Pattern-Guided File Compression with User-Experience Enhancement for Log-Structured File System on Mobile Devices

Cheng Ji, *Nanjing University of Science and Technology;* Li-Pin Chang, *National Chiao Tung University, National Yang Ming Chiao Tung University;* Riwei Pan and Chao Wu, *City University of Hong Kong;* Congming Gao, *Tsinghua University;* Liang Shi, *East China Normal University;* Tei-Wei Kuo and Chun Jason Xue, *City University of Hong Kong*

https://www.usenix.org/conference/fast21/presentation/ji

This paper is included in the Proceedings of the 19th USENIX Conference on File and Storage Technologies.

February 23–25, 2021

978-1-939133-20-5

Pattern-Guided File Compression with User-Experience Enhancement for Log-Structured File System on Mobile Devices

Cheng Ji[1], Li-Pin Chang[2,3], Riwei Pan[4], Chao Wu[4], Congming Gao[5], Liang Shi[6], Tei-Wei Kuo,[4] and Chun Jason Xue[4]

[1]*Nanjing University of Science and Technology* [2]*National Chiao Tung University* [3]*National Yang Ming Chiao Tung University*
[4]*City University of Hong Kong* [5]*Tsinghua University* [6]*East China Normal University*

Abstract

Mobile applications exhibit unique file access patterns, often involving random accesses of write-mostly files and read-only files. The high write stress of mobile applications significantly impacts on the lifespan of flash-based mobile storage. To reduce write stress and save space without sacrificing user-perceived latency, this study introduces FPC, file access pattern guided compression. FPC is optimized for the random-writes and fragmented-reads of mobile applications. It features dual-mode compression: Foreground compression handles write-mostly files for write stress reduction, while background compression packs random-reading file blocks for boosted read performance. FPC exploits the out-of-place updating design in F2FS, a log-structured file system for mobile devices, for the best effect of the proposed dual-mode compression. Experimental results showed that FPC reduced the volume of total write traffic and executable file size by 26.1% and 23.7% on average, respectively, and improved the application launching time by up to 14.8%.

1 Introduction

Mobile devices including smartphones, tablets, and wearable devices are now a necessity in everyone's daily life. Recent researches reported that the number of smartphone shipments surpassed 1.37 billion in 2019 [1] and 86% of them were based on the Android [2]. Mobile devices employ flash memory for persistent data storage. While the performance of mobile processors is improving drastically, the improvement of mobile storage performance is, however, relatively slow. Recent studies report that I/O operations on mobile storage are write-dominant [3–7], and the write pattern is highly random and synchronous. These write operations are identified closely related to user-perceived latencies due to the relatively high write latency of flash memory [8, 9]. In addition, as flash memory technology is evolving toward high cell-bit-density at the cost of degraded endurance, the high write stress negatively impacts on the flash-storage lifespan.

Android-based mobile devices exhibit very distinct file usage patterns compared with desktop systems: First, mobile applications heavily rely on an embedded database layer, SQLite, for transactional data management; Second, Android packs various runtime resources such as executable binaries and compiled resources into large executable files. We examined the contents of these files and found that they are highly compressible. While the database files contribute to a large portion of the write traffic, executable files are large in size. Intuitively, existing file compression techniques, such as those reported in [10–13], can be adopted to reduce write stress and to save space. However, the existing designs might not be effective or risk degraded user experience in mobile devices.

Manipulating SQLite databases generates many small file overwrite and append operations [6, 14]. These small operations are highly fragmented in the storage space and the cause has been identified related to the file fragmentation problem [8]. With the fragmented updates, file compression on top of a conventional in-place-updating file system, like Ext4 [15], may create many holes in the storage space because a compressed file block may not fit in its original space after an update. With the small append operations, file compression cannot use a large compression window on new data for a better compression result. Regarding Android executable files, although they are sequentially written upon installation, they are subject to small, random read operations during application launching. Decompressing file blocks from random file offsets significantly amplifies the I/O read overhead because of the larger unit size of block I/O [12, 13]. These unique file access patterns of database files and executable files in Android mobile devices are, however, not well studied in prior file compression work.

We believe that file compression should be judiciously applied to files based on their access patterns. This study presents ***F**ile **P**attern-guided **C**ompression (**FPC**)* for mobile devices. FPC features foreground compression and background compression. Considering the timing overhead of compression, foreground compression is applied only on the write-intensive, highly compressible SQLite files. In particular, SQLite journal files are barely read (write-ahead logging journal) or never read (roll-back journal). For the journal files, FPC further applies deep compression by packing file-system metadata with user data and compressing them using a larger compression window. Executable files, which are also highly compressible, are subject to small, random reads upon application launching. Hence they are not suitable for sequential compression in the foreground. Instead, this paper proposes to apply infrequent background compression to re-organize read-critical blocks of executable files through compression. As a result, both user-perceived application launching latency and storage space utilization can be improved.

