MedFS: Pursuing Low Update Overhead via Metadata-Enabled Delta Compression for Log-structured File System on Mobile Device

Chao Wu and Cheng Ji, *Nanjing University of Science and Technology;*
Li-Pin Chang, *National Yang Ming Chiao Tung University;* Zongwei Zhu, *University of Science and Technology of China;* Congming Gao, *Xiamen University;* Weichao Guo and Chao Yu, *Guangdong Oppo Mobile Telecommunications Corp., Ltd;*
Yanzhi Wang, *Northeastern University*

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

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

MedFS: Pursuing Low Update Overhead via Metadata-Enabled Delta Compression for Log-structured File System on Mobile Device

Chao Wu[1], *Cheng Ji*[1]*, *Li-Pin Chang*[2], *Zongwei Zhu*[3], *Congming Gao*[4]*, *Weichao Guo*[5], *Chao Yu*[5], *Yanzhi Wang*[6]
[1]*Department of Computer Science and Engineering, Nanjing University of Science and Technology*
[2]*Department of Computer Science, National Yang Ming Chiao Tung University*
[3]*Suzhou Institute for Advanced Research, University of Science and Technology of China*
[4] *School of Information Science, Xiamen University*
[5]*Guangdong Oppo Mobile Telecommunications Corp., Ltd*
[6]*College of Engineering, Northeastern University*

Abstract

The increasing deployment of data-intensive applications on mobile devices poses a formidable challenge in designing flash-based file systems tailored to these needs. Studies have shown that adopting delta compression in log-structured file systems is promising for such an environment as it can effectively reduce the write stress and improve flash longevity. Unfortunately, delta compression suffers from large maintenance overhead. While prior works have introduced non-volatile memory buffer or battery-backed DRAM to mitigate this, they are less appealing for cost-sensitive mobile devices.

This paper introduces MedFS, a Metadata-enabled delta compression on log-structured File System for mobile devices, to achieve a good design trade-off in log-structured file systems employing delta compression. Through a comprehensive analysis of mobile applications and file update patterns, we develop *delta-inlining* technique, which consolidates delta updates within the inline area of the file's inode block. By leveraging the inherent inode structure and automatically flushing dirty inodes to storage, we effectively address the maintenance overhead associated with delta compression. Additionally, we propose a complementary delta maintenance strategy that selectively manages delta chunks in the data area, overcoming the space constraints of the inline area. Experimental results show that MedFS significantly reduces the write traffic by 55.1% on average, leading to the prolonged storage endurance by 122.7% and improved I/O performance by up to 37.3% over existing work. [1]

*Corresponding authors: Cheng Ji, Congming Gao, Email: cheng.ji@njust.edu.cn, gaocm@xmu.edu.cn. This work was supported in part by the National Natural Science Foundation of China (Grant No. 62106146, 62102179, and 62102219). This research is also partially supported by the National Science and Technology Council, Taiwan, (Grant No. NSTC 113-2221-E-A49-188-MY3) and the Fundamental Research Funds for the Central Universities, No.30923010933 and 20720230071.

[1]The source code and benchmarks are available at https://github.com/juanhair/MedFS.

1 Introduction

The fast technological advances in recent years have enabled the deployment of traditional desktop applications, e.g., video and photo streaming/editing, and social media, in modern flash-based mobile systems. Such applications often generate a large number of small files and updates while demanding short response time, making it critical to design the corresponding file system for achieving a good trade-off among I/O performance, energy consumption, and flash storage lifetime [3, 23, 53, 74].

Schemes have been proposed to optimize the file system (FS) performance on mobile systems. F2FS [36], an LFS (Log-structured File System) -based file system, keeps the updates in logs to convert random writes to sequential ones, which helps to mitigate write amplification and achieves superior I/O performance over traditional JFS (Journaling File System) [50]. To support small files, F2FS keeps an inline area in the file inode such that a file, if smaller than 3.69KB, can store its data in it [36, 65]. However, the applicability of this technique is limited. Based on our analysis of real mobile devices in Tab. 1, only about 25% of all files are eligible for this optimization.

Data compression is another approach that can effectively reduce the number of I/O writes to flash devices by compressing several pages into one [11, 13, 21, 33, 55, 58, 60, 86]. Adopting conventional compression methods, e.g., LZ4, achieves better compression ratios on large files [63, 70] but tends to be less effective on small files [9, 38] and incur large compression/decompression overheads [63, 70]. Recent works achieve a better trade-off between high compression ratio and low overhead by adopting delta compression [5, 16, 29, 66, 83, 84], which only maintain the content difference (XOR) between the new data and the old data. Delta compression exhibits high compression efficiency when data are updated with few content differences. Prior work showed that 70% of updating data in mobile devices are updated with less than a 10% content difference [29].

Instead of focusing on space savings, we exploit delta com-

pression to reduce write stress and enhance flash longevity. However, delta compression requires managing both delta chunks (XORed values) and base pages (original data). Without careful design, it can introduce significant overheads, including additional reads of base pages and deltas, as well as increased metadata accesses, such as references to delta pointers. Storing delta chunks in an external buffer, e.g., NVRAM [4, 29], or battery-supported DRAM [66], helps to mitigate flash writes but leads to increased hardware cost, which is less appealing for cost-sensitive mobile devices.

In this paper, we propose MedFS, a Metadata-enabled delta compression-based approach on log-structured File System, to address the above challenges. Tiny deltas are embedded into file system metadata, such as inodes, or multiple deltas are packed into fewer data blocks to decrease the block write count for file updates. Firstly, by exploiting the I/O behaviors of mobile applications, we develop a *delta inlining* technique to save delta chunks in the inline area of a file's inode block. Because file updating in LFS makes inode dirty, embedding small deltas to the inode inline space may only require rewriting the inode, avoiding data block writes for updates and therefore reducing write I/Os. Secondly, constrained by the limited inline area space, e.g., 3.69KB in F2FS, we develop effective delta chunk management policies to exploit its design potential, in particular, for those deltas that cannot be stored in the inline area, they are packed into fewer data blocks. Both the two techniques help decrease the block write count required to handle file updates and thus enhance flash longevity. To summarize, we make the following contributions.

- We study mobile I/O workloads and their behaviors in log-structured file systems. We then analyze the pros and cons of adopting delta compression in mobile devices.
- We develop DCI (Delta Chunk Inlining), a robust metadata-enabled delta compression technique, which leverages the unused inline area to achieve aggressive compressing efficacy for reducing write I/Os with marginal system overheads.
- We develop DCM (Delta Chunk Maintenance), a file hotness-based management method for handling delta chunks and updates that cannot be saved in the inline area. DCM saves compressed delta chunks and updates to the compact data pages, which effectively mitigates the latency in handling I/O writes.
- We prototype MedFS in a real mobile system and compare it with existing approaches [23,36]. Our experimental results show that MedFS substantially reduces the write volume by 55.1% on average, improves the average write and read I/O performance by 28.8% (up to 37.3%) and 25.3% (up to 35.6%), respectively, and prolongs the storage endurance by 122.7%.

To the best of our knowledge, MedFS is the first scheme that implements delta compression on mobile devices that mitigates read/write amplification without extra hardware to enhance file system performance and prolong flash longevity.

2 Background

2.1 File System on Mobile Device

JFS (Journaled File System), e.g., EXT4, records updates in the journal and updates files directly, which generates random I/Os and degrades I/O performance [25, 50, 52]. In contrast, LFS (Log-structured File System) records random updates in sequentially organized log entries to retain the sequential I/O performance [15, 64]. Flash Friendly File System (F2FS) is a typical LFS, which has been widely adopted in commercial mobile devices due to its performance advantages [23, 36, 44, 78]. The logical address space in F2FS is divided into multiple zones. Each zone consists of several segments, which is further divided into hundreds of blocks. The granularity of read/write operations is block, while segment cleaning is maintained to reclaim invalid blocks [71]. In F2FS, each block is 4KB, matching the size of each page in the page cache. For the sake of simplicity in this paper, we use 'page' to denote the fundamental unit of I/O.

Inode and Inline Area. The file structure in F2FS encompasses three node types: **inode**, direct node, and indirect node. F2FS designates a 4KB page for the inode, with approximately 3.69KB of space allocated to the inline area, while the remaining space is utilized for storing metadata, direct/indirect node information, and node footer. Small files ($< 3.69KB$) store their data content within the inline area, while larger files maintain the indices referring to the data pages within it. Considering the page size as 4KB, a file can maintain all its page indices within the inline area if the file size is smaller than 923 $\times$ 4KB = 3.69MB, allowing for a maximum of 923 4-byte indices in the inline area. Both of these methods aim to enhance the file system's efficiency by enabling direct access to more data content or page indices from the inline area, thereby accelerating I/O processing, rather than traversing through direct or indirect nodes. Meanwhile, designers may choose to configure the file attribute field ($Xattr$) in the inline area, which takes a space of 200-byte.

2.2 Storage Endurance on Mobile Device

On mobile devices, storage endurance is a critical issue with respect to the device lifetime. Since NAND flash only supports limited programming/erasing cycles (P/E cycles), storage endurance is directly impacted by the write volume. During past years, plenty of works have been devoted to prolong the storage endurance [10, 26, 49, 69, 72, 77, 85]. According to [2], the storage lifetime influenced by the write volume of data compression solutions can be calculated by Eq. 1, where WA and OP are respectively write amplification and over-provisioning factors, PEC is the P/E cycles of each flash block, and $R_{compress}$ is the compression ratio. The storage lifetime is inversely proportional to the number of full disk writes

per day (DWPD) which depends on the amount of data written. Thus, from the perspective of the host system, the write traffic release which is determined by DWPD and $R_{compress}$, is crucial for mobile storage endurance.

$$Lifetime = \frac{PEC \times (1+OP)}{365 \times DWPD \times WA \times R_{compress}} \quad (1)$$

2.3 Data Compression in File Systems

Compression techniques, such as LZO [23], LZ4 [13,43], and Zlib [51], have been extensively investigated due to the effectiveness in write traffic reduction. However, these methods exhibit reduced effectiveness on mobile devices due to their high computation overhead [23,48]. Thus, recent works have delved into delta compression techniques to meet the demand for high compression ratios and minimal system overheads in mobile devices [1,5,16,83,84].

Delta compression operates by only maintaining the XORed data between updated and original data, named delta. The delta is typically quite small (e.g., only a few dozen bytes for a 4KB page) and can alleviate the I/O traffics. However, this approach also introduces additional write/read amplification due to the maintenance of original page, delta chunk, and their respective indices. Specifically, during the write path, the indices linking the original page and the delta chunk must be persisted to storage. On the other hand, on the read path, besides the delta chunk, the indices and the original page should be retrieved as well. Each delta chunk needs to be assigned an in-page offset and delta size. Hence, the additional space overhead increases when the differences between the updated data and the original data are discrete and non-continuous.

2.4 The I/O Behaviors of Mobile Workloads

With the increasing deployment of data-intensive applications on mobile devices, their excessive I/O requests frequently contend for limited hardware resources, resulting in increased I/O latency and degraded system performance [17, 80]. In particular, the large performance impact on block read I/Os leads to a worse user experience [54], and frequent I/O writes consume the flash storage lifetime rapidly.

Further studies of mobile workloads have revealed that small I/O requests (e.g., <2MB) constitute 50% or more of I/O operations [23, 24]. This contrasts sharply with desktop or server workloads, where large I/O requests dominate. Consequently, optimizations on traditional FS demonstrate diminished efficacy on flash-based mobile systems. For instance, compressing small I/O updates randomly in JFS results in storage space fragmentation. Similarly, FPC imposes strict constraints on compression ratios, such as compressing five updating pages into at most one page [23]. A signature of file usage in mobile devices is the heavy involvement of multimedia files, which are typically larger than other files. However, they do not dominate the write traffic because they are write-once and read mostly [24]. By contrast, we observe that other files, e.g., SQLite files, and temporary files, receive intensive updates, which can be optimized by our MedFS. Specifically, mobile apps store small objects like cookies and history in SQLite files, whose updates trigger eager data flushing through frequent fsync() calls [22]. Updating temporary files and even accessing immutable multimedia files generates small file updates and history logs. These small changes are a perfect target for write stress reduction.

3 Empirical Study on Delta Compression

In this section, we study delta compression applied to different mobile workloads, elaborate on its limitations, and motivate our design. For discussion purposes, we denote *page* as the minimum read/write unit in both page cache and file system.

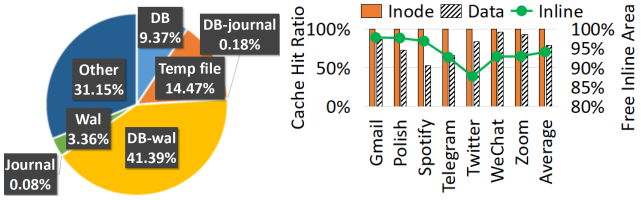
The importance and limitation of delta compression. To study the effectiveness of applying delta compression in mobile systems, we evaluated 17 popular user actions/tasks on seven popular applications. On average, we observe that there are 90% of the files generated/updated in modern applications are smaller than 3.69MB, those that are smaller than 3.69KB account for a relatively small percentage, i.e., around 25%. The details are summarized in Tab. 1. We further find that the 77.1% of the total write traffic comes from file updates, while the average update difference between modified and original page is only 13.8%, which exposes the great potential for applying delta compression to mobile workloads.

We analyzed the file updates in the tasks shown in Tab. 1 and summarized the results in Fig. 1 (a). From the figure, there are two major types, i.e., updates to SQLite files and updates to temporary files, which account for 54.4% and 14.5% of all files, respectively. The SQLite files include db, db-journal, db-wal, wal, and journal. The temporary files include the files starting with todelete_ or random characters, and the files in temp/tmp directories. These files are closely coupled with popular user actions/tasks such that writing a large number of small deltas tends to have a critical impact on user experience. For example, when editing a photo in Polish, 39.6% temporary files are generated to record the intermediate editing images. When launching applications, 88.1% of their created files are related to SQLite. Notably, other file types may also be suitable for delta compression in certain apps. For Polish, files including xml, json, etc, account for 54.3%. For Telegram, files including dat, xml, etc, account for 69.3%.

Delta compression may incur additional maintenance overheads, such as extra I/Os for accessing the base data page and its associated delta page during file reads and writes, as well as increased space requirements for delta management metadata. Storing delta chunks in an external buffer helps to mitigate flash writes but increases hardware cost, which is less appealing for mobile devices.

Table 1: I/O update (overwrite) characteristics of typical mobile application scenarios. *Vol* is the volume of write traffic in MB, Small is the percentage of files smaller than 3.69MB, *UR* is the file updating ratio in total write traffic, and *UD* is the average updating difference between modified and original page data.

App	Task	Vol	*Small*	UR	UD
Gmail	Launch application	0.7	100.0%	62.4%	31.2%
	Send 1 image	6.3	94.8%	89.6%	24.3%
	Send 100 characters	9.0	92.4%	91.5%	25.3%
	Receive 1 image	12.3	89.1%	71.1%	30.2%
	Receive 100 characters	7.2	98.6%	62.8%	29.0%
Polish	Launch application	0.1	100.0%	78.6%	0.9%
	Edit a photo	130.6	99.9%	51.8%	2.7%
Spotify	Launch application	4.4	100.0%	93.1%	5.5%
	Listen to music 5min	67.4	100.0%	62.3%	11.2%
Telegram	Send 1 image	2.4	52.9%	85.1%	5.2%
	Send 10sec voice	3.4	88.1%	92.7%	3.6%
	Receive 100 characters	4.8	64.4%	95.1%	4.1%
	Receive 1 image	3.5	51.7%	84.4%	1.6%
	Receive 10sec voice	4.7	72.1%	86.9%	2.5%
Twitter	Launch application	9.2	100.0%	61.4%	25.2%
	Post 100 characters	12.5	100.0%	92.9%	11.1%
	Post 1 image	15.8	100.0%	84.5%	8.3%
	Watch news 5min	117.5	99.2%	75.6%	14.4%
Wechat	Launch application	2.0	97.0%	67.1%	14.4%
	Send 100 characters	2.3	94.6%	75.9%	21.7%
	Receive 100 characters	1.8	97.7%	76.5%	22.3%
	Receive 10sec voice	2.8	85.4%	55.3%	9.2%
	Post 1 image	0.7	92.1%	61.2%	14.7%
Zoom	Attend 5min meeting	207.4	82.0%	92.5%	13.1%



(a) File types. (b) Hit Rate and Inline Area Utilization.

Figure 1: File types and opportunity for delta-inlining.

The motivation. To tackle the aforementioned design challenge, we propose **delta-inlining**, a technique that leverages the observation that most I/O updates are small and maintains deltas in the file's inline area. We studied the opportunities of maintaining deltas in the inline area and summarized the results in Fig. 1 (b). The vertical right axis indicates the average free inline area of files generated in the experiments. The results indicate that delta-inlining is feasible while the majority of files are smaller than 3.69 MB and make the inline area space underutilized (94.0% inline area of involved files is free), leaving available inline area for storing the deltas. Moreover, since maintaining deltas in an inline area can be accomplished alone with its inode update (e.g., logical block address (LBA) modification [36], Dirty Inode Flag Update [36]), the delta compression process can be realized with minimal management overhead.

The vertical left axis in Fig. 1 (b) indicates the cache hit rate of file inodes and data. The hit rates of the file inodes are very high, i.e., up to 100% and on average 99.97%. The high hit rate is due to the fact that the file's inode is always accessed before any other file operations (such as read or write). Once the inode is retrieved by the first file operation, the following file operations have high probability to hit the inode in the cache. Based on this, if deltas are maintained in the inode, they can effectively leverage the hit rates to reduce the maintenance overhead. Moreover, despite the hit rate of data pages is relatively high (76.63%), most updated data pages (more than 50.0%) cluster in a small LBA range and have a relatively low hit rate. Hence, updated data still dominate the write pressure on mobile device.

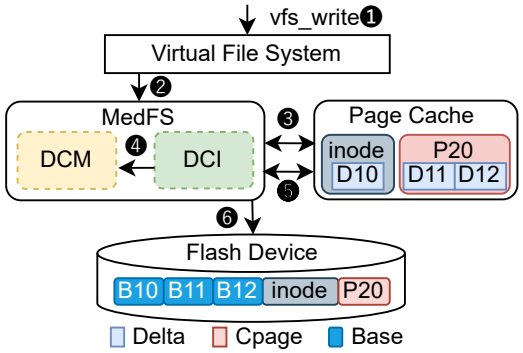


Figure 2: Architecture of MedFS.

4 The MedFS Design

In this section, we present the design details of MedFS. Instead of simply adopting delta compression in F2FS, MedFS seamlessly combines the features of delta compression and F2FS to reduce write traffic with marginal overhead [2]. Specifically, MedFS leverages the out-of-place nature of LFS to reduce the write amplification incurred by delta compression, and the file inode organization for delta maintenance.

Fig. 2 presents an overview of MedFS. It consists of two enhancements: Delta Chunk Inlining (DCI) and Delta Chunk Maintenance (DCM). The former manages the generation of delta chunks in response to update requests, while the latter aims to capitalize on the advantages of delta compression and leverage the data pages to further reduce I/O traffics. In detail, when receiving update requests from the virtual file system (❶) that modify chunks (C10, C11, and C12), MedFS activates DCI (❷) to retrieve the base chunks (B10, B11, and B12) from the flash if they miss in the cache to generate deltas (D10, D11, and D12) (❸). If sufficient unused space exists in the inline area, DCI places all delta chunks there. However, if space is insufficient, as many delta chunks as possible are stored in the inline area, for example, D10 (❹). The remaining delta chunks (D11 and D12) are then passed to DCM (❺), which compacts these chunks into one data page (P20) in the

[2] MedFS could be developed on top of prevalent LFS to be deployed on commercial mobile devices, here we take F2FS as an example.

page cache. Finally, P20 and the inode are synchronized with the storage (❻).

MedFS perceives the system idle status in FS to reduce the overheads of DCM. As such, MedFS is implemented in the file system instead of other layers, e.g., SQLite. In the following discussion, *Base* and *New* refer to the original data page and the updated data page, respectively.

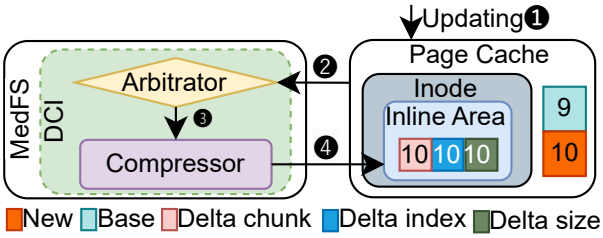


Figure 3: Workflow of DCI.

4.1 Delta Chunk Inlining (DCI)

In MedFS, DCI conducts delta compression at the same 4KB page size, which is consistent with F2FS. For the update operation, MedFS compresses the XORed value between the *Base* and *New* into a single delta chunk using a conventional compression algorithm (e.g., LZO). In decompression, the delta is first decompressed from the inline area, followed by an XOR operation with its *Base* page. Unlike traditional delta compression methods [1, 73], since the delta chunk is maintained in the inline area and the *Base* page remains valid and accessible via the original LBA, no additional address information is required to be stored in MedFS. Moreover, because the content is largely identical between the new and base pages, their XOR results predominantly contain zeros and compress really well.

4.1.1 Workflow of DCI

Fig. 3 depicts the process of DCI (Delta Chunk Inlining) during the write path. Upon receiving a file update request in the page cache (❶), the arbitrator retrieves not only the file's inode but also the *Base* pages from the page cache or flash memory. These *Base* pages are then forwarded to the arbitrator (❷) for delta computation, specifically the compressed XOR difference between the *Base* and *New* states. Based on the delta's size and the remaining space of the inline area, the arbitrator determines whether to store the delta chunk in the inline area ❸. If inline area space is sufficient and the delta is smaller than a given threshold (discussed in Sec. 6.3), the data chunk is identified as compressible chunk and its delta is stored in the inline area, along with a delta tag and their corresponding address and size (❹), as illustrated by chunk 10 in the figure. Otherwise, the Delta Chunk Maintenance (DCM) mechanism is invoked, as detailed in Sec. 4.2. Incompressible chunks (chunk 9 in the figure) are served as in existing LFS. Note that MedFS keeps *one delta* per 4KB page. Let a page be encoded by base B and delta D1. Upon update again, base B is compared against the update to create a new delta D2, which then replaces D1 in inline area without causing additional space overhead. Over successive updates, the delta gradually grows until incompressible.

In the read path, if the required *New* data is found in the page cache, it is immediately returned. Otherwise, the file's inode is read to examine the delta tag. If the tag value is true, indicating that the inline area holds the delta chunk corresponding to the *New* data, DCI reconstructs the *New* data after reading the *Base* data to fulfill the request. Otherwise, only the *Base* data containing the most up-to-date content is returned to the users.

Conventional delta compression requires additional I/O for writing deltas to flash storage, regardless of whether updates occur in the page cache or flash. In contrast, since the inode in MedFS is consistently marked as dirty during file updates, the delta chunks stored in the inode block can be flushed to flash alongside the file's inode without requiring additional I/O operations. Consequently, MedFS substantially diminishes write traffic, enhances system I/O performance, and extends the flash lifespan.

When to compress. Delta compression can be performed either while a data page is still in the page cache or when it is being flushed to flash storage. However, the latter tends to impact performance. This is because, in such cases, the original page (*Base*) corresponding to the updated page (*New*) may have been reclaimed, necessitating a read from external storage to perform delta compression. Therefore, MedFS prefers to execute delta compression while *Base* is still in the page cache. We integrate it in function `write_end()` in the file system layer.

4.1.2 Inline Area Organization

Given MedFS is built on top of F2FS whose inline area is less than 4KB, the delta chunks and their metadata inside the inline area should be choreographed. Figure 4 illustrates the organization of inline area. For files larger than 3.49KB (*Xattr* takes 200-byte) but smaller than 3.49MB, F2FS maintains the in-file page offset (i.e., page address) within the inline area (e.g., Data Offset Area in the figure). The remaining space in the inline area is then utilized to store delta chunks along with their metadata information. To optimize the packing of delta chunks, each chunk necessitates two additional fields to precisely determine its location within the inline area: the delta index (*c_addr*) and delta size (*c_size*). Given the varying sizes of compressed delta chunks, the *c_size* field is employed to denote the number of bytes, while the *c_addr* field indicates the page address of its *Base* page. Furthermore, given the potential file size increments due to append operations and the consequent expansion of the data offset area, MedFS dynamically manages and extends the delta zone – housing

delta chunks and their metadata – adjacent to the file's *Xattr* field, expanding from the tail of the inline area towards the head, counter to the expansion direction of the data offset area, as illustrated in the figure.

When a delta chunk undergoes subsequent updates, MedFS creates a new delta zone, expanding as necessary while obsolete delta zones are removed. In situations where space becomes insufficient but contains deleted delta zones, MedFS reorganizes these zones to free up contiguous space. Importantly, this space reorganization process takes place within the page cache, mitigating additional read/write amplification. To cope with the situation that the free inline area space is not enough to store more delta chunks, additional solution is further proposed and presented in Sec. 4.1.3.

The number of compressed delta chunks that can be maintained in the inline area depends on both the file size (i.e., data offset size) and delta zone size. To save a 2MB file, MedFS/F2FS stores file data in 256 data pages and thus allocates 256×4=1024 bytes as the data offset in the inline area, which leaves about 2.49KB of unused inline area. The *c_size* and *c_addr* fields take 1B and 4B, respectively. Based on our evaluation results (see Sec. 6), our delta compression achieves an impressive average compression ratio of 97.43%. This translates to compressing each updated page into a mere 106B. Consequently, allocating 109B per compressed delta chunk allows us to store up to 23 such delta chunks.

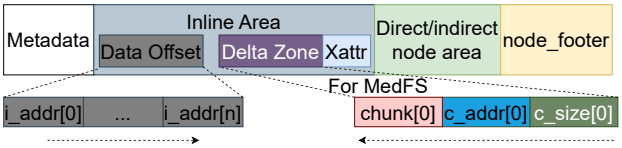


Figure 4: Inode layout of original FS and MedFS. While others are maintained by both, the delta zone is inserted specifically by MedFS. From tail to head in the inline area, delta size (*c_size*, each takes 1B to record <256Byte deltas) is placed besides *Xattr*; delta index (*c_addr*, each takes 2B) is inserted next; the delta chunk is at the end.

4.1.3 Complying with the space constraints

As DCI utilizes the inline area space to mitigate write pressure of update operations, a potential problem may arise if the free inline area space mismatches the space needed to accommodate all compressed delta chunks. For example, we assume there are 5 delta chunks, D1 to D5, and they already exhaust the inline area space. Thus, we might encounter one of the following three cases:

(1) **File Size Grows with New Write Operations:** As the file size grows during runtime, necessitating the allocation of page indices in the inline area, MedFS needs to evict some delta chunks for releasing the their Data Offset Area space. Therefore, MedFS adopts first-in-first-out eviction of delta chunks, such as delta chunks D1 and D2, to free up adequate Data Offset Area space for storing the page indices of new pages. MedFS performs delta chunk eviction by retrieving their *Base* pages, decompressing the delta chunks, reconstructing the data pages, and subsequently writing them back to the storage.

(2) **File Updates with New Update Operations:** If the file receives new compressed delta chunks, MedFS may replace delta chunks in the inline area by comparing the benefits of new delta chunks and existing delta chunks. For the benefit derived from delta chunk replacement, we assume each delta chunk occupies an average of α bytes in the inline area, and the average I/O latency for writing a single page is β. The benefit is calculated as $Ben = \frac{RD}{\alpha} \times \beta$, where *RD* represents the size of the replaced delta chunk in the inline area. Regarding the overhead incurred by writing the replaced delta chunk to flash, it can be expressed as $OH = HR \times \lambda + (1 - HR) \times \delta + \varepsilon + \beta$, where *OH* denotes the overhead of replacement; *HR* signifies the hit rate of *Base* in the page cache; λ represents the read latency of a hit base page in the page cache; δ reflects the latency of a miss base page and reading from flash; ε accounts for the decompression latency. Consequently, delta chunk replacement is deemed beneficial if *Ben* exceeds *OH*. Otherwise, the new updated data page is directly flushed to the storage. For example, from the devices under evaluation in Sec. 6.1, we have α=72B, β=954us, ε=6.9us, *HR*=74.1%, λ=9.2us, and δ=250us. Based on these, the size difference between the new and victim deltas *RD* should be larger than 77.9B for the replacement benefits to outweigh the overheads.

(3) **Updating an Existing Delta Chunk in Inline Area:** If an update to an existing delta chunk has a smaller delta chunk size, the existing delta chunk is directly replaced. Otherwise, if it results in a larger delta chunk size, MedFS abandons the delta-based compression and directly writes the page into the storage as well.

4.2 Delta Chunk Maintenance (DCM)

To adhere to the space constraints of the inline area, DCI may flush decompressed data pages (cases (1) and (2)) and/or new update pages (case (3)) into the flash during runtime, thus can not benefit from the advantages of delta compression. To mitigate this, a solution is proposed, which compacts delta chunks from inline area into a single data unit, labeled as *Compact* page to retain the benefit of delta compression. With the aid of the *Compact* page, the new update page (case (3)) also can benefit from the delta-based compression by storing its delta chunk into the *Compact* page.

However, as the *Compact* page is independent of the inode block, obtaining delta chunks stored in the *Compact* page may suffer from read amplification, especially for these read-intensive files. Therefore, we introduce a strategy named

DCM (Delta Chunk Maintenance), designed to capitalize on the advantages of delta compression while also reducing the extra I/O traffic. This is achieved by deciding whether to compact delta chunks into the *Compact* pages, based on the file access pattern. Specifically, for files with a write-hot-read-cold behavior, DCM prioritizes the creation of *Compact* pages to conserve write I/O traffic. For other types of files, DCM decides to flush data pages without applying compression to avoid read amplification, as read requests are more latency-sensitive compared to write requests.

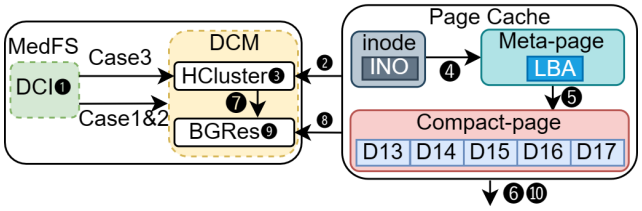


Figure 5: Workflow of DCM. Meta-page records the logical addresses of compact pages, and *Compact* page contains compressed delta chunks.

4.2.1 Workflow of DCM

Fig. 5 presents how DCM works. For delta chunks and update pages that cannot be processed in DCI ❶, DCM employs a hotness clustering (HCluster) component to determine if a file qualifies as a write-hot-read-cold file ❷. If it does (where a file's hotness is gauged by its read/write frequency tracked in the file inode ❸), DCM consolidates the delta chunks from new updates (case (3)) or evicted delta chunks from case (1) and case (2) into the *Compact* page. Otherwise, new updates retain their original page format, while evicted delta chunks are decompressed before being flushed to flash memory ❿. To locate the *Compact* page, DCM uses a meta-page to store the logical block address (LBA) of each *Compact* page and records meta-page's address in the inode block of a dedicated file (more details are presented in Section 5) ❹ Thus, when a *Compact* page is conducted, its LBA is updated in meta-page ❺. After the completion of DCM, the *Compact* page, meta-page, and the file's inode are flushed to the flash memory ❻.

DCM incorporates background data restoration (BGRes), akin to background cleaning [71], which activates during system idle periods. For files housing delta chunks within *Compact* pages, BGRes employs Hcluster to assess their hotness status ❼. If a file no longer write-hot-read-cold, DCM decompresses its delta chunks resided in the *Compact* page, forms the data pages in an uncompressed format ❽, marks them as dirty, and flushes them to the flash ❾❿.

Dynamic file hotness clustering (Hcluster). While the majority of current studies concentrate on clustering hot data at the page/block granularity access frequency [35, 40, 76], some opt for clustering based on semantic/program context [7, 31,32,36] or file types [62]. MedFS opts for file granularity for two primary reasons. Firstly, it is tailored for mobile systems and specializes in managing small files, where the disparity in hotness between different data pages tends to be minimal. Secondly, enhancing user experience in mobile systems is crucial, with file access latency being a critical determinant. In such scenarios, clustering hot data based on page/block granularity becomes irrelevant, as file access latency hinges on the longest access latency for accessing cold data.

The hotness cluster component, Hcluster, operates as an online algorithm designed to monitor file hotness. It records the read and write counts of each file, along with the elapsed time since the last invocation, as the inputs. These inputs are maintained within each file's inode and are incremented as a result of *vfs_read* and *vfs_write* calls. MedFS then computes the average read and write times within a specified time window and employs the K-Means clustering algorithm to categorize files into four distinct groups: read-hot-write-cold, read-cold-write-hot, read-cold-write-cold, or read-hot-write-hot [14, 28]. Notably, the updating of cluster centroids occurs off the critical path, specifically during BGRes when the system is idle, and all inputs are reset after the invocation.

Background data restoration (BGRes). BGRes focuses on monitoring file hotness and employs decompression of data chunks within *Compact* pages to alleviate the performance impact of handling read requests. To facilitate this, BGRes maintains a dual-linked inode list to keep track of files containing *Compact* pages, with the list head stored in the superblock. Within each inode's inline area, an additional inode number is stored to track the next inode in the list. During system idle periods, BGRes traverses the files in the list, determining the hotness status of each file. If a file is deemed read-hot, BGRes proceeds to decompress all delta chunks of the file, which are managed by DCM. Following decompression, the data pages are flagged as dirty and queued for flushing to the flash storage.

5 Implementation Remarks

The proposed approach is ready to be integrated as a compatible module in existing F2FS. In this section, we discuss issues arising in implementation.

I/O path. In MedFS, when fulfilling a data read request, MedFS first checks if the most up-to-date *New* page is present in the page cache; if not, it retrieves the *Base* page from flash. Subsequently: (1) If the page is uncompressed, MedFS retrieves the *Base* page from flash and returns it to users. (2) If the page is compressed, MedFS then fetches the delta chunk to reconstruct the *New* page.

For a data write request, the incoming data page is the most up-to-date *New* page and is marked as dirty. If compression applies, a delta chunk is generated and stored in either the inline area or the *Compact* page, depending on space availability. Otherwise, MedFS flushes the page to flash using a standard write operation.

D2D (delta chunks to data) mapping in DCI. DCI maintains the mapping between delta chunks and data in the inline area, as depicted in Fig. 4. Each delta chunk contains a *c_addr* field recording its in-file page offset, which corresponds to a location in the Data Offset Area within the inline area. And each *i_addr* in the Data Offset Area records the LBA of the *Base* Page. To retrieve a compressed data page, MedFS traverses the delta zones in the inline area to locate the specific delta chunk with the matching *c_addr*. Due to the small size of delta zones, the traversal overhead negligible.

D2D (delta to data) mapping in DCM. To recover a compressed data page, both *Base* and its delta chunk are needed, which are indexed by the LBA and the offset of the *Base* page in the inline area. Besides, if there exist delta chunks stored in the *Compact* page, the corresponding mapping stored in the meta-page should be retrieved as well.

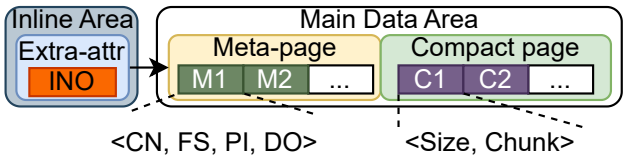


Figure 6: D2D mapping in DCM. Size is the delta size, Chunk is the delta chunk. CN is the number of delta chunks in the compact page, FS is the free space left on the compact page, PI indexes to the compact page, and DO records each delta offset in the compact page.

To index the meta-page, DCM creates a pseudo file to accommodate both the meta-page and *Compact* pages. The inode number of the pseudo file (INO), occupying 4 Bytes, is stored in the extra-attribute area located at the beginning of the inline area, as illustrated in Fig. 6. Within the meta-page, there exist multiple mapping pairs. Each pair includes the number of delta chunks in a *Compact* page (CN, using 1B), the available free space in the *Compact* page (FS, using 2B), the index of the *Compact* page (PI, using 4B), and the in-file offsets of the corresponding delta chunk in the *Compact* page (DO, each using 2B). In each compact page, all delta chunks are sequentially listed, along with the size of each delta (using 2B), positioned before the respective delta chunk. Consequently, when accessing the delta chunks within the *Compact* page, DCM utilizes the INO to pinpoint the specific file. Subsequently, all mapping pairs in the meta-page are traversed to verify the matched in-file offset, thereby determining the corresponding *Compact* page to retrieve the required delta chunk.

Data consistency. Delta compression might lead to data corruption since the delta and *Base* are maintained separately. If inconsistency happens between the delta chunk and its *Base* or the versions of compressed and uncompressed data in a file, the file might not be recovered if system crash occurs To avoid this, MedFS strictly follows the writing sequence of flushing file data ahead of delta chunks and finally inode for both DCI and DCM. DCI retains data consistency by ensuring that updated data is written to flash along with the dirty inode. For DCM, MedFS guarantees data consistency by flushing with the sequence of *Compact* pages, meta-pages, and file inode. Experimental results indicate that the percentages of meta-pages and *Compact* pages are small, consequently keeping low flushing overhead for DCM. In this manner, MedFS keeps data consistency via the current checkpoint mechanism even in extreme cases, e.g., system crash happens while file data and delta are flushed and inode is not, or none is flushed. The system could be recovered by reading the last version of file data, whose index is kept in the inode, as in existing LFS.

Another potential inconsistency happens in delta restoration of delta eviction or replacement in DCI (Case 1 or 2 in Fig. 5), and BGRes in DCM. If system crash happens when delta is evicted/replaced but the corresponding data is not restored, the data could be corrupted. MedFS addresses this issue by copying the target delta in memory. Only after the data is restored and flushed, the target delta will be evicted/replaced from the inline area for DCI, or *Compact* page for DCM. As such, if system crash happens before the data is restored, the system could be recovered since the target delta is not lost.

Segment Cleaning. To reclaim a victim segment in F2FS, the system conducts segment cleaning, a process that involves migrating valid blocks from the victim segment, updating the page's LBA of the migrated blocks in the inline area, and subsequently reclaiming the segment [36]. In MedFS, DCI exclusively updates the LBAs of corresponding delta chunks in the Data Offset Area, without altering the in-file offset of data pages. **Obsolete base pages are reclaimed during segment cleaning, which will not introduce extra burden as file updating generates the same number of obsolete pages.** However, DCM introduces additional overhead during segment cleaning by generating meta-pages and compact pages.In data block allocation, the corresponding bitmap entries of both pages will be assigned as valid ('1' in value, each 'block' in file system corresponds to a 4KB 'page' in the storage device). As such, segment cleaning process traverses and migrates all valid blocks in the victim segment to reclaim the segment, during which the migration of meta-pages and compact pages follows the procedure of migrating other pages. The migration overhead of both pages proves negligible, as we demonstrate in Sec. 6.4.

6 Evaluation

6.1 Experiment Setup

Experimental Environment. The experiments leverage a OnePlus 8T smartphone equipped with an 8-core Snapdragon 865 CPU, 8GB DRAM, and 128GB UFS3.1 2-lane flash storage. This device runs on LineageOS [46], featuring Android 14 and Linux kernel version 4.19. The default compression

algorithm employed in delta chunk generation is LZO [56]. Note that MedFS mainly relies on the out-of-place updating feature and inode organization of LFS, which are the key features of LFS and does not change with the versions of Linux. As such, MedFS could be implemented on existing Linux with different versions. MedFS is developed as a prototype built upon F2FS based on default file system settings including the default POSIX sync [36]. Moreover, although MedFS adopts XOR-based delta compression, repetition-based methods [1, 72] can be seamlessly integrated with MedFS as they share the common logic for delta management.

Table 2: I/O workload characteristics. *Vol* is the write volume to the storage disk in GB, *IO* is the number of I/O requests, *RW* is the percentage of write I/O in total, *CCR* is the compression ratio of a conventional compression method LZO, and *DCR* is the compression ratio of delta compression.

App	Vol	IO	RW	CCR	DCR
GM	3.12	60091	96.1%	37.7%	96.9%
PS	0.39	11391	94.8%	53.1%	97.3%
ST	0.27	33035	98.7%	28.2%	97.6%
TG	0.45	12400	65.6%	24.6%	98.4%
TW	0.88	19231	61.8%	30.0%	97.1%
WC	1.38	32805	75.5%	39.4%	98.2%
ZM	0.38	2041	72.7%	44.7%	96.5%

I/O Workload. MedFS is evaluated with a set of popular applications, including Gmail (**GM**), polish (**PS**), Spotify (**ST**), telegram (**TG**), Twitter (**TW**), WeChat (**WC**), and zoom (**ZM**). These applications are performed with various common user scenarios, including sending/receiving messages (**TG, WC**), listening to web music (**ST**), sending/receiving email (**GM**), editing photos (**PS**), watching web news ((**TW**)), and attending cloud meetings (**ZM**). During the experiments, each application undergoes a cold launch and is operated for 30 minutes, simulating a variety of user behaviors outlined in Table 1. In the evaluation phase, we track and gather the application's file operation workloads from `submit_bio()` underneath the cache, capturing details such as file name, inode number, operation type, timing, file content, and data offset. To ensure a fair comparison across different compression methods, we precisely replicate the same set of workloads through a user-level process, faithfully adhering to the sequence of each file operation, which is akin to the prior works [23, 79]. Table 2 provides a comprehensive overview of the I/O characteristics of our workloads. Notably, the predominant user behaviors in the selected applications exhibit a bias towards write-intensive operations, where delta compression emerges as a superior method compared to conventional approaches. Finally, to ensure statistical reliability, all experiments are repeated five times, and the average values are computed as the final results.

SOTA Works. We conduct a comparative analysis of MedFS against several existing approaches, including the original F2FS without data compression (referred to as **F2FS-NODC**) and the current F2FS with conventional data compression (**F2FS-DC**). Additionally, we establish **FPC** [23] as our baseline, representing a state-of-the-art (SOTA) technique that employs conventional compression methods to compress random I/O requests. It's important to note that SOTA delta compression-based methodologies [29, 66] are omitted from our evaluation. These methodologies rely on an external buffer cache for delta maintenance, a solution deemed impractical for mobile devices due to associated costs, size constraints, and complexities in I/O management.

Our comparison encompasses three distinct approaches: sole **DCI**, sole **DCM**, and **MedFS**. In the context of sole **DCI**, we promptly decompress and flush all evicted delta chunks upon occurrences of delta replacement or contention. Conversely, in sole **DCM**, we compress selected data pages and directly maintain delta chunks within the *Compact* pages, as elaborated in Section 4. Notably, the hyperparameter T in Hcluster is configured to 60 seconds, aligning with the triggering frequency of background segment cleaning [36].

Metrics. MedFS utilizes delta compression to mitigate the overhead of small-sized update operations. Consequently, the primary focus of MedFS is on improving performance and extending the system's lifetime, particularly in relation to update operations, rather than on saving storage space. To demonstrate these benefits, we first introduce the metric of Write Stress Release, which highlights the reduced write volume achieved by integrating updated data into either the inline area or *Compact* pages, thus minimizing the number of file updates, thus prolonging the storage lifetime. Next, we evaluate the user-perceived latency, which is reduced due to fewer write operations being sent to the storage disk, leading to a decrease in end-to-end latency. We then assess the I/O performance to showcase the advantages stemming from the reduced write traffic. Finally, we provide a detailed evaluation of the cache hit rate, Hcluster, and BGRes to further illustrate the strengths of MedFS.

6.2 Experimental Results

6.2.1 Write Stress Release

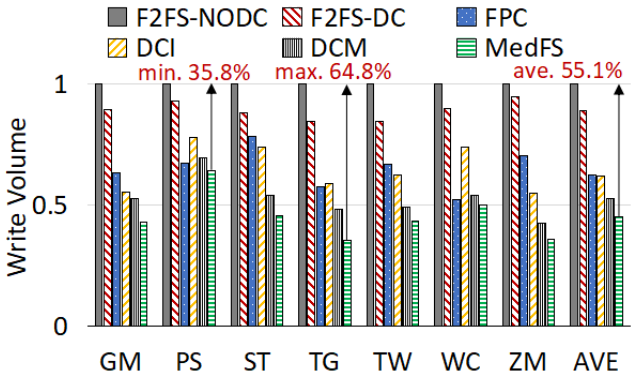


Figure 7: Normalized write volume of various approaches.

Fig. 7 illustrates the normalized write volume across vari-

ous approaches, relative to F2FS-NODC as detailed in Tab. 2. A lower value in the figure indicates superior performance, implying a reduction in write traffic. On average, MedFS reduces the write volume by 55.1%. At best, MedFS diminishes TG's writing stress by up to 64.8%. The benefits derived from MedFS are further elaborated in Fig. 1(a) and Tab. 1. In mobile workloads, SQLite and temporary files are prevalent, often undergoing frequent updates with minor content difference. MedFS effectively compresses these compressible files, predominantly SQLite and temporary files, achieving an exceptionally high compression ratio and thus significantly reducing write traffic. Both DCI and DCM are pivotal in alleviating writing stress. DCI, with an average write volume reduction of 37.9%, demonstrates a compression efficacy akin to FPC. However, the inline area space constraints limit DCI's ability to achieve aggressive compression. In contrast, DCM achieves a higher average write volume reduction of 47.4% over F2FS-NODC. Nevertheless, DCM's maintenance of delta chunks in data pages results in a lower hit rate compared to inodes, as depicted in Fig. 1(b). Given the considerable speed gap between DRAM and flash, this reduced hit rate may affect the system performance. MedFS capitalizes on the strengths of both DCI and DCM by predominantly maintaining delta chunks in the inline area while selectively retaining a few in the *Compact* pages, thereby enhancing compression efficiency.

In contrast, traditional compression methods exhibit suboptimal compression effects. F2FS-DC, for instance, reduces the write volume by only 10.9% on average compared to F2FS-NODC, as it exclusively compresses sequential data with a compression ratio exceeding 75%. FPC addresses this limitation by compressing random pages, resulting in an average reduction in write traffic of 37.6%. However, FPC remains constrained by the relatively modest compression ratios of conventional compression methods. Furthermore, MedFS is orthogonal to conventional compression methods like FPC. Thus, developers have the flexibility to employ conventional compression methods for appending file data while leveraging MedFS for updating file data, thereby achieving a more aggressive compression efficacy. This possibility merits further investigation as our future work.

In summary, MedFS significantly alleviates write stress on mobile devices by predominantly storing delta chunks in the inline area and selectively retaining them in the data area when inline resources are exhausted.

Storage lifetime: The alleviated write traffic facilitated by MedFS contributes to extending the lifespan of flash devices [6,8,30,45,84]. Flash devices inherently possess limited P/E cycles, thereby imposing a finite lifespan. Previous studies have consistently demonstrated the inverse relationship between the volume of written data and the longevity of flash devices [23]. As per Eq. 1 [2], the storage lifespan enhancement achieved by MedFS amounts to 122.7% (calculated as $1/(1-55.1\%)-1$) over F2FS-NODC, with 55.1% representing the average reduction in write volume. According to [68], a phone typically receives 131GB of writes per month. Now, consider a 128 GB storage based on QLC flash, which endures 1000 P/E cycles [61]. Ideally, the flash lifespan is approximately 81.4 years (1000*128/131/12). However, it can drastically reduce to 4.9 years, because the file system and the mobile storage firmware amplify the write volume by factors of 2 [39] and 8.27 [18], respectively. As the phone replacement cycle has lengthened in recent generations, ranging from 3 to 6 years for SAMSUNG smartphones [12,67], storage lifespan has become a significant concern, demanding immediate attention. With our MedFS, the lifespan can be restored to around 11 years (4.9*(1+122.7%)). Moreover, phone vendors avoid enabling swapping on flash storage, opting to use zRAM instead. Interestingly, they adopt an experimental feature to re-enable this feature [41,42], and the increased write stress from swapping can be compensated by the write reduction achieved by our MedFS.

6.2.2 User Perceived Latency

To study the impact of MedFS on user-perceived latency, we conducted evaluations across three typical scenarios: app launching (`GM` and `PS`), app switching from Earth to Facebook (`EA`) and vice versa (`FB`), and importing 100 contacts from SIM card to device (`IM`). In app switching, the former app was actively used for 5 minutes (browsing news on Facebook or viewing land-forms on Earth) to generate intensive I/O.

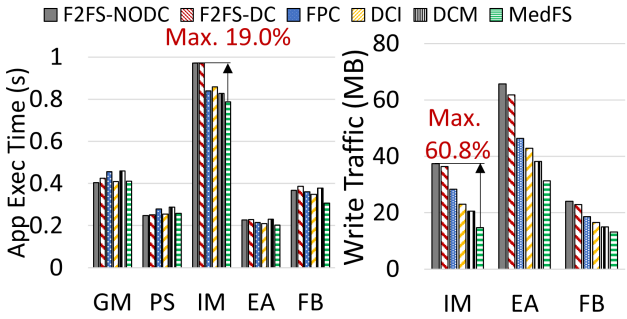


Figure 8: User perceived latency of various approaches.

As depicted in Fig. 8, by reducing write traffic, MedFS enhances user-perceived latency by an average of 7.9% compared to F2FS-NODC. In `IM`, the latency is predominantly influenced by SQLite file writing, constituting 90.2% of all I/O operations, with 88.2% dedicated to SQLite file writing. Consequently, MedFS significantly reduces `IM`'s write traffic by 60.8%, leading to a performance improvement of 19.0%. Similarly, `EA` experiences a latency reduction of 10.6% over F2FS-NODC, with SQLite file writing accounting for 86.3% of its write traffic. MedFS mitigates this issue by reducing intensive writes during app switching, decreasing it by 52.3%. Additionally, the latency of app switching in `FB` sees a 16.3% reduction. In Facebook, since temporary files, constituting 74.5% of write traffic, undergo frequent but minor updates, as

discussed in Sec. 3, MedFS effectively reduces FB's update traffic by 45.2%, thus reducing its switching latency.

Note that MedFS did not demonstrate performance improvements in app launching, with a slight increase of 3.0% on average for both GM and PS over F2FS-NODC. The latency during app launching is primarily influenced by I/O read latency, which could be affected by MedFS's decompression latency. This can overshadow the benefits of write traffic reduction. Alternatively, users could opt to avoid compressing critical I/O data, as in [23]. In summary, MedFS enhances user-perceived latency in scenarios characterized by excessive small file writings through aggressive data compression.

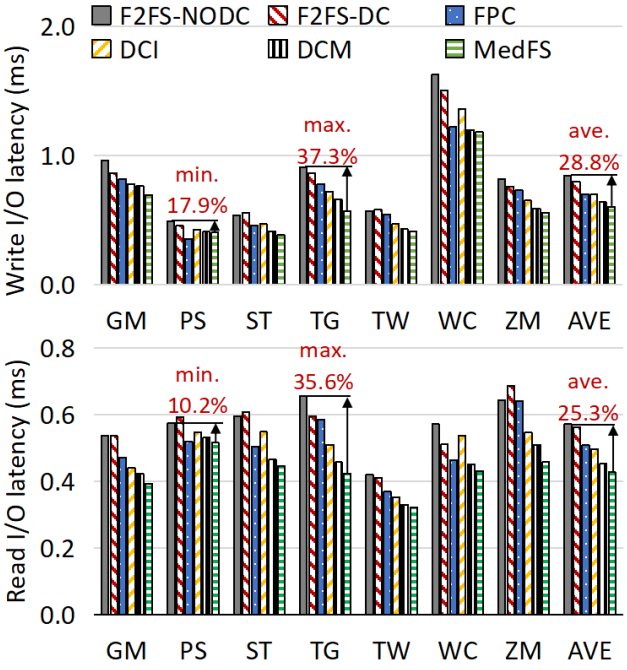


Figure 9: I/O latency of various approaches.

6.2.3 I/O Performance

The alleviated write stress on mobile devices has a positive impact on I/O performance due to the reduction in I/O traffic [45]. Fig. 9 provides an overview of the I/O latency across various approaches. MedFS demonstrates outstanding I/O performance, 25.3% faster on average for read I/Os and 28.8% faster for write I/Os compared to F2FS-NODC, stemming from the reduction in write volume. For example, MedFS reduces the read and write latencies of TG by 35.6% and 37.3%, respectively. The substantial reduction in TG's write traffic helps alleviate the contention between I/O requests. Despite MedFS introduces additional compression/decompression overhead as discussed in Sec. 6.4, the I/O performance remains unaffected, as delta compression occurs within the page cache rather than in the I/O path.

To further examine the performance advantages of MedFS, we also analyze the I/O performance of DCI and DCM, which are presented alongside. Compared to F2FS-NODC, DCI reduces the average latency of read I/O by 13.0% and that of write I/O by 17.5%. The achieved performance improvement comes from not only the reduced I/O traffic, but also the high cache hit rate (78.0%, as illustrated in Fig. 10). The enhanced cache hit rate of DCI potentially enhances system performance due to the significantly faster cache access speed compared to flash devices. On the other hand, DCM enhances I/O performance by 20.8% for read I/Os and 24.5% for write I/Os on average over F2FS-NODC. The superior I/O performance compared to DCI primarily stems from more aggressive write traffic reduction. However, as a trade-off, DCM cannot address the issue of low cache hit rates caused by delta compression, with an average cache hit rate of 59.9%, as detailed in Fig. 10.

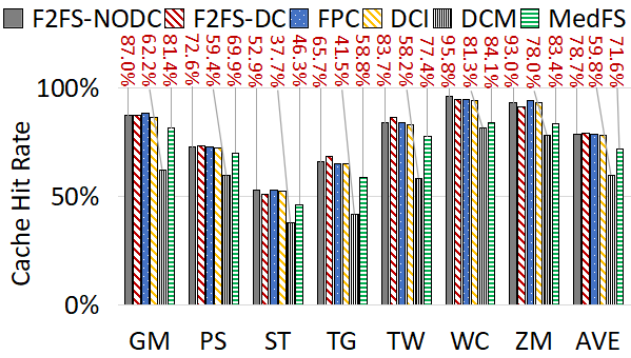


Figure 10: Cache hit rate of page data.

6.2.4 Cache Hit Rate

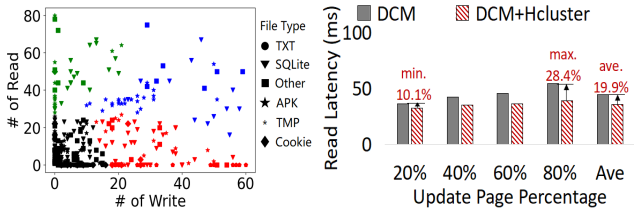
In this section, we assessed the hit rate of file data in the page cache, examining the cache hit rates of data pages, delta chunks, and meta-block for DCI, DCM, and MedFS, and presented in Fig. 10. As previously discussed, DCI retains delta chunks within file inodes, ensuring their presence in memory during file access. Consequently, the average cache hit rate of DCI closely mirrors that of F2FS-NODC, standing at 78.0% compared to F2FS-NODC's 78.7%. In contrast, DCM exhibits a lower cache hit rate, averaging 59.8%, as delta chunks are exclusively preserved in *Compact* pages, resulting in diminished cache hit rates.

MedFS seamlessly integrates DCI to embed delta chunks within the inode inline space and DCM to store delta chunks in *Compact* pages. Consequently, MedFS achieves an average page cache hit rate of 71.6%, much higher than that of using DCM alone. The improved hit rate is primarily attributed to inode hits enabled by DCI, as evidenced by the significant write reduction of 37.9% achieved by DCI alone compared to F2FS-NoDC, while MedFS achieves a total write reduction of 55.1%. However, storing deltas in *Compact* pages can negatively impact the page cache hit ratio. To address this issue, as described in Sec. 4.2, MedFS applies DCM to read-cold files only, effectively mitigating this drawback. As a

result, MedFS maintains a page cache hit ratio close to that of F2FS-NoDC (71.6% versus 78.7%) while delivering good user-perceived performance, as demonstrated in Fig. 8.

6.2.5 Evaluation on Hcluster and BGRes

To mitigate the overhead of maintaining delta chunks within *Compact* pages, MedFS introduces Hcluster in DCM. Fig. 11 (a) illustrates the clustering effect of files. We monitor the read/write times of files within each minute during a 30-minute period using GM and employ HCluster to classify file hotness into four categories. The results indicate that the hotness distribution of SQLite files across all clusters suggests the inefficiency of file type-based clustering [62]. In contrast, temporary files, which are a primary compression target in MedFS, exhibit read-hot behavior, with the majority clustering on the left side of the figure. That is, the adopted HCluster can well identify the file access behavior.



(a) Clustering effect. (b) File hotness changing.

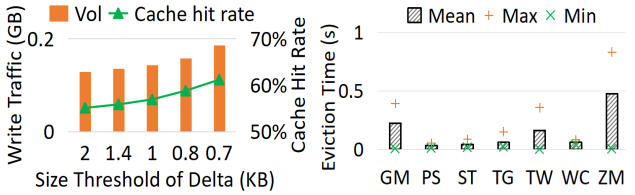
Figure 11: Effect of Hcluster. In (a), green points are read-hot-write-cold files, blue points are read-hot-write-hot files, red points are write-hot-read-cold files, black points are read-cold-write-cold files.

To evaluate the effectiveness of Hcluster on read performance, the FS is configured as DCM with and without Hcluster. Small files were generated to trigger DCM: a 1MB new file was generated and subsequently updated by multiple pages, each with minor content variations (ranging from 1 to 100 bytes) to facilitate compression. The proportion of updated pages was set at 20%, 40%, 60%, and 80% respectively. Subsequently, the file underwent random reads 1000 times to emulate a read-hot clustered state, and the average file read latency was recorded. The results are depicted in Fig. 11 (b). We can see that Hcluster contributes positively to enhancing file read performance. As the number of compressed pages within the file increases, the discrepancy in read performance between DCM with and without Hcluster becomes more evident. In the absence of Hcluster, DCM retains more hot delta chunks in *Compact* pages, resulting in a read performance decrease. In summary, maintaining delta chunks from read-hot files in *Compact* pages can detrimentally affect read performance, underscoring the necessity of Hcluster.

Furthermore, file hotness is subject to change based on user behaviors. For instance, as illustrated in Fig. 11 (a), the hotness of 56.4% of files changes over time. Consequently, MedFS introduces BGRes to periodically assess the hotness of compressed files within DCM and decompress read-hot files accordingly. Since BGRes is activated during idle period, its time cost can be ignored.

6.3 Sensitivity Study

MedFS demonstrates sensitivity to the size of delta chunks within both DCI and DCM. A smaller delta size corresponds to greater savings in write volume. Consequently, we have established a default delta size of 256 bytes, taking into account the constraints of the inline area space, to determine whether data pages should undergo compression for enhanced write volume reduction. This default value is chosen carefully to optimize space utilization within the inline area. If the delta size exceeds 256 bytes, it necessitates additional space to accommodate both the delta size and the delta chunk value, leading to inefficient usage of the inline area space. Therefore, in DCI, only a delta chunk smaller than 256 bytes can be maintained in the inline area.



(a) Sensitivity study. (b) Delta eviction time.

Figure 12: Sensitivity study and delta eviction time.

For DCM, `TG` is selected as the case study to investigate sensitivity. The findings are depicted in Fig. 12 (a), The x-axis represents the size threshold of compressed delta chunks, determined by the number of original pages that can fit into a *Compact* page, e.g., 0.8KB (819 bytes) corresponds to that a *Compact* page storing deltas from 5 pages. Several observations emerge. Firstly, there exists a trade-off between reducing write traffic and maintaining cache hit rate, influenced by the size threshold. A larger threshold results in more delta chunks in *Compact* pages, potentially exacerbating the read/write amplification issue. We set the threshold to 5 to achieve a Pareto-optimal balance between reducing write traffic and maintaining cache hit rate, although this may not always be the optimal choice. For instance, system designers could exclusively maintain evicted/replaced deltas from DCI within *Compact* pages for enhanced performance. Secondly, a larger size threshold does not necessarily ensure significantly improved compression efficacy. For instance, reducing the threshold from 3 to 2 only yields a 1.5% reduction in write traffic, whereas a reduction from 6 to 5 leads to a 6.1% reduction. This phenomenon arises because DCM maintains a meta-page to record mapping information for each file. In cases where files contain only two delta chunks, DCM fails to benefit the system with a size threshold of 2.

6.4 Overhead Analysis

Compress/Decompress Overhead. Delta compression offers a significant advantage in terms of reduced compress and decompress overhead compared to conventional compression methods. In our experiments, conventional compression exhibits average compress and decompress latencies of 48.8 and 45.2 microseconds, respectively. In contrast, delta compression within MedFS demonstrates notably lower average compress/decompress latencies of 7.4 and 8.3 microseconds. These latencies are marginal in the I/O latency of reading operations, which stands at 507.5 microseconds.

Energy consumption. We evaluated the device's power consumption using the Moonson High Voltage Power Monitor (HVPM) [19]. Experiment results reveal that MedFS reduces the average power consumption of the evaluated applications detailed in Tab. 2 by 9.2% on average compared to F2FS-NODC and by 2.6% compared to FPC.

Overhead of DCI. In DCI, when evicted delta chunks are decompressed and flushed to ensure data consistency, the system must temporarily suspend writing pages whose data offset is about to be updated in the inline area. Consequently, this delta eviction process results in an excessively prolonged tail latency for page writing, violating the QoS requirements [47]. Since delta eviction occurs prior to page writing, we evaluate the worst-case I/O latency, containing the processing time required for DCI to decompress and flush all evicted delta chunks. As depicted in Fig. 12 (b), the average processing time ranges from 42 to 475 milliseconds. Such extended processing time could lead to frame-blocking or dropping, which is unacceptable for user-experience-centric mobile devices. This underscores the necessity for DCM in MedFS.

The second overhead of DCI arises from traversing delta chunks in file inodes to locate the target. This traversal overhead averages at 0.2 microseconds, which is negligible.

Overhead of DCM. DCM necessitates the creation of meta-pages and *Compact* pages. In our approach, we propose compressing a minimum of 5 pages into 1 page through DCM, ensuring that flushing meta-pages and Compact pages does not result in read/write amplification. In terms of storage overhead, our experiments indicate that it requires only 7.1MB to store meta-pages and *Compact* pages when generating 4.2GB files. Furthermore, HCluster averages 1.2 microseconds for clustering during data compression, a marginal addition to the average I/O latency. Centroid updating in HCluster occurs exclusively in the background process BGRes, consuming an average of 0.9 milliseconds. This updating process remains hidden in the background, mitigating any impact on foreground operations. The third overhead is attributed to BGRes, which does not compromise system performance by amortizing its overhead in the background.

7 Related Work

Data Reduction in File System. Betrfs [20] optimizes the wandering tree problem of B-tree FS (e.g., BtrFS, XFS) by caching the file data into the inode. KevinFS [34] proposes a key-value interface between the file system and the SSD to reduce the I/O traffic between the host and SSD. UFIA [75] identifies the frequently updated inodes and reorganizes them in adjacent physical locations on the block device. However, they do not focus on data compression.

Conventional Data Compression. Ji *et al.* [23] propose FPC, which performs foreground compression on write-intensive barely-read I/O data, and background compression to re-organize read critical blocks of executable files. Under FPC, the compression ratio could be limited to 0.2 due to the constraint of the max number of compressed logical pages in one physical page. EROFS *et al.* [13] is a compression-friendly read-only file system that leverages fixed-sized output compression and memory-efficient decompression to achieve high performance on mobile devices. Zhang *et al.* [82] propose to accelerate the executable files compression with hardware in FS. Zhang *et al.* [81] design a F2FS-coupled deduplication scheme for removing duplicated data chunks. These works cannot overcome the constraint of conventional compression methods.

Delta Compression. To mitigate the detrimental effect of conventional compression, some works perform delta compression in the storage [1, 27, 59, 73, 83], which are orthogonal to our MedFS. On the host side, Lee *et al.* [37] adopt differential logging to reduce the write stress of in-memory DB systems. Other works [29, 57, 66] propose external non-volatile buffers as the log buffer, which could increase the cost of mobile devices, and thus are less attractive.

While there exist plenty of compression works, this study delves into mobile device delta compression, using MedFS to integrate compression modules into FS metadata. MedFS stores most delta chunks inline to avoid external buffer costs, fetches chunks from memory-resident inodes to limit read amplification, and leverages LFS benefits alongside delta compression to reduce write amplification.

8 Conclusion

This paper proposes a metadata-enabled delta compression-based approach (MedFS) for aggressive write traffic reduction in LFS-based mobile devices. MedFS leverages the benefits of the large inline area in delta maintenance and manages the inline area carefully for utmost compression efficacy. In addition, MedFS introduces a novel main data area delta-maintenance scheme, to break through the constraint of the limited inline area. Finally, MedFS leverages LFS to mitigate the write amplification issue of delta compression. Evaluation results show that MedFS reaches a significant write traffic reduction by effective data compression.

References

[1] Alexandra Angerd, Angelos Arelakis, Vasilis Spiliopoulos, Erik Sintorn, and Per Stenström. Gbdi: Going beyond base-delta-immediate compression with global bases. In *Proceedings of the 28th IEEE International Symposium on High-Performance Computer Architecture (HPCA'22)*, pages 1115–1127, 2022.

[2] Yu Cai, Saugata Ghose, Erich F. Haratsch, Yixin Luo, and Onur Mutlu. Error characterization, mitigation, and recovery in flash-memory-based solid-state drives. *Proceedings of the IEEE*, 105(9):1666–1704, 2017.

[3] Chandranil Chakraborttii and Heiner Litz. Reducing write amplification in flash by death-time prediction of logical block addresses. In *Proceedings of the 14th ACM International Conference on Systems and Storage (SYSTOR'21)*, 2021.

[4] Tseng-Yi Chen, Yuan-Hao Chang, Shuo-Han Chen, Chih-Ching Kuo, Ming-Chang Yang, Hsin-Wen Wei, and Wei-Kuan Shih. wrjfs: A write-reduction journaling file system for byte-addressable nvram. *IEEE Transactions on Computers*, 67(7):1023–1038, 2018.

[5] Zhangyu Chen, Yu Hua, Pengfei Zuo, Yuanyuan Sun, and Yuncheng Guo. Reducing bit writes in non-volatile main memory by similarity-aware compression. In *Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC'20)*, pages 1–6, 2020.

[6] Zhuan Chen and Kai Shen. Ordermergededup: Efficient, failure-consistent deduplication on flash. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*, pages 291–299, 2016.

[7] Dongsoo Choi and Dongkun Shin. Semantic-aware hot data selection policy for flash file system in android-based smartphones. In *Proceedings of the 19th International Conference on Parallel and Distributed Systems (ICPADS'13)*, pages 444–445, 2013.

[8] Wonil Choi, Bhuvan Urgaonkar, Mahmut Kandemir, Myoungsoo Jung, and David Evans. Fair write attribution and allocation for consolidated flash cache. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, page 1063–1076, 2020.

[9] Jace Courville and Feng Chen. Understanding storage i/o behaviors of mobile applications. In *Proceedings of the 32nd Symposium on Mass Storage Systems and Technologies (MSST'16)*, pages 1–11, 2016.

[10] Jinhua Cui, Kai Tang, and Laurence T Yang. A fast secure deletion strategy for high-density flash memory through wom-v codes. In *Proceedings of the 60th ACM/IEEE Design Automation Conference (DAC'23)*, pages 1–6, 2023.

[11] Arjun Deb, Paolo Faraboschi, Ali Shafiee, Naveen Muralimanohar, Rajeev Balasubramonian, and Robert Schreiber. Enabling technologies for memory compression: Metadata, mapping, and prediction. In *Proceedings of the 34th IEEE International Conference on Computer Design (ICCD'16)*, pages 17–24, 2016.

[12] EVERYPHONE. What is the average mobile lifeexpectancy. https://everphone.com/en/blog/smartphone-lifespan/.

[13] Xiang Gao, Mingkai Dong, Xie Miao, Wei Du, Chao Yu, and Haibo Chen. Erofs: A compression-friendly readonly file system for resource-scarce devices. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'19)*, pages 149–162, 2019.

[14] Amin Ghasemazar, Prashant Nair, and Mieszko Lis. Thesaurus: Efficient cache compression via dynamic clustering. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, pages 527–540, 2020.

[15] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The unwritten contract of solid state drives. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys'17)*, page 127–144, 2017.

[16] Yuan He, Lingfeng Xiang, Wen Xia, Hong Jiang, Zhenhua Li, Xuan Wang, and Xiangyu Zou. Dsync: a lightweight delta synchronization approach for cloud storage services. In *Proceedings of the 36th Symposium on Mass Storage Systems and Technologies (MSST'20)*, pages 1–14, 2020.

[17] Qingda Hu, Jinglei Ren, Anirudh Badam, and Thomas Moscibroda. Log-structured non-volatile main memory. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*, pages 703–717, 2017.

[18] Joo-Young Hwang, Seokhwan Kim, Daejun Park, Yong-Gil Song, Junyoung Han, Seunghyun Choi, Sangyeun Cho, and Youjip Won. Zms: Zone abstraction for mobile flash storage. In *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC'24)*, pages 173–189, 2024.

[19] Moonson Solutions Inc. Moonson high voltage power monitor, 2021.

[20] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, et al. Betrfs:

A right-optimized write-optimized file system. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, pages 301–315, 2015.

[21] Geonhwa Jeong, Bikash Sharma, Nick Terrell, Abhishek Dhanotia, Zhiwei Zhao, Niket Agarwal, Arun Kejariwal, and Tushar Krishna. Understanding data compression in warehouse-scale datacenter services. In *Proceedings of the 2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'22)*, pages 221–223, 2022.

[22] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/o stack optimization for smartphones. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC'13)*, pages 309–320, 2013.

[23] 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 the 19th USENIX Conference on File and Storage Technologies (FAST'21)*, pages 127–140, 2021.

[24] Cheng Ji, Riwei Pan, Li-Pin Chang, Liang Shi, Zongwei Zhu, Yu Liang, Tei-Wei Kuo, and Chun Jason Xue. Inspection and characterization of app file usage in mobile devices. *ACM Transactions on Storage*, 16(4):1–25, 2020.

[25] Yizheng Jiao, Simon Bertron, Sagar Patel, Luke Zeller, Rory Bennett, Nirjhar Mukherjee, Michael A. Bender, Michael Condict, Alex Conway, Martín Farach-Colton, Xiongzi Ge, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. Betrfs: A compleat file system for commodity ssds. In *Proceedings of the 17th European Conference on Computer Systems (EuroSys'22)*, page 610–627, 2022.

[26] Ziyang Jiao, Xiangqun Zhang, Hojin Shin, Jongmoo Choi, and Bryan S Kim. The design and implementation of a capacity-variant storage system. In *Proceedings of the 22nd USENIX Conference on File and Storage Technologies (FAST'24)*, pages 159–176, 2024.

[27] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. X-ftl: transactional ftl for sqlite databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*, pages 97–108, 2013.

[28] Saeed Kargar, Heiner Litz, and Faisal Nawab. Predict and write: Using k-means clustering to extend the lifetime of NVM storage. In *Proceedings of the 37th IEEE International Conference on Data Engineering (ICDE'21)*, pages 768–779, 2021.

[29] Junghoon Kim, Changwoo Min, and Young Ik Eom. Reducing excessive journaling overhead with small-sized nvram for mobile devices. *IEEE Transactions on Consumer Electronics*, 60(2):217–224, 2014.

[30] Juno Kim, Yun Joon Soh, Joseph Izraelevitz, Jishen Zhao, and Steven Swanson. Subzero: Zero-copy io for persistent main memory file systems. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys'20)*, page 1–8, 2020.

[31] Taejin Kim, Sangwook Shane Hahn, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. PCStream: Automatic stream allocation using program contexts. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'18)*, 2018.

[32] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. Fully automatic stream management for multi-streamed ssds using program contexts. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*, pages 295–308, 2019.

[33] Yannis Klonatos, Thanos Makatos, Manolis Marazakis, Michail D. Flouris, and Angelos Bilas. Transparent online storage compression at the block-level. *ACM Transactions on Storage*, 8(2):1–33, 2012.

[34] Jinhyung Koo, Junsu Im, Jooyoung Song, Juhyung Park, Eunji Lee, Bryan S Kim, and Sungjin Lee. Modernizing file system through in-storage indexing. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*, pages 75–92, 2021.

[35] Kevin Kremer and André Brinkmann. Fadac: A self-adapting data classifier for flash memory. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR'19)*, pages 167–178, 2019.

[36] 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'15)*, pages 273–286, 2015.

[37] Juchang Lee, Kihong Kim, and Sang Kyun Cha. Differential logging: A commutative and associative logging scheme for highly parallel main memory database. In *Proceedings of the 17th International Conference on Data Engineering (ICDE'01)*, pages 173–182, 2001.

[38] Kisung Lee and Youjip Won. Smart layers and dumb result: Io characterization of an android-based smartphone. In *Proceedings of the 10th ACM international*

conference on Embedded software (EMSOFT'12), pages 23–32, 2012.

[39] Yongmyung Lee, Jong-Hyeok Park, Jonggyu Park, Hyunho Gwak, Dongkun Shin, Young Ik Eom, and Sang-Won Lee. When f2fs meets address remapping. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'22)*, page 31–36, 2022.

[40] Bingzhe Li, Chunhua Deng, Jinfeng Yang, David Lilja, Bo Yuan, and David Du. Haml-ssd: A hardware accelerated hotness-aware machine learning based ssd management. In *Proceedings of the 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'19)*, pages 1–8, 2019.

[41] Changlong Li, Yu Liang, Liang Shi, Chao Wang, Chun Jason Xue, and Xuehai Zhou. Flexible and efficient memory swapping across mobile devices with legoswap. *IEEE Transactions on Parallel and Distributed Systems*, 35(1):140–153, 2024.

[42] Changlong Li, Liang Shi, and Chun Jason Xue. Mobileswap: Cross-device memory swapping for mobile devices. In *Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC'21)*, pages 115–120, 2021.

[43] Chunguang Li, Dan Feng, Yu Hua, Wen Xia, Leihua Qin, Yue Huang, and Yukun Zhou. Bac: Bandwidth-aware compression for efficient live migration of virtual machines. In *Proceedings of the 2017 IEEE Conference on Computer Communications (INFOCOM'17)*, pages 1–9, 2017.

[44] Yu Liang, Chenchen Fu, Yajuan Du, Aosong Deng, Mengying Zhao, Liang Shi, and Chun Jason Xue. An empirical study of f2fs on mobile devices. In *Proceedings of the 23th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'17)*, pages 1–9, 2017.

[45] Yu Liang, Riwei Pan, Tianyu Ren, Yufei Cui, Rachata Ausavarungnirun, Xianzhang Chen, Changlong Li, Tei-Wei Kuo, and Chun Jason Xue. CacheSifter: Sifting cache files for boosted mobile performance and lifetime. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST'22)*, pages 445–459, 2022.

[46] Lineage. Lineageos android distribution. http://https://lineageos.org/.

[47] Lei Liu, Xinglei Dou, and Yuetao Chen. Intelligent resource scheduling for co-located latency-critical services: A Multi-Model collaborative learning approach. In *Proceedings of the 21th USENIX Conference on File and Storage Technologies (FAST'23)*, pages 153–166, 2023.

[48] Tao Lu, Wen Xia, Xiangyu Zou, and Qianbin Xia. Adaptively compressing iot data on the resource-constrained edge. In *Proceedings of the 3th USENIX Workshop on Hot Topics in Edge Computing (HotEdge'20)*, 2020.

[49] Yina Lv, Liang Shi, Qiao Li, Congming Gao, Yunpeng Song, Longfei Luo, and Youtao Zhang. Mgc: Multiple-gray-code for 3d nand flash based high-density ssds. In *Proceedings of the 29th IEEE International Symposium on High-Performance Computer Architecture (HPCA'23)*, pages 122–136, 2023.

[50] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33, 2007.

[51] Alysha Puti Maulidina, Rachel Anastasia Wijaya, Kimberly Mazel, and Maria Seraphina Astriani. Comparative study of data compression algorithms: Zstandard, zlib & lz4. In *Proceedings of the 2023 International Conference on Science, Engineering Management and Information Technology (SEMIT'23)*, pages 394–406, 2023.

[52] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. Sfs: random write considered harmful in solid state drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, volume 12, pages 1–16, 2012.

[53] Jayashree Mohan, Rohan Kadekodi, and Vijay Chidambaram. Analyzing io amplification in linux file systems. *arXiv preprint arXiv:1707.08514*, 2017.

[54] David T Nguyen, Gang Zhou, Guoliang Xing, Xin Qi, Zijiang Hao, Ge Peng, and Qing Yang. Reducing smartphone application delay through read/write isolation. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'15)*, pages 287–300, 2015.

[55] Michael F. Nowlan, Bryan Ford, and Ramakrishna Gummadi. Non-linear compression: Gzip me not! In *Proceedings of the 4th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'12)*, page 11, 2012.

[56] Oberhumer. Lzo real-time data compression library. http://www.oberhumer.com/opensource/lzo/.

[57] Gihwan Oh, Sangchul Kim, Sang-Won Lee, and Bongki Moon. Sqlite optimization with phase change memory for mobile applications. *Proceedings of the VLDB Endowment*, 8(12):1454–1465, 2015.

[58] O. Ozturk, G. Chen, M. Kandemir, and I. Kolcu. Compiler-guided data compression for reducing memory consumption of embedded applications. In *Proceedings of the 11th Asia and South Pacific Conference on Design Automation (ASP-DAC'06)*, pages 1–6, 2006.

[59] Jisung Park, Jeonggyun Kim, Yeseong Kim, Sungjin Lee, and Onur Mutlu. DeepSketch: A new machine Learning-Based reference search technique for Post-Deduplication delta compression. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST'22)*, pages 247–264, 2022.

[60] Gennady Pekhimnko, Vivek Seshadri, Yoonqu Kim, Hongyi Xin, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Linearly compressed pages: A low-complexity, low-latency main memory compression framework. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*, pages 172–184, 2013.

[61] PURESTORAGE. What is qlc ssd. https://www.purestorage.com/knowledge/what-is-qlc-flash.html.

[62] Eunhee Rho, Kanchan Joshi, Seung-Uk Shin, Nitesh Jagadeesh Shetty, Jooyoung Hwang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. Fstream: Managing flash streams in the file system. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*, pages 257–264, 2018.

[63] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage*, 9(3):1–32, 2013.

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

[65] Samsung. F2fs open source code. https://github.com/jaegeuk/f2fs-stable.

[66] Mungyu Son, Junwhan Ahn, and Sungjoo Yoo. Non-volatile write buffer-based journaling bypass for storage write reduction in mobile devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(9):1747–1759, 2017.

[67] STATISTICA. Average lifespan. https://www.statista.com/statistics/619788/average-smartphone-life/.

[68] visualcapitalist. Visualizing the top countries, by mobile data usage. https://www.visualcapitalist.com/top-countries-by-mobile-data-usage/.

[69] Daniel Lin-Kit Wong, Hao Wu, Carson Molder, Sathya Gunasekar, Jimmy Lu, Snehal Khandkar, Abhinav Sharma, Daniel S Berger, Nathan Beckmann, and Gregory R Ganger. Baleen:ml admission & prefetching for flash caches. In *Proceedings of the 22th USENIX Conference on File and Storage Technologies (FAST'24)*, pages 347–371, 2024.

[70] David Woodhouse. Jffs2. http://www.linux-mtd.infradead.org/doc/jffs2.html.

[71] Chao Wu, Yufei Cui, Cheng Ji, Tei-Wei Kuo, and Chun Jason Xue. Pruning deep reinforcement learning for dual user experience and storage lifetime improvement on mobile devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3993–4005, 2020.

[72] Guanying Wu and Xubin He. Delta-ftl: improving ssd lifetime via exploiting content locality. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*, EuroSys '12, page 253–266, 2012.

[73] Guanying Wu and Xubin He. Delta-ftl: improving ssd lifetime via exploiting content locality. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 253–266, 2012.

[74] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-Tail flash: Near-Perfect elimination of garbage collection tail latencies in NAND SSDs. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*, pages 15–28, 2017.

[75] Chaoshu Yang, Duo Liu, Xianzhang Chen, Runyu Zhang, Wenbin Wang, Moming Duan, and Yujuan Tan. Reducing write amplification for inodes of journaling file system using persistent memory. In *Proceedings of the 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE'19)*, pages 866–871, 2019.

[76] Jing Yang, Shuyi Pei, and Qing Yang. Warcip: Write amplification reduction by clustering i/o pages. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR'19)*, page 155–166, 2019.

[77] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. Fifo queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*, pages 130–149, 2023.

[78] Lihua Yang, Zhipeng Tan, Fang Wang, Dan Feng, Hongwei Qin, Shiyun Tu, Jiaxing Qian, and Yuting Zhao. Improving f2fs performance in mobile devices with adaptive reserved space based on traceback. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(1):169–182, 2022.

[79] 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'19)*, pages 633–646, 2019.

[80] Tao Zhang, Aviad Zuck, Donald E. Porter, and Dan Tsafrir. Apps can quickly destroy your mobile's flash: Why they don't, and how to keep it that way. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'19)*, page 207–221, 2019.

[81] Tiangmeng Zhang, Renhui Chen, Zijing Li, Congming Gao, Chengke Wang, and Jiwu Shu. Design and implementation of deduplication on f2fs. *ACM Transactions on Storage*, 20(4):1–50, 2024.

[82] Xuebin Zhang, Jiangpeng Li, Hao Wang, Danni Xiong, Jerry Qu, Hyunsuk Shin, Jung Pill Kim, and Tong Zhang. Realizing transparent os/apps compression in mobile devices at zero latency overhead. *IEEE Transactions on Computers*, 66(7):1188–1199, 2017.

[83] Xuebin Zhang, Jiangpeng Li, Hao Wang, Kai Zhao, and Tong Zhang. Reducing solid-state storage device write stress through opportunistic in-place delta compression. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*, pages 111–124, 2016.

[84] Yucheng Zhang, Wen Xia, Dan Feng, Hong Jiang, Yu Hua, and Qiang Wang. Finesse: Fine-grained feature locality based fast resemblance detection for post-deduplication delta compression. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*, pages 121–128, 2019.

[85] You Zhou, Fei Wu, Ping Huang, Xubin He, Changsheng Xie, and Jian Zhou. An efficient page-level ftl to optimize address translation in flash memory. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)*, page 1–16, 2015.

[86] Aviad Zuck, Sivan Toledo, Dmitry Sotnikov, and Danny Harnik. Compression and SSDs: Where and how? In *Proceedings of the 2th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW'14)*, 2014.