Don't Maintain Twice, It's Alright: Merged Metadata Management in Deduplication File System with GogetaFS

Yanqi Pan and Wen Xia, *Harbin Institute of Technology, Shenzhen;*
Erci Xu, *Alibaba Group;* Hao Huang, Xiangyu Zou, and
Shiyi Li, *Harbin Institute of Technology, Shenzhen*

https://www.usenix.org/conference/fast25/presentation/pan

This paper is included in the Proceedings of the
23rd USENIX Conference on File and Storage Technologies.

February 25–27, 2025 • Santa Clara, CA, USA

ISBN 978-1-939133-45-8

Open access to the Proceedings
of the 23rd USENIX Conference on
File and Storage Technologies
is sponsored by

NetApp®

Don't Maintain Twice, It's Alright: Merged Metadata Management in Deduplication File System with GOGETAFS

Yanqi Pan[†], *Wen Xia*[†], *Erci Xu*[‡], *Hao Huang*[†], *Xiangyu Zou*[†], *Shiyi Li*[†]
[†]*Harbin Institute of Technology, Shenzhen* [‡]*Alibaba Group*

Abstract

Emerging storage technologies, such as persistent memory and ultra-low latency SSD, enable the deduplication file system (DedupFS) to use non-cryptographic hash for fast fingerprinting. However, we find that the accelerated computation exposes another major performance penalty: the seemingly innocuous in-storage deduplication metadata maintenance incurs up to 38% overhead in the I/O path.

We find the root cause is that DedupFSes maintain dedup-specific *fingerprint-to-physical* mappings, which incurs additional crash consistency overheads. However, this overhead is unnecessary. Our insight is that the deduplication mapping can be merged with the file system *logical-to-physical* mapping, forming a *logical-fingerprint-physical* (LFP) mapping. Thus, we can persist deduplication metadata alongside file system metadata in a single I/O. We propose GOGETAFS to realize the efficiency of LFP. Using a series of techniques to manage data and metadata atop LFP, GOGETAFS achieves compatible, effective, and memory-efficient deduplication within the file system. Experiments across a range of workloads show that GOGETAFS consistently outperforms existing DedupFSes and can minimize metadata maintenance overheads.

1 Introduction

Deduplication is a classic technique to reduce storage costs by eliminating duplicates. Typically, developers would build a stand-alone deduplication system that sits atop the file system [15,20,24–26,32,40,43,55,56], maintains the fingerprints (FP) of data blocks, and identifies the potential duplicates of the incoming data stream. For a duplicated block (e.g., 4 KiB), the system can simply record/update the corresponding metadata to redirect its accesses to the existing copy.

Deduplication file systems (DedupFS) prompt a new look by embedding the deduplication logic into the file system itself [22,34,45,50,53,60]. Thus, DedupFSes can simplify implementation efforts by reusing the file index for data redirection. Moreover, the integration also means that the DedupFS can provide transparent data deduplication for applications, thereby attracting a wider range of applications [32,50]. While conventional DedupFSes can suffer from up to 65% computation overheads due to expensive cryptographic (crypto) FP computation (e.g., SHA-1) [45], recent advancements in storage devices, such as persistent memory [59] and ultra-low latency SSDs [31], enable the use of fast non-cryptographic (non-crypto) FPs (e.g., xxHash) plus content comparison, which reduces computation overheads to 15%–18% [45,50].

Despite these advantages and optimizations, we still find that the state-of-the-art DedupFSes [45,50,53] cannot deliver satisfying throughput. Particularly, we conduct a motivational study on three DedupFSes and observe that all of them leave considerable performance gaps compared to an ideal setup. To pinpoint the root cause, we break down the overhead along the I/O path and find that in-storage deduplication metadata maintenance consumes 18%–38% of I/O time.

Our in-depth analysis shows that the root cause of such overheads is the maintenance of a dedup-specific *fingerprint-to-physical (FP2P)* table in storage, which incurs extra crash consistency overheads. Specifically, DedupFSes enforce the FP2P entry (that maps the FP to a physical block number) to be written before the file system *logical-to-physical (L2P)* entry (that maps logical to physical block numbers). This update order ensures that inconsistencies between deduplication and file system metadata can be fixed efficiently, but it incurs metadata I/O twice and limits I/O parallelism due to ordering [15–17], causing a considerable penalty on DedupFS.

However, we find such overheads are unnecessary. Our key insight is that the deduplication FP2P entry and the file system L2P entry can share the same physical block number, and thus can be merged into a single *logical-fingerprint-physical (LFP)* mapping. Thus, we can persist the FP2P mapping alongside the file system L2P entry in a single I/O. This approach reuses the mature file system I/O path and crash consistency mechanisms for deduplication FP2P entry, which simplifies implementation and improves robustness. Further, it also eliminates inconsistencies between deduplication and file system metadata, as FP2P mapping is always available in L2P entries.

Notably, the LFP mapping raises a classic tradeoff between space overhead and performance: it extends FP2P mappings to all L2P entries even if they are identical (i.e., more space), but it minimizes in-storage deduplication metadata maintenance overheads (i.e., higher performance). Unfortunately, this tradeoff is not well-suited to traditional DedupFSes due to their large size of crypto FP (e.g., 20 bytes [60]) and marginal acceleration, as the overhead is dominated by computation. However, with emerging fast storage [31,59] and the use of smaller and faster non-crypto FPs (e.g., 4–8 bytes [12,19,45]), LFP presents a promising solution for DedupFSes.

We propose GOGETAFS atop LFP. The challenge is that the seemingly ideal merge of FP2P and L2P incurs incompatibilities: (1) *Data management*. While file systems intend to manage data in an extent manner [9,58], deduplication manages data block-by-block. To reconcile this, GOGETAFS

stores FPs for block extents into a storage table, which then uses the file system crash consistency mechanism for persistence without extra ordering. (2) *Metadata management*. File systems organize metadata by logical block numbers for fast indexing, but deduplication fails to leverage this to search FPes. To address this, GOGETAFS converts LFP entries into FP2P entries and organizes them into an in-memory table, using FPs as keys. This table can be rebuilt via LFP entries, enabling flexible designs without crash consistency concerns. Based on this, GOGETAFS explores various designs related to entry placement, memory allocation, and data structures to improve memory efficiency under various scenarios.

To evaluate the efficiency of GOGETAFS, we conduct a comprehensive evaluation using both synthetic and real-world workloads. Our results show that GOGETAFS outperforms state-of-the-art DedupFSes by 5.6%–35.0% in I/O throughput and significantly reduces deduplication metadata overheads. We further evaluate GOGETAFS variants under different memory scenarios and show that the variant designed for the specific scenario can improve I/O throughput by up to 10%.

In summary, this paper makes the following contributions:

- We conduct a series of motivational studies and reveal that deduplication metadata maintenance can be a major performance penalty in DedupFSes due to extra I/Os and ordering.
- We propose the LFP mapping that merges FP2P and LFP entries, which then minimizes deduplication metadata maintenance penalty by persisting deduplication metadata alongside file system metadata without twice maintenance.
- We build GOGETAFS based on the LFP mapping[1]. Using a series of techniques to manage data and metadata atop LFP, GOGETAFS achieves compatible, effective, and memory-efficient deduplication within the file system.
- Evaluations on synthetic and real-world workloads demonstrate that GOGETAFS outperforms existing DedupFSes and minimizes metadata maintenance penalty.

2 Background

2.1 Stand-alone Deduplication System

Overview. Deduplication has long been a hot topic in both academia and industry, especially with the recent exponential growth in data. The classic solution is to build a stand-alone system that can identify and eliminate redundant data blocks. Such a system usually targets a specific scenario, such as archives [24–26, 43, 55] and the cloud [13, 33, 39, 49], and sits atop a (distributed) file system that provides data persistence. Note that some stand-alone systems implement deduplication beneath the file system, such as block-layer deduplication [48]. Regardless of the type, stand-alone deduplication systems inherently interact with file systems.

Key data structure. At a high level, a stand-alone deduplication system typically sets up two key tables (see Figure 1a). The L2P table maps the logical block number (LBN) to the physical block number (PBN)[2]. The FP2P table maps the fingerprint (FP) to the corresponding PBN, along with a reference count that indicates the redundancy of the very block. The block size is typically fixed, such as 512 bytes [48] or 4 KiB [51]. In addition, an in-memory *FP index* is usually introduced to accelerate FP lookups in the FP2P table [48, 51].

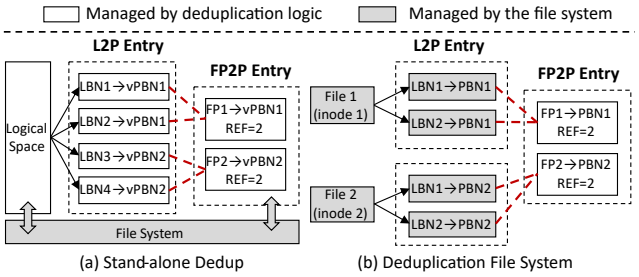

Figure 1: **The comparison of critical in-storage mappings in deduplication systems.** *Compared to stand-alone deduplication systems, deduplication file systems can simplify implementation efforts and avoid double metadata maintenance overheads. Here, vPBN in (a) indicates that the PBN is exposed as the "block" in a large volume file [63] instead of a physical block in disk.*

Workflow. In practice, a stand-alone system performs deduplication in the following steps. For an incoming data block, the system first calculates its cryptographic FP and queries the FP2P table. If an FP2P entry is found (i.e., duplicated), the system increments the corresponding reference count instead of writing the data. If not, the system inserts a new entry into the FP2P table and then invokes the underlying file system to persist the data block. In both cases, the system updates the L2P table to reflect the new mapping. Deletion is simply the other way around: the system first removes the L2P entry, decreases the reference count of the FP2P entry, and only releases the physical block when the count reaches zero.

Metadata management. With redundant writes converted into updates on the L2P and FP2P tables, the management of these metadata becomes critical. Existing solutions include write-ahead logging/journaling [41, 46], copy-on-write [40], and soft update [27] to ensure the atomicity of metadata updates and guarantee crash consistency.

2.2 Deduplication File System

Overview. Apart from having an independent system, another viable option is to embed deduplication logic into the file system to form a DedupFS (see Figure 1b) [34, 45, 50]. In this setup, when the DedupFS receives an incoming write, it tries to locate the corresponding FP2P entry and updates the L2P entry associated with the inode of the target file. Then, similar to the stand-alone approach, the DedupFS updates the FP2P entry accordingly based on the presence of the incoming data.

Advantages. The benefits of DedupFS are two-fold. First, it avoids the double metadata maintenance overheads. Stand-alone deduplication systems, built atop file systems (e.g.,

[1] Our source code, test scripts/tools, and results are available at https://github.com/GogetaFS/.

[2] Note that another option is maintaining the LBN-to-FP mapping to simplify data movement [40, 51, 63], but this increases PBN lookup overheads with one FP indirection. This paper focuses on the LBN-to-PBN mapping.

EXT4), maintain their own L2P entries instead of using file system's L2P entries, leading to double metadata maintenance overheads. DedupFS avoids this issue since the two L2P tables are now merged[3] and deduplication can only maintain the FP2P table with simplified crash consistency techniques, such as soft update [15]. Second, DedupFS can provide transparent deduplication services for any application that uses them.

Reduce DedupFS memory overhead. In the past decades, DedupFS has focused on reducing memory overheads due to the deduplication FP index. SmartDedup [60] builds a three-level radix tree to index FP2P entries across both memory and mobile flash storage. DeNOVA [34] is an offline/background deduplication approach that adopts the FP index in the byte-addressable storage. However, both of these approaches compute crypto FPs, incurring high computation overheads. Unfortunately, DeNOVA fails to hide such overheads as its background thread can block the foreground I/O operation [45].

Improve DedupFS performance. Recent advancements in storage devices, such as persistent memory (PM, e.g., Optane DCPMM [59]) and ultra-low latency SSDs (ULL SSDs, e.g., Z-SSD [31]), have led to disruptive changes in the deduplication landscape. One notable change is the shift of bottlenecks from I/O to CPU [45, 50], enabling the replacement of crypto FPs with the non-crypto FPs plus content comparison [45, 65]. Specifically, this approach first calculates fast non-crypto FPs and then performs content comparison only when FPs are the same. As a result, state-of-the-art non-crypto-based DedupFSes, such as HFDedup [53] and Light-Dedup [45], reduce redundancy identification overheads to 20% or less [45]. NV-Dedup, another example, also employs fast deduplication by calculating non-crypto FPs at low duplication ratios, but it falls back to crypto FPs at higher deduplication ratios [50].

3 Observation and Motivation

3.1 DedupFS Overhead Overview

With a series of optimizations in both hardware and software, one question arises: *does the overhead of introducing deduplication logic remain to be excessive*? To answer this question, we set up a series of motivational experiments to explore and understand the overheads of different DedupFSes.

Methodology. We start by using FIO [2] to stress three DedupFSes, i.e., DeNOVA, NV-Dedup, and Light-Dedup, and a theoretical upper bound Ideal-Dedup. Ideal-Dedup is set up in two configurations: (1) with a 0% duplication ratio, Ideal-Dedup functions as a normal file system (i.e., NOVA [58]); and (2) with non-zero duplication ratios, Ideal-Dedup only performs FP computation, FP indexing, and byte-by-byte content comparison. To cover different scenarios, we form the workloads as 4 KiB sequential writes but with different duplication ratios, where 0% indicates each 4 KiB block is unique, while 25% indicates a quarter of the total 4 KiB blocks have the exact same content, etc. We use the same hardware platform and software environment as described in §6.1.

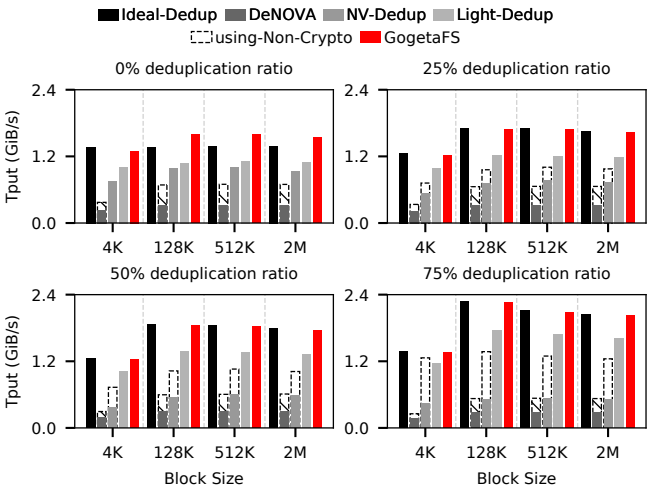

Figure 2: **Throughput comparison among existing DedupFSes**. *We stress sequential-write workloads with different block sizes per I/O and duplication ratios. "using Non-Crypto" indicates the DedupFS (i.e., DeNOVA and NV-Dedup) that simply replaces the cryptographic FP with the non-cryptographic FP. GogetaFS is our proposed DedupFS. We leave the reason why GogetaFS is even faster than Ideal-Dedup in the 0% duplication ratio in §4.3.*

Observation #1: State-of-the-art DedupFSes fail to achieve promising performance improvements. Figure 2 shows that, under the 0% duplication ratio, all DedupFSes are slower than Ideal-Dedup. This is reasonable since, for each write, a DedupFS needs to calculate FPs and look up the FP2P table. Hence, compared to Ideal-Dedup, DedupFSes (with non-crypto FPs) can suffer a 20% to 42% performance loss. Note that DeNOVA is 1.9×–3.4× slower than others since it adopts the more expensive crypto FP calculation.

To our surprise, with an increasing duplication ratio from 0% to 25% and finally 75%, the performance gaps *do not* narrow. This result is unexpected because, theoretically, the performance should converge, given that both Ideal-Dedup and DedupFSes perform deduplication on every block and write the same amount of data. Furthermore, even when we replace the crypto FP of DeNOVA and NV-Dedup with non-crypto FPs[4] (i.e., *using Non-Crypto* in Figure 2), the performance gap persists. Note that this situation remains unchanged even with larger block I/O sizes (e.g., 128 KiB to 2 MiB).

3.2 DedupFS Overhead Breakdown

The above observations lead us to ask a follow-up question: *what exacts the toll on DedupFS processing*?

Methodology. We categorize DedupFS overheads into five procedures: (1) *Dup Identify* includes FP calculation, FP lookup, and content comparison time; (2) *Dedup Metadata* is the time to record/update the FP2P entry; (3) *Data Write* is the time to write data blocks; (4) *FS Metadata* is the time to write file system L2P entry; and (5) *Others* denotes the rest

[3] Some works propose in-device deduplication, which reuses the L2P table of device, e.g., FTL in SSD [62]. However, this requires modifying firmware, which is inaccessible in off-the-shelf SSDs, and thus is beyond our scope.

[4] Note the replacement is technically incorrect, as different blocks can be identified as duplicates, but it allows us to better observe performance trends.

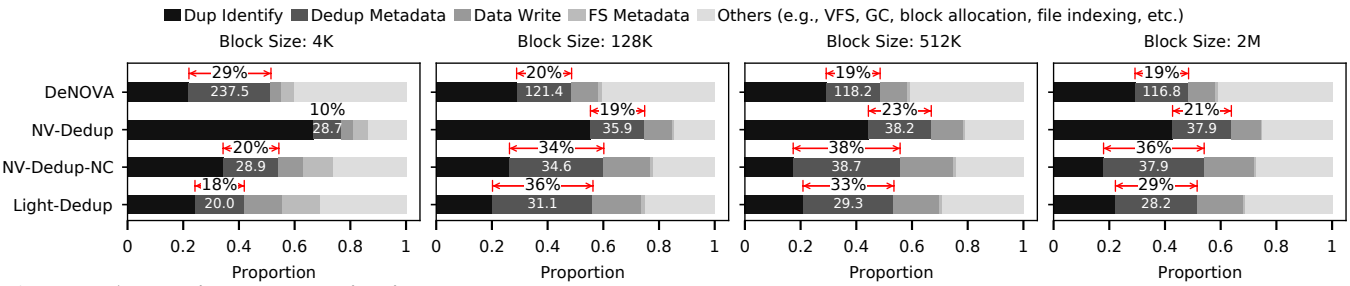

Figure 3: **Approximate deduplication overheads breakdown**. *We measure the breakdown under the 50% duplication ratio as a representative case. NV-Dedup-NC is short for NV-Dedup using the non-cryptographic FP. The white numbers are Dedup Metadata time in (s).*

of the file system time consumption, such as VFS overheads, etc. We insert the timing code around each part to measure the time consumption with the same settings as described in §3.1. Note that, due to unpredictable CPU reordering, the measurement might deviate but only at a small scale (i.e., nanoseconds level), which does not affect the overall trend.

Observation #2: Deduplication metadata overheads are non-negligible. We select the 50% duplication ratio scenario as a representative case to analyze the deduplication overheads. Figure 3 reveals that crypto FP calculation does lead to heavy overheads (e.g., NV-Dedup). However, when FP computation is accelerated with non-crypto FPs, one major penalty arises from deduplication metadata maintenance. For example, NV-Dedup with the non-crypto FP (i.e., NV-Dedup-NC) and Light-Dedup suffer from 20%–38% and 18%–36% metadata maintenance overheads across 4 KiB to 2 MiB block I/O size. These overheads, as a result, dominate over 50% deduplication time (i.e., *Dup Identify* + *Dedup Metadata*).

Overhead analysis. The root reason for such overheads is that DedupFSes strictly order the FP2P entry to survive system failures [15] (e.g., power outages). Specifically, to maintain the integrity of FP2P entries, DedupFS often leverages the soft update to track dependency between L2P and FP2P entries, and enforces the FP2P entry to be written/updated before the L2P entry is persisted [15, 34, 45, 50, 60]. Consequently, a crash occurring between the two procedures can only result in a *higher-than-actual* reference count issue [15]. Therefore, the DedupFS can scan L2P entries to re-calculate the reference count of each PBN [45] and remove the FP2P entry with a zero reference count to fix inconsistency.

However, writing and ordering FP2P entries incur two major problems: (1) I/O amplification due to the mismatched I/O granularity between small metadata and coarse storage I/O unit (e.g., 256 bytes in Optane DCPMM and 4–16 KiB in modern SSD); (2) reduced I/O parallelism due to waiting for the completion of the previous I/O [15–17]. Although adopting coarse-grained metadata management to align with the storage I/O granularity (e.g., Light-Dedup) can partially alleviate the issue, our results still show the performance penalty is non-negligible (e.g., 18%–36% for Light-Dedup in Figure 3).

3.3 Merged Metadata: A New Hope?

The Logical-Fingerprint-Physical (LFP) mapping. Intuitively, one solution to minimize such overhead is to reduce

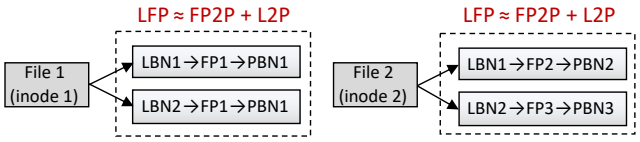

Figure 4: **The Logical-Fingerprint-Physical mapping**. *LFP integrates the deduplication FP2P entry into the file system L2P entry to minimize the deduplication metadata maintenance overheads.*

the number of I/Os and ordering points. To this end, a viable approach, as Figure 4 shows, is to merge L2P and FP2P entries into an LBN-FP-PBN (LFP) entry, indexed by the inode. In this way, DedupFSes can persist the FP2P mapping along with the file system L2P entry in a single I/O. This merging raises a classic *tradeoff* between space and performance: the LFP entry incurs redundant FP for the same PBN (i.e., more space), but it can minimize the dedup metadata maintenance penalty (i.e., higher performance).

Why LFP mapping was not proposed in the past. We believe the primary reason is that traditional DedupFSes are designed for slow storage, which is why they use crypto FPs for fingerprinting. This, however, makes LFP less attractive due to the large space overheads and marginal performance improvement. Quantitatively speaking, let the size of PBN/LBN, reference count, and crypto FP be 8, 8, and 20 bytes, respectively. Given N input blocks with a duplication ratio R, traditional DedupFSes maintain $16 \times N$ bytes of L2P entries (=LBN+PBN) and $36 \times N \times (1-R)$ bytes of FP2P entries (=FP+PBN+ReferenceCount), totaling $52N - 36NR$ bytes. In contrast, LFP-based DedupFSes (with crypto FPs) maintain $36 \times N$ bytes of LFP entries (=LBN+FP+PBN), which incurs up to $2.25\times$ space overhead compared to the traditional approach (see Figure 5). However, the performance gains are limited (e.g., a 10%–23% improvement, see NV-Dedup in Figure 3), as crypto FP calculation remains the bottleneck.

Opportunities for LFP today. Emerging fast storage devices bring opportunities for non-crypto FPs, offering new hopes for LFP mapping: with a typical non-crypto FP being 8 bytes, the LFP entry size is reduced to $24 \times N$ bytes. Figure 5 shows that this can even save metadata space when the duplication ratio is $< 80\%$ (which is common to satisfy [32,60]); meanwhile, non-crypto-FP-based DedupFSes can be potentially accelerated by 18%–36% (see Light-Dedup in Figure 3). However, existing DedupFSes simply follow traditional approaches to manage

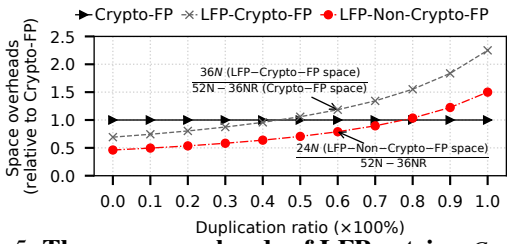

Figure 5: **The space overheads of LFP entries**. *Crypto-FP is the traditional DedupFS that separates the L2P entry from the FP2P entry. LFP-Crypto-FP and LFP-Non-Crypto-FP denote the LFP entries with the crypto and non-crypto FP, respectively. N is the number of input data blocks, and R is the duplication ratio.*

the FP2P table, thereby missing the opportunity to revisit LFP.

Advantages. We summarize the benefits of using the LFP mapping in DedupFSes as follows:

- *Reuse the mature file system I/O path and crash consistency mechanism*. DedupFS leverages the file system to guarantee the persistence and crash consistency of the LFP entry by using the same I/O path of the file system L2P entry, thereby reducing implementation efforts and improving robustness.
- *Reduce metadata I/O amplification and ordering points*. The LFP entry avoids maintaining an FP2P table in storage, reducing I/O amplification and ordering points.
- *Eliminate inconsistencies between the deduplication and file system*. As the FP2P mapping is always available in the crash-tolerant LFP entry, there are no inconsistencies to fix, which can further accelerate failure recovery.

Challenges. The challenge of LFP for DedupFSes lies in its incompatibility: (1) *Data management*. While file systems intend to manage data in extents [9, 58], deduplication manages data block-by-block (for block-level deduplication). Consequently, merging FP2P and L2P can lead to variable-length L2P entries: assume a block is 4 KiB, L2P entries need one FP to manage one block but 512 FPs for contiguous 2 MiB blocks. This, however, complicates file system I/O path. (2) *Metadata management*. Since file systems organize LFP entries by LBN for fast file indexing, FP searches (for deduplication) within LFP entries degrade to expensive linear scans, significantly reducing deduplication efficiency.

4 GogetaFS Design

Our previous analysis on the possibility of combining the L2P and FP2P motivates us to design GOGETAFS, a new DedupFS with merged metadata management. Yet, the potential challenges indicate that the design of GOGETAFS is non-trivial. Therefore, we start by identifying the design goals.

4.1 Design Goals

- *Generic*. GOGETAFS key techniques should be adapted to different file systems in an general manner.
- *Effective*. GOGETAFS aims to provide block granularity (e.g., 4 KiB) deduplication and efficient FP searching.
- *Memory-efficient*. GOGETAFS should be memory-efficient towards a wide spectrum of scenarios.
- *Configurable*. GOGETAFS should allow users to disable the deduplication feature, without (or with marginal) impacts on the original file system efficiency and functionality.

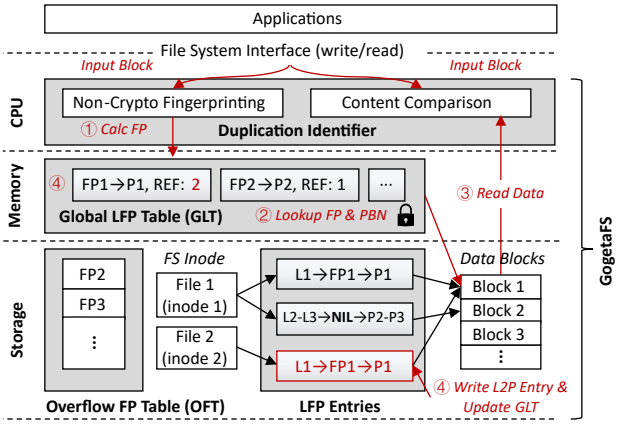

Figure 6: **GogetaFS overview**. *Gray areas are the major components, including LFP entries, Overflow FP Table, Global LFP Table, and a non-cryptographic-FP duplication identifier.*

4.2 Overview

GogetaFS components. Figure 6 illustrates the major components in GOGETAFS. First, GOGETAFS extends the file system L2P entry with one fixed-size FP (for a 4 KiB block) as the LFP entry. Second, it reserves *Overflow FP Table (OFT)* in storage to store FPs for contiguous data blocks. The crash consistency of OFT is guaranteed alongside the file system crash consistency mechanism, requiring no extra ordering. Third, GOGETAFS maintains *Global LFP Table (GLT)* that aggregates and converts LFP entries into FP2P entries for global deduplication. GLT can be reliably rebuilt during file system recovery by examining crash-tolerant LFP entries and OFT, allowing for a flexible design without crash consistency concerns. Finally, GOGETAFS provides a non-crypto-FP-based duplication identifier for fast redundancy identification.

Deduplication workflow. Given the above components, we describe the basic workflow of GOGETAFS as follows:

- *Write a duplicate block.* As Figure 6 illustrates, assume a user writes a block to L1 (LBN1) of file 2, GOGETAFS ① calculates the FP for the input data block (i.e., FP1) and ② queries GLT, finding that there is a corresponding FP1 → P1 mapping (of file 1). Then, GOGETAFS ③ reads the data block P1, compares it with the input data block, and determines it is a duplicate. Finally, GOGETAFS ④ writes a new LFP entry L1 → FP1 → P1 to file 2 without block allocation, and increments the reference count of P1.
- *Write (a) unique block(s).* In the previous example, if step ② finds no mapping, GOGETAFS allocates and writes the data block, and writes a new LFP entry to the file. If step ③ indicates two blocks are different, GOGETAFS treats the input block as a *non-dedup block* (that is not maintained by GLT) similar to other non-cryptographic-FP-based DedupFSes [45]. An exceptional case is writing contiguous unique data blocks (e.g., writing two blocks to L2 and L3 of file 1), GOGETAFS leverages OFT to store the FPs and sim-

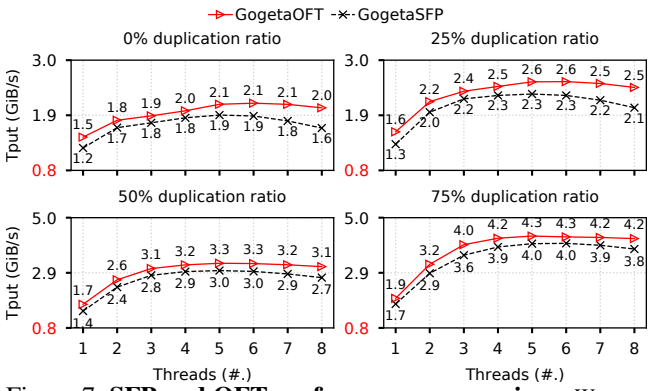

Figure 7: **SFP and OFT performance comparison**. *We conduct sequential contiguous 2 MiB block I/O using FIO.*

ply assigns a `NIL` FP to the corresponding LFP entry (see L2–L3 → NIL → P2–P3[5] in the Figure 6).

- *Others.* To read a block, GOGETAFS simply follows traditional file systems read path: retrieve the PBN according to the LFP entry, and read the data block from the storage. To delete a block (e.g., caused by `truncate`), GOGETAFS invalidates the corresponding LFP entry, decrements the reference count, and frees the block if the count is zero.

Advantages. With these techniques, GOGETAFS achieves its design goals: (1) it is generic as it ensures compatible I/O paths with file systems by introducing fixed-size LFP entries with OFT (see §4.3); (2) it is effective as it can achieve effective deduplication across files by enabling fast FP comparison with GLT (see §4.4); (3) it is memory-efficient as it can adjust the memory consumption by providing various GLT structures to accommodate different scenarios without consistency overheads (see §4.5); and finally, (4) GOGETAFS is configurable as deduplication can be disabled by setting the FP to zero and bypassing the deduplication logic. The extra overheads are negligible as measured in §6.2.

4.3 Fine-grained Data Management

To reconcile the incompatible data management between the deduplication (i.e., block-by-block) and file system (i.e., extent [9, 58]), we explore two orthogonal approaches.

Approach #1: coarse-grained data management to match with file system. One naïve approach is to align deduplication granularity with the extent size. Specifically, we can calculate a super-fingerprint (SFP) for whole contiguous data blocks and store it as one FP. However, SFP reduces the deduplication ratio due to the increased deduplication granularity. Moreover, calculating an SFP followed by a large data I/O leads to low computation and I/O parallelism, as these two operations are serialized. In contrast, block-level FP calculations can overlap with block I/Os, thanks to the hardware buffer that allows asynchronous I/Os [45]. Note that the computation also reduces buffer contention by lowering the I/O issue frequency. As a result, it can lead to even higher throughput than Ideal-Dedup under a 0% duplication ratio (see Figure 2).

Approach #2: fine-grained data management to match with deduplication. Consequently, we explore an alternative approach: keep block-level deduplication and store FPs (of contiguous blocks) in a reserved in-storage area, namely *Overflow FP Table (OFT)*, which is built as a one-to-one PBN-to-FP mapping array. To store FPs, GOGETAFS first locates array slots by PBNs, and then stores their FPs sequentially.

Using OFT has four advantages. First, OFT is a lock-free data structure that can be updated in parallel as one PBN only corresponds to one FP. Second, OFT benefits from the spatial locality as the FPs of contiguous data blocks are stored together. Third, when a contiguous group of blocks is deleted, it is unnecessary to invalidate OFT slots as these blocks are not referenced by any LFP entry, and these slots can therefore be overwritten safely for future write requests. Fourth, OFT does not introduce an extra ordering point by leveraging the file system crash consistency. We consider two common cases:

- *Journaling file systems* wrap L2P entry updates to a journal area before updating it in place [14, 23]. GOGETAFS can add OFT updates to the journal together with LFP entry updates. As a result, OFT updates and the LFP entry can both be persisted atomically without extra ordering points.
- *Log-structured file systems* append L2P entry updates to the log and then commit them by updating the log tail [35, 54, 58]. Thus, GOGETAFS can also append OFT updates to the log, and rely on the ordered log tail commitment to ensure crash consistency without extra ordering points.

Performance comparison. Figure 7 compares these two approaches under the same deduplication ratios with 2 MiB block I/O. The results demonstrate that GOGETAOFT consistently outperforms GOGETASFP due to low OFT update overheads and high computation and I/O parallelism.

4.4 Dedup-friendly Metadata Management

To address the incompatible LFP entry organization (e.g., by LBN), one solution is to add a per-file FP index alongside the per-file index to organize LFP entries by FP, but this approach limits FP searches to intra-file scope, reducing deduplication ratios. To address this, we propose *Global LFP Table (GLT)*.

GLT for fast FP searching. As shown in Figure 6, GLT aggregates and interprets LFP entries into FP2P entries, namely, GLT entries. Specifically, GLT retrieves the FP2P mapping from LFP entries and tracks the number of identical mappings as the reference count. As Figure 8a shows, GLT organizes these entries within a dynamic hash table, using FP as keys to enable fast FP searching. The hash table is further protected via RCU locks, which support highly concurrent, read-most operations [11], making it well-suited to frequent FP lookups and comparisons in deduplication workflows.

Handle concurrency. One concern is to correctly handle concurrent duplicate I/Os. Assume thread #1 writes block A as a unique block and is about to insert a new GLT entry, while thread #2 writes the same block A but it does not know that

[5] Note that if these input blocks are all duplicates, GOGETAFS also stores `NIL` in their LFP entry as FPs have been stored in OFT.

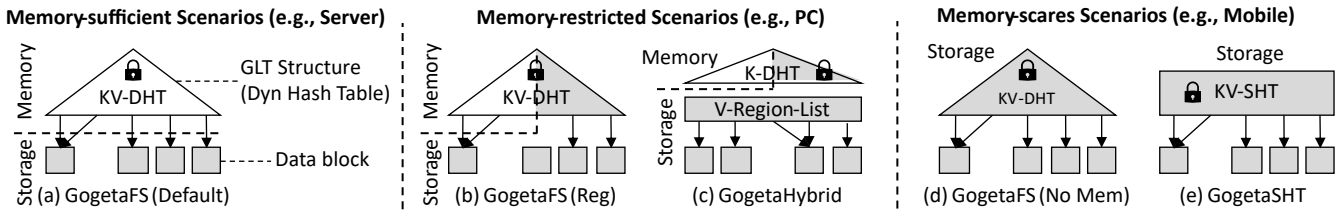

Figure 8: **GogetaFS variants design.** *The gray areas are in the storage. The key (K) is FP while the value (V) is the reference count and PBN. (a) GLT is organized by a dynamic hash table; (b) regulates memory usage for (a); (c) separates K and V, and stores V in a region-grained list, similar to a log structure. Only K is cached. Light-Dedup shares the similar architecture of (c), but it requires explicit flushing and ordering for values to ensure crash consistency; (d) deploys all GLT entries of (a) in the storage; (e) deploys static hash table structures in storage.*

A is a duplicate as thread #1 has not updated GLT yet. As a result, the same block A is written twice. Using a global lock to exclusively lookup and insert a GLT entry can address this issue but limits scalability. To address this, GOGETAFS takes an optimistic concurrency control approach: it checks the existence of the FP right before insertion without locking the lookup routine. If the FP already exists, GOGETAFS drops the insertion and redoes the current-thread deduplication process to ensure correctness. Indeed, the check-in-insertion process requires locking for correctness, but the lock is held for a short time in a per-bucket manner. With RCU locks, entry lookups (by other threads) can be performed concurrently with insertions, thereby improving concurrency.

Advantages. GLT offers two key advantages over traditional FP2P table: (1) GLT has no crash consistency requirements (i.e., no explicit flushing or ordering points) as it can be rebuilt during file system recovery by examining crash-tolerant LFP entries and OFT (see §4.6). (2) GLT can thus be deployed in memory, benefiting from direct access without the need for an extra FP index or loading data from storage for access.

4.5 Techniques Towards Memory Efficiency

Maintaining GLT in memory is feasible for memory-sufficient scenarios. However, it is impractical for the scenarios with limited memory. To overcome the issue, we design GOGETAFS variants with different GLT designs that empirically consider the placement of table entries, the memory allocation strategy, and data structures to accommodate different scenarios.

Memory-constrained scenarios. For these scenarios (e.g., PC), the system software should be conservative in memory usage [34]; therefore, it is crucial to regulate GLT memory usage. We track the memory usage of GLT and trigger in-storage allocation when the memory usage exceeds a predefined threshold. With this regulation, as shown in Figure 8b, GOGETAFS can access GLT entries residing in memory and storage (via virtual memory or *direct access*). Additionally, we introduce GOGETAHYBRID, which separates GLT entries into FP (as keys) and PBN with reference count (as values). This design caches only the keys, allowing for more keys to be cached and speeding up FP searches, as shown in Figure 8c.

Memory-scarce scenarios. In these scenarios (e.g., mobile devices), memory is extremely restricted. However, the deduplication requirements are still there, especially considering the streaming sensor data, ever-growing event logs, and limited storage space [5, 44, 54]. For these small devices, we aim to deploy GLT in storage. Intuitively, we put all the keys and values in storage, as shown in Figure 8d. Nevertheless, building dynamic hash tables in storage requires maintaining fine-grained pointers of each node, which can lead to excessive read-modify-writes and I/O amplification. To address the overhead, we design GOGETASHT, which builds a static hash table (SHT) in storage, as shown in Figure 8e.

4.6 Crash Consistency and Recovery

The recovery of the GOGETAFS requires reconstructing GLT.

Normal recovery. During normal recovery, it is feasible to scan LFP entries and recalculate the reference count of each PBN to reconstruct GLT. However, this usually incurs random I/O overheads as file systems intend to separately store LFP entries for different files to achieve spatial locality. To address this, GOGETAFS dumps GLT entries into a reserved file during a normal shutdown, and the file is scanned concurrently to rebuild GLT during the next recovery.

Failure recovery. For system failures detected by file systems (e.g., via magic numbers [23,58]), many file systems scan LFP entries to ensure file system metadata consistency [21, 58]. During scanning, GOGETAFS retrieves the FP from either the LFP entry or from OFT if the LFP entry has a `NIL` FP (§4.3). The FP and PBN are used to generate a new GLT entry with a reference count equal to one or update the count if the entry already exists in GLT. As a result, GLT can be reconstructed along with the file system failure recovery.

Crash consistency. GOGETAFS leverages file system crash consistency mechanisms to ensure correct failure recovery (§4.3). Particularly, (1) for a journaling file system, LFP and OFT updates can be replayed, thus there are no partially updated entries; (2) for a log-structured file system, if an LFP entry is valid, the corresponding OFT entries are also valid as they are committed atomically. In cases where a crash occurs before the entry commitment, leading to partial OFT updates, the incomplete OFT entries remain unreferenced by any valid LFP entry. Thus, GOGETAFS never accesses incomplete OFT entries. Based on this, GLT can then be reliably reconstructed by scanning LFP entries and their associated OFT entries.

4.7 Discussion

Random data read effects. Similar to other existing DedupFSes [45, 50], GOGETAFS requires CoW-style data updates

to avoid corrupt LFP entries (i.e., the stored FP mismatches with modified data content), which breaks file locality and causes random block I/O. However, random block I/O is fast for modern storage devices as they have high I/O parallelism that can hide the random block I/O overheads [29, 57]. Particularly, sequential and random 4 KiB write/read I/Os of PM have similar performance [57].

Space overhead of OFT. GOGETAFS reserves OFT to store FPs for block extents. However, the space might never be used if there are no contiguous data blocks. It is possible to dynamically allocate the OFT space, but it requires extra metadata management and potentially increases the locating overheads. Considering the space overheads are limited (i.e., $\frac{8}{4096} \times 100\% \approx 0.2\%$ space overheads for 8 byte FPs), we argue that reserving OFT space for performance is feasible.

5 Implementation

We have built GOGETAFS based on NOVA (in PM) and F2FS (in SSD). We introduce them in §5.1 and §5.2.

5.1 GogetaFS Based on NOVA

Duplication identifier. We borrow Light-Redundant-Block-Identifier (LRBI) from Light-Dedup [45] as our duplication identifier, which adopts non-crypto FPs and a PM-optimized speculative-prefetching-based content comparison: this technique maintains a pointer from the last deduplicated block to the current one as a hint, and uses this hint to speculate and prefetch the next to-be-deduplicated block to accelerate content comparison. For example, given a write stream with unique blocks A and B, GOGETAFS maintains a pointer from A to B. When A is written again, GOGETAFS checks the pointer of A and speculatively prefetches B to boost further content-comparison. For further acceleration, we replace the xxHash with a faster wyHash [10, 12] for fingerprinting.

LFP entry. One bonus of NOVA is that we can integrate the FP2P entry into the L2P entry without extra space overheads: we reuse the 4 byte crc32 checksum and 4 byte padding in `nova_write_entry` (i.e., the L2P entry). The checksum is originally used to guarantee the integrity of the write entry, but it is optional as PM promises to be reliable [42, 64]. We thus provide two modes: the default is *fast mode*, which overwrites both the checksum and padding with the 8-byte wyHash FP; another is *secure mode*, which stores the high 4-byte FP in the padding without affecting the checksum. Compared to *fast mode*, *secure mode* can lead to more hash collisions and content comparison. However, our evaluation in §6.3 shows that the performance overheads are minor thanks to prefetch.

OFT. OFT is reserved next to the superblock of NOVA as an FP array. OFT is updated by first issuing memory stores to the corresponding contiguous slots, and then flushing in batches without ordering points (i.e., calling `clwb` without `sfence`).

GogetaFS variants. We present five variants (in Figure 8):

- GOGETAFS variants in three scenarios. To enable prefetching, we assign each GLT entry (§4.4) with an extra 8 byte `hint` for prediction (pointing to another entry) and 8 byte `counter` to avoid use-after-free issues. We use the *relativistic hash table* [1] as the dynamic hash table to organize these entries. By allocating GLT entries entirely in DRAM, partially in PM, or entirely in PM, we implement three variants corresponding to Figures 8a, 8b, and 8d.
- GOGETAHYBRID. Built on GOGETAFS, GOGETAHYBRID further separates GLT entry into the FP (as the key) and the PBN with reference count (as the value), and uses a region-grained list to organize values in storage [45]. Each region is 4 KiB, allocated via file system allocators, and linked through a reserved `next` pointer. Values are allocated to regions with more than half of free space to improve spatial locality, in a sequential manner. The memory regulation technique is applied only to keys, as shown in Figure 8c.
- GOGETASHT builds GLT as an in-PM static hash table (SHT). The SHT is organized at a 256-byte bucket granularity, where each bucket can store up to 8 GLT entries. By default, the table is reserved to store entries for twice the number of blocks (i.e., capacity). Entry placement is determined by taking the modulo of the hash value with the capacity, using the lowest 3 bits as the offset within the bucket, and the remaining bits to determine the bucket index. Linear probing is used to resolve hash collisions. For simplicity, table extension is not currently supported.

Workload	I/O (GB)	Dup Ratio	Target Memory Scenarios
FIO [2]	32	0%-100%	Any memory scenarios
CP [45]	20.84	100%	Enough/Constrained memory
OSLab [6]	63.52	84%	Enough memory
WebVMs [32]	54.53	47%	Enough memory
Mails [32]	173.27	95%	Enough memory
Homes [32]	43.92	69.03%	Constrained memory
Nexus [60]	20.12	32.40%	Scarce memory

Table 1: **Evaluated workloads**. *FIO measures basic I/O performance; CP copies a compiled Linux folder, which measures backup performance. OSLab is a trace collected from the HITSZ OS lab server; Homes, WebVMs and Mails are three FIU traces for studying I/O patterns in personal laptop usage, web and mail servers. Nexus is the trace collected from real-world mobile devices by VISA Lab.*

Recovery. Normal recovery is trivial: we reserve an inode in NOVA and storing GLT mappings there. For failure recovery, we integrate GLT construction into `nova_set_file_bm`, in which GOGETAFS scans all the collected write entries along with NOVA to rebuild GLT mappings. Meanwhile, the `hint` and `counters` are reinitialized during recovery.

5.2 Extending GogetaFS to Other File Systems

GogetaFS atop F2FS. F2FS is a block-based file system that leverages the block layer for storage access [35]. Our implementation involves three adaptations. First, for the duplication identifier, we replace the prefetch function from `prefetcht0` used in the PM platform to `sb_breadahead`, as SSD does not support byte-addressability. Second, for the in-storage layout, we extend the F2FS block index entry to store one FP. OFT is eliminated as each block index entry in F2FS maps only one block. Third, in memory-constrained scenarios, as SSDs do not allow direct access to GLT entries, we adopt a

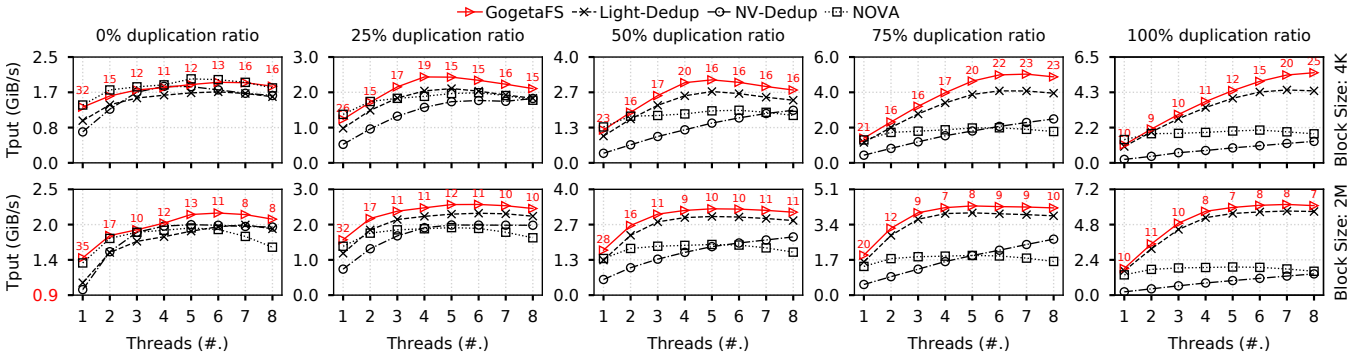

Figure 9: **FIO write throughput comparison under sufficient memory**. *The top five graphs show the write throughput with a common 4 KiB per I/O while the bottom five graphs show the write throughput with a larger 2 MiB per I/O. The red numbers highlight the percentage of throughput improvement of GogetaFS over Light-Dedup.*

GogetaHybrid-like variant that keeps *all* keys in memory while loading (then caching) values from the SSD as needed.

Others. Extending GogetaFS to other non-NOVA file systems is similar to F2FS. To this end, we shall identify and modify their L2P entries to accommodate an FP field. Particularly, for EXT4 and BtrFS, we can insert an FP field into the `ext4_extent` and `btrfs_file_extent_item`, respectively. Then, we can adapt OFT structures, GLT structures, and the remaining deduplication logic (that manipulates memory) to port GogetaFS to these file systems.

6 Evaluation

Our evaluation answers the following questions:

- Does GogetaFS introduce extra overheads when the deduplication feature is disabled? (§6.2)
- How does GogetaFS perform compared to the traditional DedupFSes and the base file system? (§6.3)
- How much throughput improvement does the LFP mapping bring and how it brings such improvements? (§6.4)
- How do the different GogetaFS variants perform? (§6.5)
- Can GogetaFS improve DedupFS recovery performance by recovering along with the file system recovery? (§6.6)
- How does GogetaFS scale with ULL SSDs? (§6.7)

6.1 Setup and methodology

Testbed. If not otherwise specified, experiments are carried out on a server with an Intel Xeon Gold 5218 CPU@2.3 GHz, 16 cores (with hyper-thread enabled), and 22 MiB of L3 cache. The machine is equipped with 512 GiB Intel Optane DCPMM (2×256 GiB DIMMs) configured in the non-interleaved AppDirect Mode and 128 GiB DRAM (4 × 32 GiB DIMMs). The server runs CentOS stream with kernel 5.1.0 [3]. We further evaluate GogetaFS (atop F2FS) on an emulated FEMU-based Z-SSD platform in §6.7.

Competitors. We compare GogetaFS with Light-Dedup, NV-Dedup, DeNOVA, and NOVA[6]. Light-Dedup is configured with no *delay flushing* to ensure that FP2P entries are correctly written and not lost. By default, all evaluated (deduplication) file systems are configured with no metadata protection (i.e., fast mode for GogetaFS).

Methodology. We summarize the workloads used to emulate different scenarios in Table 1. We utilize *mcp* [7] and *trace replayer* [8] to back up Linux and replay traces. During experiments, we use up to 8 threads, consistent with prior works [45], as a higher number of threads can lead to performance degradation due to PM's internal contentions [45, 57, 59]. If PM concurrency can be doubled, we believe GogetaFS can also double its throughput when the threads reach 16 due to its high-concurrent lock-free OFT and RCU-based GLT.

6.2 Deduplication as a Feature

We compare GogetaFS with deduplication enabled/disabled using FIO, with dup ratios from 0% to 100%. We conduct *t-tests* [30] over ten runs to identify the statistical significance of the performance difference. The results suggest that the p-value is ∼0.20 (>0.05), indicating that the differences are not significant. Hence, GogetaFS allows using deduplication as a feature without affecting original file system's performance.

6.3 Performance Evaluation

We measure the default GogetaFS performance under the memory-sufficient scenarios (e.g., server applications).

Basic I/O performance. Figure 9 shows that GogetaFS consistently outperforms NV-Dedup and Light-Dedup. Specifically, with a 0% duplication ratio, GogetaFS is very close to NOVA under 4 KiB block I/O due to the reduced number of I/Os and ordering points. Notably, for 2 MiB block I/O, GogetaFS even outperforms NOVA. This is because FP computation overlaps with PM internal I/Os, reducing PM buffer contention by lowering the I/O issue frequency [61].

Real-world workloads performance. Figure 10 presents results similar to those observed in the basic I/O tests. However, GogetaFS exhibits more pronounced performance improvements with 2 MiB block I/O, particularly in workloads such as Mails and WebVMs. These workloads feature more realistic I/O patterns, including overwrites that trigger block deletions. GogetaFS handles deletions more efficiently: while traditional DedupFSes must invalidate up to 512 FP2P entries (by setting reference counts to zero) to delete a 2 MiB block,

[6] NOVA is an optimized variant, which delays the inode integrity verification until the file `open`/`close` and updates the in-memory file data index once the corresponding *write entry* is durable.

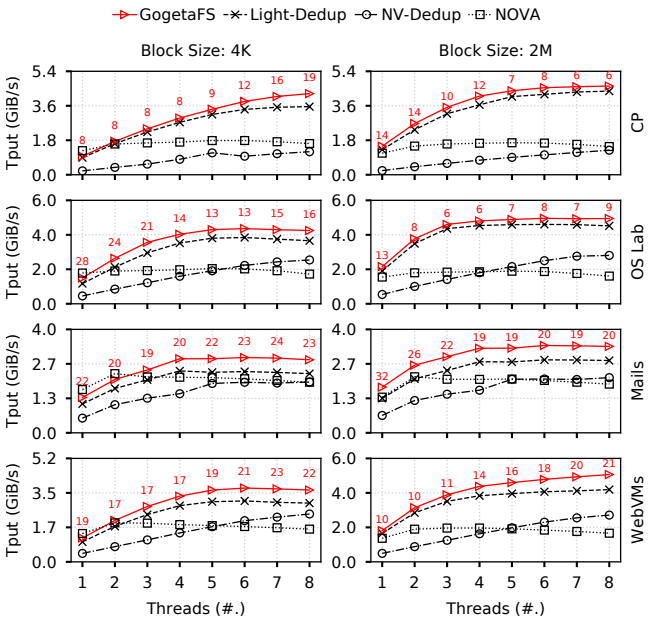

Figure 10: **I/O throughput comparison under real-world scenarios**. *CP denotes the backup scenarios while traces represent server scenarios. The red numbers highlight the percentage of throughput improvement of GogetaFS over Light-Dedup.*

GOGETAFS only needs to invalidate a single LFP entry (§4.3).

Secure mode overheads. We conduct t-tests on GOGETAFS in both fast and secure modes across all above workloads. The p-values, ranging from 0.96 to 0.98 for FIO workloads and 0.75 to 0.98 for real-world workloads (>0.05), indicate that the performance overheads of secure mode are minor thanks to the fast speculative-prefetching-based content comparison.

6.4 Metadata Maintenance Overhead

In this subsection, we first demonstrate that the deduplication performance improvement of GOGETAFS does come from the reduced metadata maintenance overheads. We then measure the deduplication-induced I/O amplification to understand how the LFP mapping improves the I/O throughput.

Deduplication performance breakdown. Figure 11 shows that GOGETAFS has 75.4%–92.8% lower deduplication metadata overheads compared to Light-Dedup. The remaining overheads are mainly due to the in-DRAM GLT lookup and insertion, which cannot be avoided. Additionally, we find that wyHash (in GOGETAFS) does not always outperform xxHash (in Light-Dedup), and the primary overheads for redundancy identification arise from the byte-by-byte content comparison.

I/O amplification. We measure extra deduplication-induced read/write bytes per 4 KiB block (RPB/WPB) in GOGETAFS using *ipmctl* [4]. The same workload used to evaluate Light-Dedup is employed to simulate a fresh and aged system [45]. Specifically, we first populate a 128 GiB file with no duplication, then randomly punch 50% holes inside the file to simulate the aged scenario. Finally, we write another 64 GiB file with the same content as the first file's first half. Table 2 shows that GOGETAFS consistently achieves much lower deduplication-induced I/O than Light-Dedup. With reduced

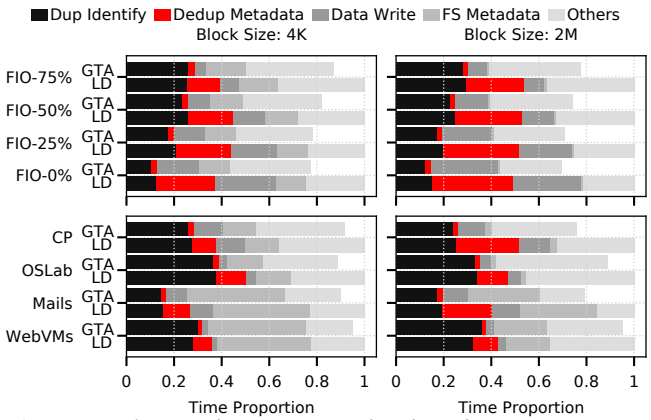

Figure 11: **Approximate deduplication time breakdown**. *We compare normalized breakdown performance between GogetaFS (GTA) and Light-Dedup (LD) under a single thread. GLT and OFT maintenance are reported as Dedup Metadata for GogetaFS.*

System	Metric	RPB	WPB	BW	Lat
Light-Dedup	Fresh-4K	132.2	115.6	1104.4	3537.1
GOGETAFS		∼0	∼0	1328.6	2940.1
Light-Dedup	Aged-4K	226.3	158.2	1085.5	3598.4
GOGETAFS		∼0	∼0	1228.7	3179.3
Light-Dedup	Fresh-2M	214.9	126.7	1181.9	3305.2
GOGETAFS		70.7	26.7	1591.6	2454.4
Light-Dedup	Aged-2M	288.8	174.2	1124.2	3474.8
GOGETAFS		146.0	82.0	1298.9	3007.3

Table 2: **I/O amplification comparison**. *RPB and WPB denote extra read/write bytes per 4 KiB block I/O. BW is bandwidth in MiB/s. Lat is latency in nanoseconds for deduplicating one block.*

metadata I/Os, GOGETAFS can improve the I/O throughput by 13%–35% compared to Light-Dedup in both fresh and aged scenarios. Specifically, GOGETAFS achieves nearly zero deduplication I/O amplification for 4 KiB block I/O; as a result, aging negligibly affect the I/O throughput. For 2 MiB block I/O, OFT introduces several I/O amplification, but GOGETAFS still outperforms Light-Dedup by 15%–35% thanks to the spatial locality of OFT (§4.3).

6.5 Performance Under Different Scenarios

This subsection studies the performance of GOGETAFS variants under other scenarios with limited memory.

Memory-constrained scenarios. We study the performance of GOGETAFS (with partial keys and values in PM, see Figure 8b), GOGETAHYBRID, and Light-Dedup under constrained memory across four settings. As Figure 12 shows, GOGETAFS variants consistently outperform Light-Dedup as the LFP mapping eliminates the extra I/Os and ordering for FP2P entries. We also find that (1) GOGETAHYBRID can achieve higher I/O throughput than GOGETAFS when memory is low (e.g., sufficient only for half of keys) as it requires fewer in-storage key accesses; (2) when memory increases (e.g., enough for more than half of GLT entries), GOGETAFS outperforms GOGETAHYBRID as it can access many keys and values in DRAM, while the latter stores all values in PM, thereby losing opportunities for fast value accesses in DRAM.

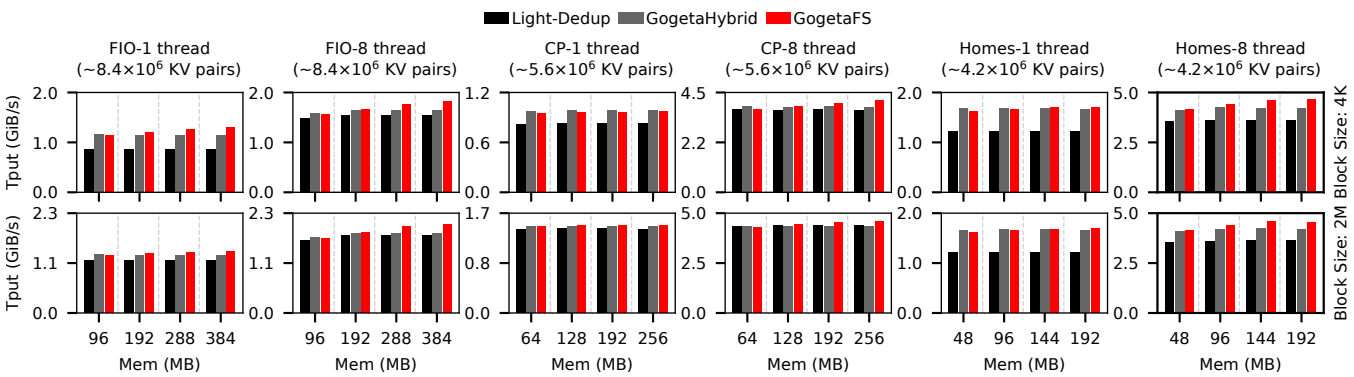

Figure 12: **I/O throughput comparison under constrained memory**. *The memory is regulated for keys (24 bytes) in Light-Dedup/GogetaHybrid, and GLT entries (48 bytes) in GogetaFS: half keys (a quarter of entries), all keys (half of entries), three quarters of entries, and all entries. We evaluate workloads across different duplication ratios (i.e., 0% for FIO, 100% for CP, and 69.03% for Homes).*

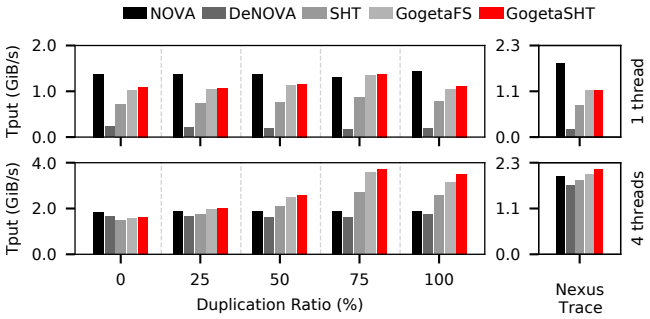

Figure 13: **I/O throughput comparison of emulated logging (left) and mobile trace (right) under scarce memory**. *We use 4 KiB block I/O and up to 4 threads to emulate this scenario.*

Memory-scarce scenarios. We compare GOGETAFS (Figure 8d), GOGETASHT, SHT (GOGETASHT without LFP), DeNOVA, and NOVA using FIO to simulate sensor data collection and event logging scenarios. In practice, these collected/logged data are stored and then uploaded to a server [5]. Here, we focus on the data storage phase, excluding uploads, as all DedupFSes show similar read performance. We also evaluate Nexus trace to assess mobile workloads. Given the limited resources, we conduct the evaluation with 4 KiB I/O and up to 4 threads, aligning with prior studies [60]. Figure 13 suggests that even with no dedicated deduplication memory, GOGETAFS variants outperform other DedupFSes (i.e., SHT and DeNOVA). Among these variants, GOGETASHT outperforms GOGETAFS by 1%–10% due to its reduced random I/O for pointer accesses and updates. However, NOVA outperforms GOGETASHT by 10%–36% when the duplication ratio is below 50% under a single thread. This is not surprising as in-PM GLT access still leads to I/O amplification.

6.6 Recovery Overhead

Table 3 compares normal shutdown, recovery, and failure recovery of NOVA, Light-Dedup, and GOGETAFS. GOGETAFS spends more time on normal shutdown and recovery as it requires dumping both keys and values from GLT into PM and restoring them during recovery. However, GOGETAFS shows around 9.5%–30.8% faster failure recovery than Light-Dedup as it can recover along with the file system recovery without extra checking for deduplication metadata (i.e., FP2P table). Note that NOVA outperforms Light-Dedup and GOGETAFS as it does not need to reconstruct FP index or GLT.

System	Metric	FIO	CP	OS	MA	VM
NOVA	Umount Time (s)	0.39	0.97	0.75	0.51	0.92
Light-Dedup		0.67	1.18	0.95	0.75	1.15
GOGETAFS		1.16	1.52	1.11	1.12	1.22
NOVA	Normal Recovery Time (s)	0.02	0.01	0.01	0.01	0.01
Light-Dedup		0.66	0.37	0.21	0.43	0.14
GOGETAFS		0.69	0.42	0.23	0.49	0.17
NOVA	Failure Recovery Time (s)	0.47	0.34	0.78	0.57	0.89
Light-Dedup		1.88	1.28	2.37	2.00	2.90
GOGETAFS		1.70	1.01	1.64	1.68	2.09

Table 3: **Recovery overheads**. *FIO creates 32 GiB data; OS, MA, and VM are short for OSLab, Mails, and WebVMs.*

6.7 GogetaFS Beyond Optane DCPMM

To emulate ultra-low latency SSDs, we run FEMU in `nossd` mode to simulate Z-SSD performance [36]. We reproduce SmartDedup and HFDedup based on F2FS. We also port Light-Dedup to F2FS and compare it with GOGETAFS. These DedupFSes persist FP2P entries to a sparse file using F2FS I/O interfaces. SmartDedup calculates SHA-1 for fingerprinting; HFDedup is optimized to use wyHash FP. Light-Dedup leverages `sb_breadahead` to accelerate content comparison. Figure 14 shows that GOGETAFS (atop F2FS) significantly outperforms competitors under both single-threaded FIO and real-world traces. Light-Dedup shows some improvement when the duplication ratio increases, but metadata maintenance overheads become the bottleneck. SmartDedup performs the worst since it has to calculate cryptographic hash for each block. Notably, compared to PM, SSDs exhibit relatively minor read-write asymmetry (for deduplication optimization); therefore, GOGETAFS achieves only 79%–99.6% of F2FS's performance. However, GOGETAFS is much faster than existing SSD-based DedupFSes thanks to its merged LFP entries that minimize metadata overheads.

7 Limitation and Future Work

Portability. GOGETAFS intrusively modifies the file system, which may increase the development complexity. Fortunately,

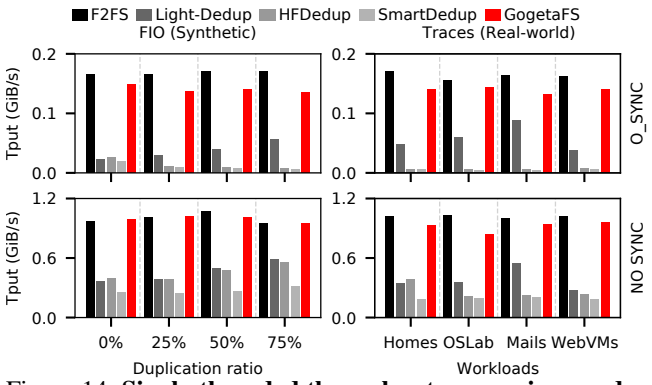

Figure 14: **Single-threaded throughput comparison under Z-SSD**. *The upper two figures perform strong persistence [15] with O_SYNC; the lower two figures have no persistence guarantees.*

the changes are often easy to maintain: to support LFP entries, GogetaFS adds only ∼50 LOC in F2FS (e.g., mainly for macros and block index); while the remaining deduplication logic is modularized, which mainly manipulates in-memory structures and can be easily ported. However, memory regulation techniques are storage-specific. For example, on SSDs, GogetaFS cannot store keys in storage due to the lack of byte-addressability. To reduce memory usage on SSDs, it is feasible to group values [60] to reduce the number of keys needed for indexing, which we leave as future work.

Memory swap in PM. Currently, GogetaFS (with memory regulation) directly accesses GLT entries in PM without swapping. While swapping hot GLT entries from PM to DRAM is feasible, the extra overheads and potential performance gains are uncertain. Figure 12 shows that even with ideal swapping, i.e., no thread blocking and all accesses hitting DRAM (equivalent to having the maximum available memory), GogetaFS achieves only 1%–19% throughput improvement. Therefore, we leave memory swapping to future work.

Supporting offline/background deduplication. The use of non-cryptographic FPs can result in two different data blocks having the same FP. In this case, GogetaFS (like others [38, 45]) manages only the first input data block while treating the second one as a non-dedup block (§4.2). This might lead to the loss of deduplication opportunities. Fortunately, the collision rate for 8 byte xxHash/wyHash is low (i.e., lower than the expected collision rate) [10]. However, for scenarios where a high deduplication ratio is a critical concern, background deduplication could be introduced to re-deduplicate these blocks, which we plan to explore as future work.

Supporting data movement. Many flash-based file systems perform data block migration for garbage collection or wear leveling [35, 54]. To support data movement, we can extend the P2L mapping (e.g., SSA in F2FS) to track the mapping from PBN to multiple LBNs for fast LFP entry updates during migration (similar to [28]), leaving it as future work.

8 Related Work

To the best of our knowledge, GogetaFS is the first DedupFS to explore file system's I/O paths and crash consistency mechanisms to minimize deduplication metadata overheads. Before our work, deduplication focuses on the following aspects:

Duplication identification acceleration. Purity [18] deduplicates enterprise flash storage by computing (no more than) 8 byte hashes across 512 bytes and stores 1/8 of the hashes. On PM, except for NV-Dedup [50] and Light-Dedup [45], DeWrite [65] notices the write-read asymmetry of PM and deploys non-crypto fingerprinting and content comparison in the cache line level for hardware deduplication. Our proposed LFP mapping stands on the shoulders of these works to minimize deduplication metadata maintenance penalty.

Deduplication metadata space reduction. Wei et al. [52] propose to alleviate FPs of zero blocks using a special code word (e.g., 1 byte). Meister et al. [40] proposes to use compression techniques to reduce FP space. Metadedup [37] deduplicates metadata to save space. These methods focus on slow disks and crypto FPs, while GogetaFS targets modern fast storage with smaller, non-crypto FPs. By integrating the FP2P entry into the L2P entry, GogetaFS minimizes deduplication metadata maintenance penalty with minor space overheads.

Metadata management in stand-alone deduplication systems. These systems leverage various crash consistency techniques to guarantee the integrity of deduplication metadata. iDedup [47] leverages an NVRAM to stage writes in a log-structured layout; dedupv1 [41] maintains a redo journaling for recovery; Dmdedup [48] uses CoW to ensure the atomicity of deduplication metadata updates. However, these approaches interact with file systems, leading to double metadata maintenance overheads (§2.1). In contrast, GogetaFS leverage the LFP mapping that merges deduplication and file system metadata to eliminate crash consistency overheads.

9 Conclusion

We propose GogetaFS, a novel DedupFS that leverages the file system mature I/O path and crash consistency mechanism to improve deduplication throughput. The key insight is to build a DedupFS with *logical-fingerprint-physical (LFP)*, which is a novel mapping technique that merges the deduplication FP2P entry with the file system L2P entry. We implement and evaluate GogetaFS in both PM and (emulated) ULL SSD platforms. The results suggest that GogetaFS outperforms existing DedupFSes, sometimes by an order of magnitude (e.g., in the SSD platform), and achieves minimized deduplication metadata maintenance overheads.

Acknowledgment

We thank our shepherd, Gala Yadgar, and anonymous reviewers for their constructive comments and insightful suggestions. This research was partly supported by the National Natural Science Foundation of China under Grant 62472127; Shenzhen Science and Technology Program under Grants RCYX20210609104510007, KJZD20230923114610021, and GXWD20231128111309001; Guangdong Basic and Applied Basic Research Foundation under Grant 2023A1515110072. Wen Xia and Erci Xu are the corresponding authors.

References

[1] Relativistic Hash Tables, Part 1: Algorithms. https://lwn.net/Articles/612021/, 2014.

[2] Flexible I/O Tester. https://github.com/axboe/fio, 2017.

[3] NOVA: NOn-Volatile memory Accelerated log-structured file system. https://github.com/NVSL/linux-nova, 2017.

[4] Ipmctl. https://github.com/intel/ipmctl, 2018.

[5] An introduction to iot logging types and practices. https://www.techtarget.com/iotagenda/tip/An-introduction-to-IoT-logging-types-and-practices, 2022.

[6] Hitsz oslab trace. https://github.com/Light-Dedup/tests/releases/tag/homes-2022-fall-50, 2023.

[7] Multiple-Thread Copy Tool. https://github.com/Light-Dedup/mcp.git, 2023.

[8] Trace Replayer. https://github.com/Light-Dedup/nvm_tools/blob/master/helper/replay.c, 2023.

[9] Ext4 Design. https://ext4.wiki.kernel.org/index.php/Ext4_Design, 2024.

[10] SMhasher. https://github.com/rurban/smhasher.git, 2024.

[11] Using rcu to protect read-mostly linked lists. https://www.kernel.org/doc/html/latest/RCU/listRCU.html, 2024.

[12] wyhash. https://github.com/wangyi-fudan/wyhash.git, 2024.

[13] Pramod Bhatotia, Rodrigo Rodrigues, and Akshat Verma. Shredder: GPU-accelerated Incremental Storage and Computation. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, pages 1–15, San Jose, CA, 2012.

[14] Mingming Cao, Suparna Bhattacharya, and Ted Ts'o. Ext4: The Next Generation of Ext2/3 Filesystem. In *Proceedings of the 2007 Linux Storage & Filesystem Workshop (LSF)*, pages 1–36, San Jose, CA, USA, 2007.

[15] Zhuan Chen and Kai Shen. OrderMergeDedup: Efficient, Failure-Consistent Deduplication on Flash. In *Proceedings of 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 291–299, Santa Clara, CA, USA, 2016.

[16] Vijay Chidambaram et al. Consistency Without Ordering. In *Proceedings of 10th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–16, San Jose, CA, USA, 2012.

[17] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 228–243, Farmington, PA, USA, 2013.

[18] John Colgrove, John D. Davis, John Hayes, Ethan L. Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. Purity: Building Fast, Highly-Available Enterprise Flash Storage from Commodity Components. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, page 1683–1694, New York, NY, USA, 2015.

[19] Yann Collet. xxhash: Extremely fast hash algorithm, 2016.

[20] Biplob K Debnath, Sudipta Sengupta, and Jin Li. ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory. In *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC)*, pages 1–16, Boston, MA, USA, 2010.

[21] David Domingo and Sudarsun Kannan. pFSCK: Accelerating File System Checking and Repair for Modern Storage. In *Proceedings of 19th USENIX Conference on File and Storage Technologies (FAST)*, pages 113–126, Virtual Event, USA, 2021.

[22] Yunsheng Dong, Boju Chen, Yanqi Pan, Xiangyu Zou, and Wen Xia. H2C-Dedup: Reducing I/O and GC Amplification for QLC SSDs from the Deduplication Metadata Perspective. In *Proceedings of the 2024 ACM Symposium on Cloud Computing (SoCC)*, page 704–719, Redmond, WA, USA, 2024.

[23] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*, pages 1–15, Amsterdam, Netherlands, 2014.

[24] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Jingning Liu, Wen Xia, Fangting Huang, and Qing Liu. Reducing Fragmentation for In-line Deduplication Backup Storage via Exploiting Backup History and Cache Knowledge. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 27(3):855–868, 2016.

[25] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Fangting Huang, and Qing Liu. Accelerating Restore and Garbage Collection in Deduplication-based Backup Systems via Exploiting Historical Information. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC)*, pages 181–192, Philadelphia, PA, USA, 2014.

[26] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Yucheng Zhang, and Yujuan Tan. Design Tradeoffs for Data Deduplication Performance in Backup Workloads. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 331–344, Santa Clara, CA, USA, 2015.

[27] Gregory R Ganger, Marshall Kirk McKusick, Craig AN Soules, and Yale N Patt. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM Transactions on Computer Systems (TOCS)*, 18(2):127–153, 2000.

[28] Cheng Ji, Li-Pin Chang, Riwei Pan, Chao Wu, Congming Gao, Liang Shi, Tei-Wei Kuo, and Chun Jason Xue. Pattern-Guided File Compression with User-Experience Enhancement for Log-Structured File System on Mobile Devices. In *Proceedings of 19th USENIX Conference on File and Storage Technologies (FAST)*, pages 127–140, Virtual Event, USA, 2021.

[29] Yuhun Jun, Shinhyun Park, Jeong-Uk Kang, Sang-Hoon Kim, and Euiseong 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)*, pages 193–208, 2024.

[30] Tae Kyun Kim. T Test as a Parametric Statistic. *Korean Journal of Anesthesiology*, 68(6):540–546, 2015.

[31] Sungjoon Koh, Changrim Lee, Miryeong Kwon, and Myoungsoo Jung. Exploring system challenges of ultra-low latency solid state drives. In *Proceedings of the 10th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)*, pages 1–7, Boston, MA, USA, 2018.

[32] Ricardo Koller and Raju Rangaswami. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. *ACM Transactions on Storage (TOS)*, 6(3):1–26, 2010.

[33] Iwona Kotlarska, Andrzej Jackowski, Krzysztof Lichota, Michal Welnicki, Cezary Dubnicki, and Konrad Iwanicki. InftyDedup: Scalable and Cost-Effective Cloud Tiering with Deduplication. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST)*, pages 33–48, Santa Clara, CA, 2023.

[34] Hyungjoon Kwon, Yonghyeon Cho, Awais Khan, Yeohyeon Park, and Youngjae Kim. Denova: Deduplication extended nova file system. In *Proceedings of the 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1360–1371, Lyon, France, 2022.

[35] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 273–286, Santa Clara, CA, USA, 2015.

[36] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator. In *Proceedings of 16th USENIX Conference on File and Storage Technologies (FAST)*, Oakland, CA, USA, 2018.

[37] Jingwei Li, Patrick P. C. Lee, Yanjing Ren, and Xiaosong Zhang. Metadedup: Deduplicating Metadata in Encrypted Deduplication via Indirection. In *Proceedings of the 35th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 269–281, Santa Clara, CA, USA, 2019.

[38] Mengting Lu, Fang Wang, Dan Feng, and Yuchong Hu. A Read-leveling Data Distribution Scheme for Promoting Read Performance in SSDs with Deduplication. In *Proceedings of the 48th International Conference on Parallel Processing (ICPP)*, Kyoto, Japan, 2019.

[39] Bo Mao, Hong Jiang, Suzhen Wu, and Lei Tian. POD: Performance Oriented I/O Deduplication for Primary Storage Systems in the Cloud. In *Proceedings of the 28th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 767–776, Phoenix, AZ, USA, 2014.

[40] Dirk Meister, Andre Brinkmann, and Tim Süß. File Recipe Compression in Data Deduplication Systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 175–182, San Jose, CA, USA, 2013.

[41] Dirk Meister and André Brinkmann. Dedupv1: Improving Deduplication Throughput using Solid State Drives (SSD). In *Proceedings of the 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–6, Incline Village, NV, USA, 2010.

[42] Ethan L Miller, Scott A Brandt, and Darrell DE Long. HeRMES: High-performance Reliable MRAM-enabled Storage. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS)*, pages 95–99, Schloss Elmau, Germany, 2001.

[43] Jaehong Min, Daeyoung Yoon, and Youjip Won. Efficient Deduplication Techniques for Modern Backup Operation. *IEEE Transactions on Computers (TC)*, 60(6):824–840, 2011.

[44] Yanqi Pan, Zhisheng Hu, Nan Zhang, Hao Hu, Wen Xia, Zhongming Jiang, Liang Shi, and Shiyi Li. HNFFS: Revisiting the NOR Flash File System. In *Proceedings of the 11th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 14–19, Virtual Event, Taiwan, 2022.

[45] Jiansheng Qiu, Yanqi Pan, Wen Xia, Xiaojia Huang, Wenjun Wu, Xiangyu Zou, Shiyi Li, and Yu Hua. LightDedup: A Light-weight Inline Deduplication Framework for Non-Volatile Memory File Systems. In *Proceedings of the 2023 USENIX Annual Technical Conference (USENIX ATC)*, pages 101–116, Boston, MA, USA, 2023.

[46] Chunlin Song, Xianzhang Chen, Duo Liu, Xiaoliu Feng, Xi Yu, Jiali Li, Yujuan Tan, and Ao Ren. CADedup: High-performance Consistency-aware Deduplication Based on Persistent Memory. In *Proceedings of the 40th International Conference on Computer Design (ICCD)*, pages 726–729, Olympic Valley, CA, USA, 2022.

[47] Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. iDedup: Latency-aware, Inline Data Deduplication for Primary Storage. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, pages 1–14, San Jose, CA, USA, 2012.

[48] Vasily Tarasov, Deepak Jain, Geoff Kuenning, Sonam Mandal, Karthikeyani Palanisami, Philip Shilane, Sagar Trehan, and Erez Zadok. Dmdedup: Device Mapper Target for Data Deduplication. In *Proceedings of the 2014 Ottawa Linux Symposium (OLS)*, pages 83–95, Ottawa, Canada, 2014.

[49] K. Venkatesh and D. Narasimhan. Revealing the Novel Precise Subset Identification and Deduplication of Audio Substance Over the Shared Public Environment. *The Journal of Supercomputing*, page 11856–11872, 2022.

[50] Chundong Wang, Qingsong Wei, Jun Yang, Cheng Chen, Yechao Yang, and Mingdi Xue. NV-Dedup: High-performance Inline Deduplication for Non-volatile Memory. *IEEE Transactions on Computers (TC)*, 67(5):658–671, 2017.

[51] Qiuping Wang, Jinhong Li, Wen Xia, Erik Kruus, Biplob Debnath, and Patrick PC Lee. Austere Flash Caching with Deduplication and Compression. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC)*, pages 713–726, Virtual Event, USA, 2020.

[52] Jiansheng Wei, Hong Jiang, Ke Zhou, and Dan Feng. MAD2: A Scalable High-throughput Exact Deduplication Approach for Network Backup Services. In *Proceedings of the 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14, Incline Village, NV, USA, 2010.

[53] Kai-Ting Weng, Yun-Shan Hsieh, Yen-Ting Chen, Yu-Pei Liang, Yuan-Hao Chang, Po-Chun Huang, and Wei-Kuan Shih. Hf-dedupe: Hierarchical fingerprint scheme for high efficiency data deduplication on flash-based storage systems. In *Proceedings of 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9, San Francisco, CA, USA, 2023.

[54] David Woodhouse. JFFS: The Journalling Flash File System. In *Proceedings of the 2001 Ottawa Linux Symposium (OLS)*, pages 1–12, Ottawa, Canada, 2001.

[55] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. SiLo: A Similarity-Locality based Near-Exact Deduplication Scheme with Low RAM Overhead and High Throughput. In *Proceedings of the 2011 USENIX annual technical conference (USENIX ATC)*, pages 26–30, Portland, OR, USA, 2011.

[56] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. FastCDC: A Fast and Efficient Content-Defined Chunking Approach for Data Deduplication. In *Proceedings of 2016 USENIX Annual Technical Conference (USENIX ATC)*, pages 101–114, Denver, CO, 2016.

[57] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. Characterizing the Performance of Intel Optane Persistent Memory: A Close Look at Its on-DIMM Buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys)*, pages 488–505, Rennes, France, 2022.

[58] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 323–338, Santa Clara, CA, USA, 2016.

[59] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*, pages 169–182, Santa Clara, CA, USA, 2020.

[60] Qirui Yang, Runyu Jin, and Ming Zhao. Smartdedup: Optimizing Deduplication for Resource-constrained Devices. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC)*, pages 633–646, Renton, WA, USA, 2019.

[61] Shawn Zhong, Chenhao Ye, Guanzhou Hu, Suyan Qu, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Michael Swift. MadFS: Per-File Virtualization for Userspace Persistent Memory Filesystems. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST)*, pages 265–280, Santa Clara, CA, 2023.

[62] You Zhou, Qiulin Wu, Fei Wu, Hong Jiang, Jian Zhou, and Changsheng 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)*, pages 187–202, Virtual Evenet, USA, 2021.

[63] Xiangyu Zou, Jingsong Yuan, Philip Shilane, Wen Xia, Haijun Zhang, and Xuan Wang. The Dilemma between Deduplication and Locality: Can Both Be Achieved? In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, pages 171–185, 2021.

[64] Pengfei Zuo, Yu Hua, and Yuan Xie. SuperMem: Enabling Application-transparent Secure Persistent Memory with Low Overheads. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, page 479–492, Columbus, OH, USA, 2019.

[65] Pengfei Zuo, Yu Hua, Ming Zhao, Wen Zhou, and Yuncheng Guo. Improving the Performance and Endurance of Encrypted Non-volatile Main Memory through Deduplicating Writes. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 442–454, Fukuoka, Japan, 2018.

A Artifact Appendix

Abstract

The artifact provides implementation source code and evaluation scripts of GogetaFS. GogetaFS is a fast deduplication file system that leverages a novel LFP mapping to minimize extra deduplication metadata I/O overheads.

Scope

The artifacts mainly including two aspects: (1) Preparation for reproducing (e.g., environments and code source), and one-click command to run experiments, draw figures, and build latex tables. (2) Detailed reproducing steps of each Table/Figure presented in the paper and the calculation details of several metrics (e.g., read/write bytes per block).

Contents

Artifact organization. The artifacts contain six parts: (1) GogetaFS source code based on NOVA (for PM), (2) GogetaFS source code based on F2FS (for SSD), (3) Evaluation tools and scripts on PM, (4) Evaluation tools and scripts on SSD, (5) NV-Dedup source code, and (6) f2fs source code. All the corresponding test scripts and results for reference (in the "paper" directory) are included in the "tests" directory.

Results organization. The raw output files are mostly named with the prefix "performance-comparison-table" or "xx-table", which can be obtained by running `bash test.sh`. `plot.ipynb` and `craft-latex-table.py` scripts are provided for drawing figures and building latex table, respectively. Some tables require further calculation on raw output, such as calculating extra read/write bytes per block in Table 2. Thus, we provide `process.py` script to automatically process the raw output files. Each experiment can be conducted many times just by passing a `loop` variable to the `test.sh`, and a `agg.sh` script is provided to present the average values. We rename the original file with the suffix `_orig`.

Hosting

The artifacts are available at github. To reproduce the experiments, the user should get the repositories listed below:

- GogetaFS source code for PM: https://github.com/GogetaFS/Light-Dedup-J.git.
- GogetaFS source code for SSD: https://github.com/GogetaFS/GogetaFS.git.
- Evaluation tools and scripts on PM: https://github.com/GogetaFS/GogetaFS-AE.git.
- Evaluation tools and scripts on SSD: https://github.com/GogetaFS/GogetaFS-Tests.git.
- NV-Dedup source code: https://github.com/GogetaFS/nv-dedup.git.
- f2fs: https://github.com/GogetaFS/f2fs.git.

The user should run the following command to organize them so that the scripts can work correctly:

Listing 1: Commands for cloning and orgnizing essential files.

```
#!/bin/bash
cd <Your directory>
git clone https://github.com/GogetaFS/GogetaFS-AE.git
cd GogetaFS-AE

git clone https://github.com/GogetaFS/Light-Dedup-J.git
git clone https://github.com/GogetaFS/nv-dedup.git

mkdir SSD-emu && cd SSD-emu
git clone https://github.com/GogetaFS/f2fs.git
git clone https://github.com/GogetaFS/GogetaFS.git
git clone https://github.com/GogetaFS/GogetaFS-Tests.git
```

We describe the branches of Light-Dedup-J, NV-Dedup, and GogetaFS that correspond to the paper in Tables 4, 5, and 6.

Requirements

Experimental environment. Intel Xeon Gold 5218 CPU clocked at 2.3GHz, 512 GiB of Optane DCPMM (2×256 GiB DIMMs) in non-interleaved AppDirect Mode, and 128 GiB of DRAM (4 × 32 GiB DIMMs). The CPU has 16 cores (32 hyperthreads) and 22 MiB of L3 cache with CLWB support. The server runs CentOS Stream 8 with kernel 5.1.0 modified by SplitFS (at https://github.com/rohankadekodi/SplitFS-5.1). All experiments are run with `data_cow` enabled.

Branch	Description
nova-pipe	NOVA file system, the baseline
nova-pipe-failure	NOVA with failure recovery
denova	DeNova file system
denova-non-crypto	DeNova w/o crypto hash
nova-pipe-dedup	Light-Dedup file system
nova-pipe-dedup-failure	Light-Dedup with failure recovery
lfd-64bit-fp	GogetaFS with non-secure mode
lfd-64bit-fp-failure	GogetaFS with failure recovery
lfd	GogetaFS with secure mode
lfd-disable-dedup	GogetaFS without deduplication
lfd-super	GogetaFS with SFP
lfd-regulate	GogetaFS with regulated memory
lfd-pm-table	GogetaHybrid
lfd-pm-table-persistence	Light-Dedup with mem regulation
lfd-pm-all	GogetaSHT (Figure 8e)
lfd-pm-all-persistence	SHT, GogetaSHT without LFP

Table 4: **Branches in Light-Dedup-J repository**. *Note that lfd is short for light-fs-dedup.*

Branch	Description
master	NV-Dedup with the original implementation
non-crypto	NV-Dedup without using crypto hash

Table 5: **Branches in NV-Dedup repository**.

Branch	Description
main	GogetaFS for SSD
lightdedup	Light-Dedup ported to SSD
hfdedup	HF-Dedupe reproduced on SSD
SmartDedup	SmartDedup reproduced on SSD

Table 6: **Branches in GogetaFS repository**.

Hardware prerequisite. (1) *Server with PMs*. A server with at least one PM equipped (256 GiB). (2) *Sufficient memory*. Reproducing experiments requires much memory because caching the compiled Linux kernel source code (i.e., the CP workload) requires about 15–20 GiB (depending on the configuration), and the operating system itself and GOGETAFS all require significant memory usage.

Reproducing Experiments

One-click command. We have provided a one-click script `run_all.sh` in the root directory of "tests", which can automatically install the required software, run all the experiments involved in the paper, draw all the figures in our paper, and build similar Latex tables presented in our paper.

Reproducing details. Generally, typical workflows for reproducing figures and tables are presented as follows.

Listing 2: General workflow to reproducing tables.

```
cd GogetaFS-AE/tests/TABLExx/
# Step 1. Run Experiment
bash ./test.sh $loop
# Step 2. Aggregate Results
bash agg.sh "$loop"
# Step 3. Calculation on Raw output
if [ -f "process.py" ]; then
    python3 process.py
fi
# Step 4. Building Table
python3 craft-latex-table.py
```

Listing 3: General workflow to reproducing figures.

```
cd GogetaFS-AE/tests/FIGxx/
# Step 1. Run Experiment
bash ./test.sh $loop
# Step 2. Aggregate Results
bash agg.sh "$loop"
# Step 3. Drawing Figures
ipython plot.ipynb
```

Reproducing Z-SSD results. We reproduce the Z-SSD results using `git@github.com:MoatLab/FEMU.git` environment; we install a Ubuntu 20.04 LTS image with the kernel version 5.4.0-189-generic, and we modify `femu/build-femu/run-nossd.sh` to run the Z-SSD experiments:

```
#!/bin/bash
# Huaicheng Li <huaicheng@cs.uchicago.edu>
# Run FEMU with no SSD emulation logic, (e.g., for
    SCM/Optane emulation)

# Image directory
IMGDIR=$HOME/
# Virtual machine disk image
OSIMGF=$IMGDIR/ub20.qcow2

if [[ ! -e "$OSIMGF" ]]; then
    echo ""
    echo "VM disk image couldn't be found ..."
    echo "Please prepare a usable VM image and place
        it as $OSIMGF"
    echo "Once VM disk image is ready, please rerun
        this script again"
    echo ""
    exit
fi

# enable performance mode
echo performance | sudo tee /sys/devices/system/cpu/
    cpu*/cpufreq/scaling_governor

sudo ./qemu-system-x86_64 \
    -name "FEMU-NoSSD-VM" \
    -enable-kvm \
    -cpu host \
    -smp 36 \
    -m 32G \
    -device virtio-scsi-pci,id=scsi0 \
    -device scsi-hd,drive=hd0 \
    -drive file=$OSIMGF,if=none,aio=native,cache=none
        ,format=qcow2,id=hd0 \
    -device femu,queues=64,devsz_mb=32768,id=nvme0 \
    -net user,hostfwd=tcp::2333-:22 \
    -net nic,model=virtio \
    -nographic \
    -qmp unix:./qmp-sock,server,nowait
```

Inside FUME, using `scp` to copy `GogetaFS-AE/SSD-emu/*` to the VM, supposing the directory is `/home/user/SSD-emu`. Run the following commands to reproduce the Z-SSD results:

```
cd /home/user/SSD-emu/GogetaFS-Tests
bash run_all.sh
```

Detailed reproducing steps. The detailed reproducing steps of each experiment are in https://github.com/GogetaFS/GogetaFS-AE.