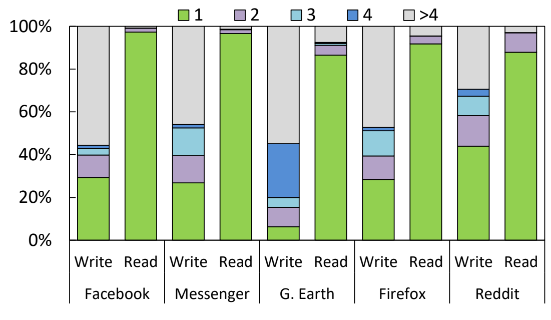
The effect of the proposed FPC is best achieved by an implementation on top of a log-structured file system, e.g., F2FS [16] for mobile storage. FPC exploits out-of-place updating and reverse mapping, which are existing mechanisms in F2FS, for foreground compression and critical block reorganization, respectively. With out-of-place updating, small, fragmented writes and appends to SQLite files and their ancillary file-system metadata can be combined as sequential, deep compression with a large compression window. With the reverse mapping of storage addresses to file block offsets, it is possible to load multiple compressed read-critical blocks of executable files through a single block I/O request to accelerate application launching. In summary, this work makes the following contributions:

- Proposing a foreground compression method for write mostly, highly compressible files to improve write stress and energy consumption of mobile storage;
- Proposing a background compression method that identifies and re-organizes read-critical blocks in executable files for fast application launching and space saving;
- Exploiting out-of-place updating and reverse mapping of F2FS for the best effect of deep, metadata-level file compression and fast application launching.

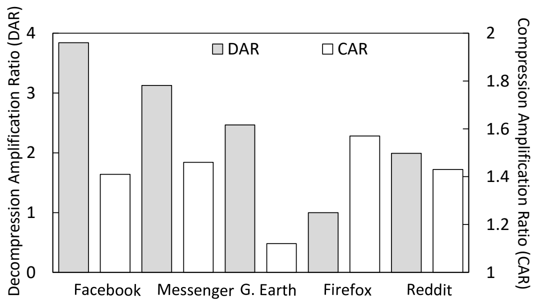
2 Background and Motivation

2.1 I/O System and Storage of Mobile Devices

Android is the dominant operating system for mobile devices today. The Android I/O system consists of host system software and a flash storage device. The host software includes a lightweight database layer, SQLite, for transactional data management. SQLite operates on top of the file system layer, where two major options are provided: Ext4 [15], an in-place updating file system, and F2FS [16], a log-structured file system. Because random file writes may unexpectedly increase the cost of garbage collection of flash storage, F2FS benefits flash storage by converting random updates into sequential, out-of-place updates. As out-of-place updates create outdated data in the storage space, F2FS needs to timely compact valid data to provide contiguous free space for future sequential writes.



(a)



(b)

Figure 1: (a) Characterization of file write and read sizes for mobile applications (unit: page). (b) Low compression efficacy of traditional sequential compression approaches. *Compression Amplification Ratio* (CAR) and *Decompression Amplification Ratio* (DAR) are presented, respectively.

High Write Stress. Applications heavily rely on the SQLite journaling mechanism for data integrity guarantee, producing enormous synchronous, random block writes [6]. The write traffic is further amplified by other components of the I/O system. Specifically, free-space defragmentation for F2FS involves many extra data migrations [16], and flash garbage collection inside mobile storage requires data movements before memory erasing [17]. The multiple levels of write amplification become even worse when the level of file system fullness is high. The amplified write traffic noticeably degrades user-perceived latency [9]. In addition, as modern flash technology is evolving toward high bit-cell density at the cost of reduced endurance, e.g., a TLC flash block can only withstand about 1,000 P/E (program-erase) cycles [18], the excessive write traffic also poses concerns to the storage lifespan.

2.2 Pitfalls of File Compression

File compression is expected to save storage space and reduce the amount of I/O. However, it may not be the case for mobile storage because of the highly random nature of file reads and writes of mobile applications. In this section, we demonstrate the high randomness of read and write under selected popular applications and show that existing file compression designs could be harmful to space utilization and read performance.

In Android devices, read traffic and write traffic are mainly contributed by executable files and SQLite files, respectively [3, 14]. Figure 1(a) reports the size distribution of file read operations on *.apk executable files and that of file write operations on SQLite files. Results indicate that roughly more than one half of the write operations to SQLite files were not larger than 4 pages (16KB). Most of these small writes were bound for random file offsets. For read operations, nearly 90% of all read operations were not larger than one page in all applications. The file offsets of these reads were also highly fragmented.

Many existing compression file systems, e.g., Btrfs [10], JFFS2 [11] and EROFS [12], allow only compressed file blocks of consecutive offsets to be stored in the same storage block. A new storage block is allocated for a compressed file block if it does not continue the file offset of the last compressed file block. This design, referred as the *sequential compression method*, can seriously degrade the space utilization in mobile storage. To assess the severity of the problems, in Fig 1(b) we report *Compression Amplification Ratio* (CAR), which is the ratio of the total number of physical blocks [1] required to store a series of compressed logical blocks with the sequential compression method to the minimal number of physical blocks required to store all the compressed logical blocks. CAR reflects how the sequential compression method degrades space efficiency on random write through internal fragmentation. The CAR values indicated that the sequential compression method incurred more than 40% extra space requirement for 4 out of the 5 applications.

We then show how application launching suffered from an amplified read overhead with the sequential compression method. We employed *Decompression Amplification Ratio* (DAR) to show the ratio of the total number of compressed logical blocks stored in the physical blocks that are read to launch an application to the total number of compressed logical blocks that are actually required to launch the application. For example, DAR is 4 if a physical block stores four compressed logical blocks and only one of them is actually used. Fig 1(b) shows the DAR values were even higher than the CAR values because one-page reads dominated the overall read traffic. The CAR and DAR results showed that the existing sequential compression method unexpectedly degrades space utilization and amplifies the read overhead for application launching on mobile devices. This underlines the need for a new space management strategy to cope with the unique random I/O pattern of mobile applications.

2.3 Benefits of File Compression with LFS

File system compression should achieve a high compression efficiency (space saving) with a reduced decompression penalty. It is possible to achieve both by taking advantage of the file system structure and application behaviors of file

[1] In the rest of this paper, we refer to blocks in the storage space as physical blocks and blocks in the file address space as logical blocks.

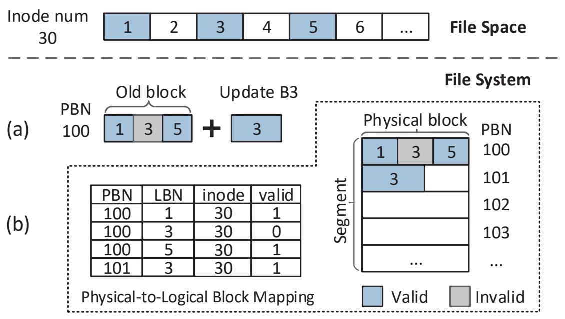


Figure 2: Updating compressed data with (a) in-place-updating file system and (b) a log-structured file system.

access. This study is based on the log-structured file system for mobile storage. Compared with conventional in-place-updating file systems, e.g., Ext4, LFS is highly friendly to file compression, as discussed below:

Out-of-Place Updating. Conventional in-place-updating file systems have a disadvantage of handling compression. In Figure 2(a), three data blocks are compressed and packed together into the same physical block (PBN 100). Consider that B3 is updated. If the new compressed size of B3 is larger than the old version, it is impossible to overwrite B3 in place. One solution is to re-write B1, B3, and B5 and repack them tightly in another free space. Another option is to align compressed data to predefined boundaries to allow future size changes of compressed data. However, with both options above, file compression suffers, from either amplified write traffic or internal fragmentation [19, 20]. By contrast, as shown in Figure 2(b), LFS appends the compressed B3 to a new block, avoiding rewriting of existing data and wasting of free space.

Reverse (Physical-to-Logical) Mapping. To support data migration of space cleaning, LFS, e.g., F2FS, maintains physical-to-logical block (P2L) mapping. The P2L mapping is necessary to determine whether a piece of data is valid (and requires migration) and to update new locations of data during cleaning. As Figure 2(b) shows, the P2L mapping provides the `inode` number and file offset of a compressed logical block (§ 4.3). Interestingly, file compression can leverage the existing P2L mapping mechanism for efficient decompression. We discerned that the launching of mobile applications, whose latency affects user experience the most [8], generated small, random reads on executable files (Section 3.3.1). With P2L mapping, it is possible to compress the file blocks necessary to application launching and pack them into physical blocks. This way, fewer block read requests are required to launch an application, improving the user-perceived latency.

Although the log-structured file system is friendly to compression, prior compression studies paid little attention to the read/write patterns of mobile systems. This study proposes to exploit the file access behaviors of mobile applications and the structure of F2FS to address two major design challenges: 1) Efficient file compression for write stress reduction and space saving and 2) Efficient re-organization and decompression of

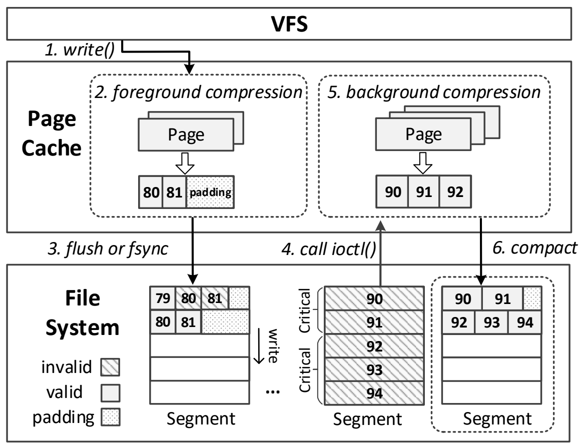


Figure 3: FPC architecture. Steps 1 to 3 show foreground compression on write-intensive files and Steps 4 to 6 show background compression/re-arrangement of read-critical data.

executable files for improved user experience.

3 Pattern-Guided File Compression

This section presents the proposed design principle and the details of foreground and background compression.

3.1 File Access Behaviors of Mobile Apps

We propose categorizing files according to their types of access (read or write) and hotness (access frequency): 1) Write-hot, read-cold files: Roll-back journals (*.db-journal) are a good fit in this category because they are frequently written [21]. However, they are rarely read (except during crash recovery). SQLite database files are also a good fit because many mobile applications frequently write SQLite database files but barely read them [22]. 2) Write-cold, read-hot files: Executable files fall in this category because they are immutable after installation or update. Android executable files are large and highly compressible [13]. However, the read latency of executable files is critical to user experience, so it is crucial to optimize the decompression overhead.

Both SQLite files and executable files are subject to random access: SQLite database files are prone to random updates, while executable files are subject to random reads. Although random updates will be converted into bulk writes through out-of-place updating, random reads, however, should be optimized through rearrangement of file blocks. Figure 3 shows the architecture of the proposed *File Pattern-guided Compression approach (FPC)*. FPC performs foreground compression on SQLite files for write stress reduction. On the other hand, executable files are left to background activities of block re-arrangement and compression for improved user experience and space saving.

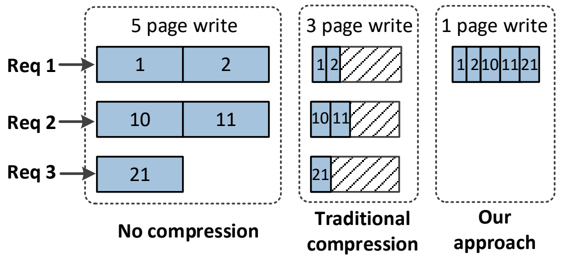


Figure 4: Comparison of traditional *sequential compression* approaches and the proposed compression approach. The Logical Block Number (LBN) is shown inside each data block.

3.2 Foreground Compression

This section presents the proposed foreground compression (FC) solution for write stress reduction. FC is focused on the compression of write-hot but read-cold files.

3.2.1 Non-Sequential File Block Compression

On-line file compression is a feature of existing file systems such as Btrfs [10] and JFFS2 [11]. In the prior designs, a physical block can only store compressed logical blocks of contiguous file offsets. This is because, first, file writes in desktop computers and servers are large in size (compared with mobile devices), so a sequential burst of compressed data sufficiently utilizes a physical block. Second, file compression is a separate layer in file system, and therefore compression of sequential file blocks minimally affects the existing index scheme of file blocks. This *sequential compression method*, which is adopted by Btrfs and JFFS2, results in poor space utilization in mobile storage, because the major write traffic contributor SQLite [4, 14] produces many small writes bound for random file offsets. In this case, a compressed file block usually demands a new physical block due to the irrelevant file offset from the prior compressed file block.

Fig. 4 shows the problem of the sequential compression method. Consider the three pending file write operations on the left-hand side. Three physical blocks are required for compression because the three file writes do not have sequential file offsets. Because compression reduces the file block sizes, these three physical blocks suffer from poor space utilization. On the contrary, we propose allowing file blocks of irrelevant file offsets to share the same physical block. As the right-hand side of Fig. 4 shows, only one physical block is used and the space utilization is high. However, compression of non-sequential file blocks is challenging because it increases the index resolution of file blocks to the sub-block level.

3.2.2 Selective Foreground File Compression

Foreground compression (FC) is performed in real time. Since unconditional compression risks poor write latency and extra energy consumption, foreground compression is highly selective to avoid such a drawback.

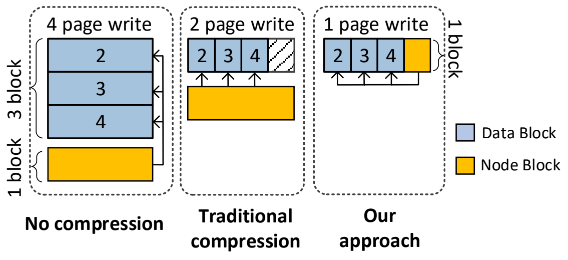


Figure 5: Comparison of traditional compression approaches and the proposed metadata-level file compression approach.

Selection of File Types. Since FC is part of the file system, it can simply ignore writes associated with the file types that are known to be incompressible, e.g., files with multimedia extensions *.jpg, *.mp4 and so on. Among all the other file types, SQLite files have been identified write-intensive and highly compressible [23]. In this study, *compression ratio* is defined as the ratio of the size after compression to that before compression. The smaller the better. We measured that the compression ratio of many SQLite files was better (lower) than 0.2, and compressing these SQLite files could benefit the write latency. On the other hand, although executable files are also highly compressible, sequential compression of executable files will significantly amplify the read overhead because such files are subject to small, random reads during application launching. FC leaves executable files to background compression.

Selection of Page Writes. Although FC can simply compress all writes associated with files having a *.db extension, it is possible that SQLite files are embedded with incompressible contents. For example, the Google Earth app stores map image tiles in *.db files using the BLOB (binary large object) format. A prior study reports that real-time identification of data compressibility is feasible [24]. Here, FC employs a sampling technique to quickly identify such incompressible contents: FC always compresses the first cached file page of an SQLite file write operation. If the first cached page is highly compressible, i.e., its data size can be reduced by at least one half, then FC compresses the rest cached pages of the write. Otherwise, FC forwards all the cached pages of the write to the original F2FS write logic.

3.2.3 Metadata-Level File Compression

F2FS stores user data in data blocks and file-system metadata (e.g., inodes) in node blocks. These two types of blocks are written to separate free spaces because node blocks are considered being updated more often. Small, synchronously-written files will experience a high metadata overhead.

The left-hand side of Fig. 5 shows the space allocation without file compression. Now, let the file undergo compression, the middle of Fig. 5 shows that the three data blocks are packed into a physical block while the uncompressed node block still occupies a second physical block. In this study, we propose writing data blocks of a *.db-journal file and their associated node block to the same segment through compression. The right-hand side of Fig. 5 shows that only one physical block is written with this method, while the traditional compression in the middle requires two. There are several rationales behind this design: First, the node block and data blocks of a *.db-journal file share the same lifetime because a rollback journal is discarded upon a successful SQLite transaction. Second, rollback journals are write-only (except during crash recovery) so packing metadata and user data together would have little impact on read performance. Although the design described above is based on rollback journaling in DELETE mode, it is also applicable to PERSIST mode. In PERSIST mode, rollback journals are reused. Like in DELETE mode, data blocks and the node block of reused journals are updated through F2FS out-of-place writing, and they are packed together for compression.

F2FS flushes data blocks before writing node blocks to avoid reference to uninitialized data. In our method, physical blocks containing all compressed data blocks are still flushed first. Let a mixed block be a physical block storing a few compressed data blocks and their associated, compressed node block. It is possible that upon crash recovery, the compressed node block in a mixed block is valid but its associated compressed data blocks in the same mixed block contain uninitialized data. To deal with this problem, we propose inserting a checksum to every mixed block. If a checksum fail is detected on a mixed block during crash recovery, the compressed node block in the mixed block is discarded.

3.3 Background Compression

Foreground compression on sequentially-written, random-read executable files risks degraded application launching performance. This part investigates the access patterns of executable files and proposes background compression (BC).

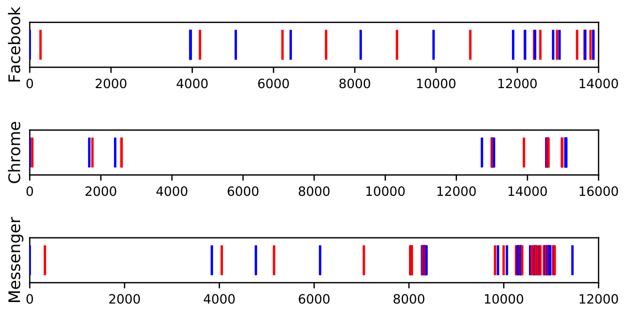


Figure 6: Distribution of read addresses within executable files during application launching. The X-axis shows the block offsets relative to the begin of files. Read requests having consecutive LBAs are marked in the same color (red and blue are used alternatively for clear presentation).

3.3.1 Highly Random Reads of Executable Files

Executable files, including .apk, .dex, .odex and .oat files, contribute to a large proportion of storage space [14], and they are highly compressible. The latencies of reading executable during application launching are crucial to user experience, because users have to wait until all necessary executable data are decompressed and loaded into memory.

We inspected how executable files were read with typical mobile applications. In order to accurately identify the required file data, we inserted routines to the Virtual File System (VFS) to extract related system calls, e.g., instrumenting `do_mmap` function to record reads on memory-mapped executable files and instrumenting `do_generic_file_read` function to extract reads on non-memory-mapped files. The duration of launching ended when the executable file of the application did not receive any read for one second.

Figure 6 shows the distributions of read addresses within the `base.apk` file, which is the main executable file for Facebook, Chrome, and Messenger. Results show that read requests of the inspected applications were small and random. While the executable files were large, only a small portion of executable file data was actually fetched for launching. For example, the executable file of Facebook was 80.6 MB, but only 632 KB of the file was read to launch Facebook. File pages were fetched through run-time demand paging, but the address distribution shows that these required pages were not well organized based on their correlation.

Apk files are actually a package of resource files. These files contribute a significant portion of read traffic during application launching, e.g., 49%, 27% and 21% of total read requests for Facebook, Chrome, Messenger, respectively. The random reads of apk files had great impacts on application launching latencies, which will be shown in Section 5. We de-compiled the Facebook's base.apk and analyzed which resource files were actually read for launching. The accessed resource files were identified by matching the read addresses and the file offsets of the resource files within the apk. The base.apk contained 17,439 resource files. Table 1 shows that only a small subset of the resource files (110 out of 17,439) were read. These required resource files (`*.xml`, `*.png`, etc.) were dispersed to random locations within the `apk` and were all accessed through single-block read requests. These small, random reads increased the block I/O count and degraded the user-perceivable application-launching latency.

Table 1: Resource files read from Facebook's *base.apk*.

Type	.xml	.arsc	.png	.dex	others	apk metadata
File count	51	1	41	9	8	1
Blocks read	1	32	1	1	1	17

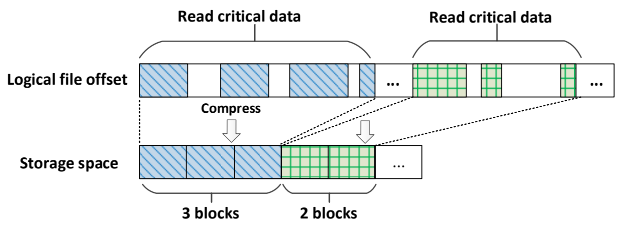


Figure 7: Compression of read-critical data in executable files.

3.3.2 Read-Guided File Compression

Decompressing small pieces of data from random file offsets negatively impacts on read performance. Compression provides an opportunity of reorganizing necessary file blocks to reshape the read patterns for better decompression efficiency.

Read-Critical Data Compression: We refer to a piece of data in an executable file as *read-critical data* if the data is required to launch an application. Because of the random read pattern of application launching, sequential compression of executable files risks the mixture of read-critical and non-read-critical data in a physical block. The mixture of data significantly amplifies application launching time because non-read-critical data are loaded and decompressed. The upper half of Figure 7 illustrates that critical data are scattered in an executable file. We propose monitoring the read-critical data set during application launching. Later on, upon requests, the file system compacts these critical data and compresses them into file blocks, as shown in the bottom half of Figure 7. Notice that the compaction changes the storage layout of file blocks but *not* the logical order of file blocks.

Compacting and compressing read-critical data avoids to load and decompress non-read-critical data and thus prevents decompression from degrading application launching time. It also complements the existing file pre-fetching mechanism, which could unexpectedly load non-read-critical data from sequential file offsets. When reading and decompressing a piece of read-critical data, the file system also brings the other read-critical data of the same physical block into the page cache. The prefetching of read-critical data requires the physical-to-logical mapping of F2FS (see Section 4.3), which is a unique feature of log-structure file systems.

Read-Critical Data Identification: The efficacy of the proposed read-critical data compression is subject to the I/O pattern of executable files. It has been reported that application launching exhibits highly predictable I/O patterns on desktop computers [25]. This phenomenon is also true for mobile applications: a cold start process of launching an application was tested through the `am start` command of the `adb shell` with the page cache cleared beforehand. We monitored the file read operations on the file blocks of the base.apk file of Facebook, Chrome, Messenger, Twitter, Google Earth, and Firefox for 5 rounds of cold starts. Results show that the set of file blocks read during the multiple cold starts barely changed (difference was between 1% to 3%). The read performance of these file blocks affects the user experience the most because the application screen is not fully rendered yet. As the applications continued to launch after the `am start` command returned, a higher degree of variation in the start-up file blocks was observed as they began to display random advertisements and splash screens.

Based on the results above, we propose capturing the core set of the read-critical data, i.e., those shared among different rounds of application cold starts, for fast application launch. When an executable file is opened (or memory-mapped), our method records the offsets of file reads. The proposed design collects the read offsets for the first three rounds of cold starts of a new application. After this, the core read-critical data will be compacted and compressed into a set of physical blocks. Non-core read-critical data will be compressed to another set of physical blocks. When the system is lightly loaded, a resident user-level process invokes an `ioctl()` with an argument of an `inode` number of an executable file, and the file system begins to compress the read-critical data of the file in the background. The compression of executable file blocks is conducted during system idle periods (see Section 4.4).

4 Implementation

This section discusses how to implement the proposed FPC in a log-structured file system, F2FS, for mobile devices.

4.1 Dynamic Compression Window

A large compression window sufficiently populates a large dictionary for effective data compression. However, partial reads in a large chunk of compressed data would induce a high overhead because the compressed data must be read and decompressed as a whole. The selection of the compression window size is based on the following rules: The default size of the compression window is set to 4 KB to avoid an amplified read overhead. Exceptions are as follows: First, for foreground compression on SQLite journals, the compression window is large as the block write request. Because these files are barely read, using a large compression window has little effect on read performance. Second, background compression uses 32 pages (128 KB) as the compression window size for *.dex and *.odex, as mobile applications often performed bulk, sequential read on such executable files through *mmap()*. Since a page fault in Linux is handled by fetching a set of 32 pages into the page cache, using a compression window as large as the pre-paging size effectively improves the compression ratio without sacrificing the read performance. Due to the bulk reading, a chunk of compressed data of *.dex and *.odex files is allowed to be stored across physical block boundaries.

4.2 Sub-Block L2P Mapping

To better accommodate small file reads and writes of mobile applications, we enhance the mapping process to improve the

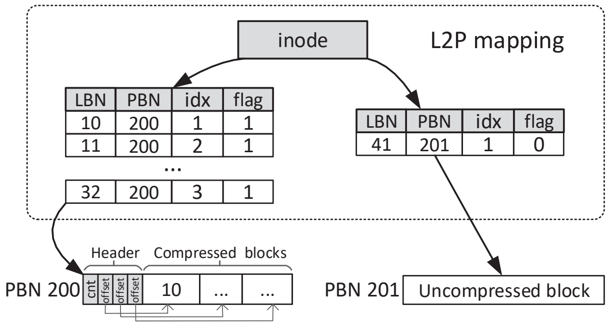


Figure 8: The extended L2P mapping.

read-write data compression efficacy. Because a compressed file system stores multiple compressed blocks in a physical block, it requires logical-to-physical (L2P) mapping at the sub-block level [2]. The existing L2P mapping for F2FS is managed at the block level using special node blocks `inode`. However, the current F2FS inode structure is 4 KB and it cannot accommodate all the extra metadata for sub-block indexing. To deal with this problem, our design appends extra bits to each mapping entry of an inode block: 3 bits for sub-block indexing and 1 bit as a compression flag. In the storage space, if a physical block contains compressed blocks, then the physical block is formatted into a header area and a compressed block area. A header entry is indexed by the sub-block number of an inode mapping entry, and the header entry contains a starting offset (16 bits) of the corresponding compressed block within the physical block.

Figure 8 shows an example of the extended L2P mapping and physical block layout. Suppose that a read of the tenth logical block (LBN 10) of a file will be served. The file system first locates the inode of the file and reads the LBN-to-PBN mapping entry for LBN 10. It identifies that the corresponding physical block PBN 200 is compressed (flag=1) and LBN 100 is the first compressed block (idx=1) in the physical block. After reading PBN 200, the file system loads the decompressed data of LBN 10 into the page cache and completes the read. It is possible that the header entries of multiple logical blocks refer to the same compressed block. To distinguish between logical blocks in the same compressed block, each header entry contains a logical block sequence number.

Since a physical block contains multiple compressed blocks, a physical block may be partially invalidated after write operations. We adopt a Block State Table (BST) to keep track of the valid/invalid status of compressed blocks in a physical block. The existing F2FS SIT (Segment Information Table) stores the valid/invalid status of data at the block level. Our BST is an extension to the SIT and is protected by the checkpoint mechanism. Let a physical block contain up to N compressed blocks. The BST extension uses a counter and a validity bitmap of the compressed blocks, which require $\log_2 N$ and N bits, respectively.

[2]In this paper, L2P mapping refers to the mapping of a file block offset (LBN within a file) to a storage address (PBN in storage).

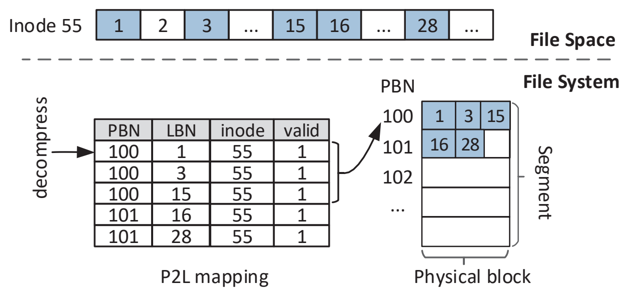


Figure 9: Decompression for read-critical blocks with physical-to-logical (P2L) mapping.

4.3 Decompression with P2L Mapping

When decompressing a logical block, the file system requires the inode number and file offset of the block to properly load the block into the page cache address space. For a file block that is explicitly requested by a read operation, this information is known to the file system upon the request. However, as shown in Figure 8, when reading the physical block at PBN 200, the file system cannot find such information for the other two compressed blocks in the physical block. Therefore, the file system will have to ignore these two compressed blocks, although they have been brought into memory. Thanks to the reverse (physical-to-logical) mapping, a unique feature of log-structured file systems, our design can identify the inode and file offset of the other two compressed blocks and opportunistically load them into memory. The reverse mapping for F2FS is provided by the Segment Summary Area (SSA) [16]. The original purpose of SSA is for the space cleaning procedure to identify the valid/invalid status and logical block address of each block in a victim segment.

Figure 9 shows an example of decompressing blocks based on the P2L mapping. When the first logical block of the file (LBN 1) is requested, the file system reads the physical block at PBN 100, decompress the logical block of LBN 1, and loads it into the page cache. By consulting the reverse mapping, the file system also decompresses and loads the other two compressed blocks along with their inode number (55) and LBNs (3 and 15) into the page cache. Decompression with P2L mapping is essential to the read-critical data compression strategy because BC compacts and compresses correlated executable file blocks into the same physical block.

4.4 Logging and Cleaning

Data Separation. F2FS writes data of different hotness (write frequency) to separate segments through six logging heads to improve cleaning efficacy. FPC inherits these logging heads and employs three new logging heads: The first new logging head is for FC to write compressed read-write files, i.e., SQLite files. A node-data-combined compressed block (Section 3.2.3) is also written to this segment because the node block and data blocks of a rollback journal share the same lifetime. The second is for new, uncompressed executable files, and the third is used by BC to write compressed executable files. New executable files are written to the second new logging head without compression. During background compression, executable files are re-organized for read-critical data and then compressed into the third new logging head. FPC inherits the original F2FS victim selection policy with a slight enhancement. A segment of the largest number of invalid compressed blocks is selected as the victim for space cleaning. Once a victim segment is selected, only valid compressed blocks in the segment are copied for space cleaning.

On-Demand Background Compression. Since executable files do not change after installation or update, background compression can be executed on demand: As described in Section 3.3, a process running in the user space is responsible for collecting the information of read-critical blocks. The process begins to profile an application upon installation or update. For an application under profiling, the file read operations on its executable file are collected for at least 5 rounds of launching. When done profiling an application, the user process issues an `ioctl` call with the name of the executable file of the application to the file system for background compression. The background compression procedure is largely based on the data migration method of the existing segment cleaning procedure of F2FS, but it selects read-critical data for migration and compression. As reported in [26], the typical update period of mobile applications is half a month. In other words, the frequency of background compression will be very low, minimally impacting the file system performance and write traffic volume.

4.5 Design Summary

F2FS Modification. Our implementation of FPC requires enhancements of the F2FS core data structures. The enhancements are described below: 1) File indexing (L2P mapping) requires to augment each direct pointer in the inode and direct node with additional information, which now may refer to a compressed block in a physical block. An original direct pointer is of 32 bits, while a new direct pointer adds 1 bit for compression indication and 3 bits as an offset in a physical block (the compressed block number in a physical block is no larger than 8). The original inode structure has an array of 923 direct pointers, and our design replaces them with an array of 820 upgraded direct pointers. This design slightly reduces the largest file size from 3.94 TB to 3.50 TB. 2) Reverse (P2L) mapping requires to modify the Segment Summary Area (SSA). A physical block can contain multiple compressed blocks, so the structure `f2fs_summary` is extended to represent the reverse mapping information of each compressed block in a physical block. 3) Metadata-level compression requires one additional bit for each Node Address Table (NAT) entry to indicate whether or not a node block is compressed with data blocks into a physical block. Our design adds a separate bitmap to NAT for this purpose.

R/W pattern	File type	Compression policy	Compression type
write-intensive, random-write	db	compression on non-sequential blocks	FC for reduced write stress
(almost) write-only	db-journal	large compression window, metadata-level compression	
write-once, random-read	apk	critical data compaction and compression	BC for fast launching and space saving
write-once, seq-read	dex, odex	large compression window, across physical block boundary	

Figure 10: Summary of compression policies for different types of files and their read/write patterns.

4) The logging of compressed data requires three extra types of segment (Section 4.4). The segment numbers of the three active logging heads (along with those of existing logging heads) are kept in Checkpoint (CP). 5) Segment cleaning requires the valid/invalid status of data. This information is provided by the original Segment Information Table (SIT). If a physical block contains compressed blocks, our Block State Table (BST, Section 4.2) uses five bits to represent the invalid/invalid status of its compressed blocks and three bits for counting invalid compressed blocks.

Crash Consistency. F2FS employs the checkpoint scheme [16, 27] to maintain the data consistency in case of system crashes. The proposed compression solutions extend a set of core data structures including the summary blocks of the added compression logging heads, and *f2fs_inode* and *direct_node* node blocks. Since all the involved metadata extended in the compression designs belong to the data types that have to be protected by existing F2FS checkpoints, system consistency can be maintained by calling the recovery procedure of F2FS.

Fig. 10 is a summary of the access patterns of file types and their associated compression policy.

4.6 Overhead Analysis and Discussions

Space overhead. The primary space overhead of FC approach is related to the Block State Table (BST) for block state tracking. Besides in the storage space, a copy of the BST is made in memory for efficient access. For FC, the largest number of compressed blocks that a physical block can store is empirically set to 5 for a good balance between the metadata overhead and compression space reduction. Each BST entry for a physical block requires 8 extra bits (3 bits for a counter of invalid compressed blocks and 5 bits for a valid/invalid bitmap of compressed blocks). For a 16 GB storage space, the extra DRAM overhead for the counters and bitmaps is 4 MB, which is affordable to modern smartphones. On the other hand, for the metadata-level file compression, the Node Address Table (NAT) is enhanced to locate the compressed node block stored together with the compressed file blocks by adding a 1-bit flag for each NAT entry. As mentioned in Section 4.5, a new bitmap of the flag bits is added to the NAT, and the bitmap requires no more than 0.5 MB storage overhead for a 16 GB storage device. At last, the checksum stored in each physical block costs 4 bytes, and 16 GB storage space introduces a maximal 16 MB space overhead.

General Applicability. It is possible to generalize our pattern-guided compression for unknown file types. This is because the proposed method employs only a few simple properties of file access, including access sequentiality and read-write tendency, as shown in Fig. 10. The file system can adopt a profiling module to observe how files are accessed and apply proper compression strategies accordingly. In particular, the access pattern discovery mechanism is already part of the proposed BC design (Section 3.3.2), and the read-write tendency of new files can be monitored using counters.

Compression Deployment. While compression of user data is feasible at the application level, this study focuses on a file-system approach because it enables deep integration with system-level mechanisms, e.g., prefetching of compressed blocks in the storage and compressing of file-system metadata, which are difficult at the application level.

5 Performance Evaluation

5.1 Experimental Setup

We implemented and evaluated the proposed FPC based on F2FS on a real platform Hikey 960 [28], which is an embedded development board for AOSP. The platform was equipped with a Kirin 960 8-core ARM processor, 4GB of RAM, and a 32GB UFS. The Android and Linux kernel versions were 9.0 and 4.9, respectively. FPC employed LZO [29] as the data compression algorithm. We configured F2FS in the `lfs` mode for using out-of-place updating only. This is because, in the current implementation, F2FS switches to in-place update when the space utilization is extremely high or when performing `fdatasync()` on small files. As mentioned previously, in-place updates risk severe internal fragmentation and thus we turned it off for the best effect of FPC. The system-level energy consumption was measured using the Monsoon power monitor [30].

FPC was evaluated using a set of popular mobile applications, including Facebook (FB), FB Messenger (MS), Google Earth (GE), Firefox(FF), Reddit (RD), Line (LN), Twitter (TW), Instagram (IG), Wechat (WC) and Chrome (CR). The evaluation of foreground compression (FC) was based on the first five applications as they produced a high volume of SQLite-related writes. The evaluation of background compression (BC) involved all the applications. The following methods were evaluated for performance comparison: 1) **Baseline**: The original F2FS without any compression. 2) **Comp**: F2FS with unconditional compression of incoming data. It employed a fixed compression window size of 4 KB and allowed only file blocks of sequential file offsets to share the same physical block. 3) **FPC-N**: The proposed FPC but without metadata-level compression. 4) **FPC**: The full-fledged version of the proposed approach (see Fig. 10). Currently,

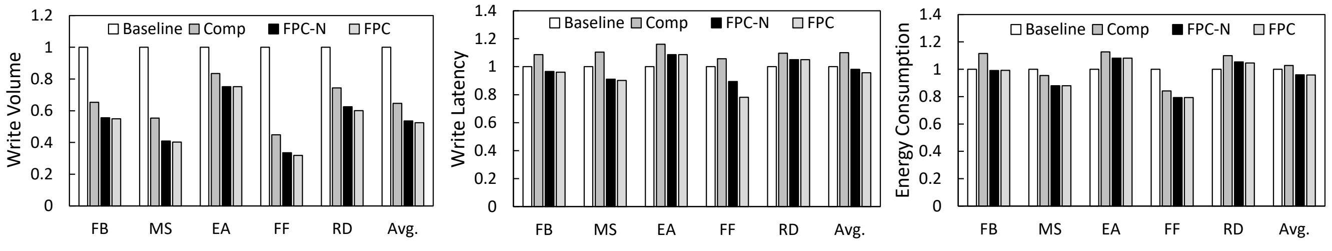


Figure 11: Results of normalized (a) SQLite write volume (b) SQLite write latency and (c) system energy consumption using different compression approaches.

there is little choice of read-write compression file system readily available for Android devices. We implemented the core idea of existing compression file systems in F2FS for performance comparison. Specifically, Comp employed the sequential compression method (Section 3.2.1), which is adopted by Btrfs and JFFS2.

Table 2: Workload characteristics. The percentage reflects the contribution of SQLite files to the total write traffic.

	write count	avg. size	write contribution	compression ratio
FB	3215	39.2 KB	31.6%	0.39
MS	2597	32.4 KB	21.4%	0.25
EA	19919	55.2 KB	99.5%	0.65
FF	17695	33.2 KB	57.8%	0.12
RD	2658	30.5 KB	39.2%	0.40

Because the efficacy of foreground compression (FC) is primarily concerned with the compression ratio of incoming data, we employed content-accurate trace replay for performance evaluation of FC: First, we used each mobile application for 30 minutes and recorded their file operations on SQLite files at the VFS layer. The traces reflected highly common user scenarios, including viewing online news feeds (FB, FF, RD), viewing online satellite maps (EA), and sending/receiving text messages (MG). Each of the recorded operation consisted of an inode number, a file name, a write time, a file block offset, and the data content. The characteristics of the collected traces can be found in Table 2. Second, a user-level process was created to replay the file writes on a set of files with their original SQLite file names, and these writes were captured and compressed by foreground compression inside of F2FS.

We conducted experiments on background compression (BC) with the following steps: For each of the listed applications, we installed an application and performed five times of cold-start launching (with the page cache cleared). The read-critical data set of the application's executable files were identified by BC during the launching. After this, when the system was idle, an `ioctl` request was sent to BC to explicitly request compression of read-critical data in executable files.

5.2 Evaluation Results

This section presents the evaluation results of 1) foreground compression, including write volume, write latency, and energy consumption and 2) background compression, involving space requirement and application launching time.

Write Volume: Figure 11(a) shows the write-traffic volumes bound for SQLite files of Baseline, