[21] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*, 2011, pp. 266–277.

[22] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee, "A superblock-based flash translation layer for NAND flash memory," in *Proceedings of the ACM International Conference on Embedded Software (EMSOFT)*, 2006, pp. 161–170.

[23] J. Bae, J. Park, Y. Jun, and E. Seo, "Dedup-for-Speed: Storing duplications in fast programming mode for enhanced read performance," in *Proceedings of the 15th ACM International Conference on Systems and Storage (SYSTOR'22)*, 2022, pp. 128–139.

[24] S. Wu, C. Du, W. Zhang, B. Mao, and H. Jiang, "DedupHR: Exploiting content locality to alleviate read/write interference in deduplication-based flash storage," *IEEE Transactions on Computers*, vol. 71, no. 6, pp. 1332–1343, 2022.

[25] R. Koller and R. Rangaswami, "I/O deduplication: Utilizing content similarity to improve I/O performance," *ACM Transactions on Storage*, vol. 6, no. 3, Art. no. 13, pp. 13:1–13:26, 2010.

[26] W. Li, G. Jean-Baptise, J. Riveros, G. Narasimhan, T. Zhang, and M. Zhao, "CacheDedup: In-line deduplication for flash caching," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*, 2016, pp. 301–314.

[27] X. Zou, J. Yuan, P. Shilane, W. Xia, H. Zhang, and X. Wang, "The dilemma between deduplication and locality: Can both be achieved?" in *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21)*, 2021, pp. 171–185.

[28] M.-C. Yen, S.-Y. Chang, and L.-P. Chang, "Lightweight, integrated data deduplication for write stress reduction of mobile flash storage," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2590–2600, 2018.

[29] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace, "Nitro: A capacity-optimized SSD cache for primary storage," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'14)*, 2014, pp. 501–512.

[30] J.-U. Kang, J.-S. Kim, C. Park, H. Park, and J. Lee, "A multi-channel architecture for high-performance NAND flash-based storage system," *Journal of Systems Architecture*, vol. 53, no. 9, pp. 644–658, 2007.

[31] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity," in *Proceedings of the 25th ACM International Conference on Supercomputing (ICS'11)*, 2011, pp. 96–107.

[32] C. Lee, T. Kumano, T. Matsuki, H. Endo, N. Fukumoto, and M. Sugawara, "Understanding storage traffic characteristics on enterprise virtual desktop infrastructure," in *Proceedings of the ACM International Systems and Storage Conference (SYSTOR'17)*, 2017, Art. no. 13, pp. 13:1–13:11.

[33] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble, "Sparse indexing: Large scale, inline deduplication using sampling and locality," in *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST'09)*, 2009, pp. 111–123.

[34] X. Zou, W. Xia, P. Shilane, H. Zhang, and X. Wang, "Building a high-performance fine-grained deduplication framework for backup storage with high deduplication ratio," in *Proceedings of the 2022 USENIX Annual Technical Conference (USENIX ATC'22)*, 2022, pp. 19–36.

[35] M. Kaczmarczyk, M. Barczynski, W. Kilian, and C. Dubnicki, "Reducing impact of data fragmentation caused by in-line deduplication," in *Proceedings of the 5th Annual International Systems and Storage Conference*, 2012, pp. 1–12.

[36] M. Lillibridge, K. Eshghi, and D. Bhagwat, "Improving restore speed for backup systems that use inline chunk-based deduplication," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, 2013, pp. 183–197.

[37] Y. Tan, J. Wen, Z. Yan, H. Jiang, W. Srisa-an, B. Wang, and H. Luo, "FGDEFRAG: A fine-grained defragmentation approach to improve restore performance," in *Proceedings of the 33rd IEEE International Conference on Massive Storage Systems and Technology (MSST'17)*, 2017, pp. 1–10.

[38] Y. Nam, G. Lu, N. Park, W. Xiao, and D. H. Du, "Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage," in *Proceedings of the 13th International Conference on High Performance Computing and Communications (HPCC'11)*. IEEE, 2011, pp. 581–586.

[39] D. Park, Z. Fan, Y. J. Nam, and D. H. Du, "A lookahead read cache: improving read performance for deduplication backup storage," *Journal of Computer Science and Technology (JCST)*, vol. 32, no. 1, pp. 26–40, 2017.

[40] B. Mao, H. Jiang, S. Wu, Y. Fu, and L. Tian, "SAR: SSD assisted restore optimization for deduplication-based storage systems in the cloud," in *Proceedings of the 2012 IEEE Seventh International Conference on Networking, Architecture, and Storage (NAS'12)*, 2012, pp. 328–337.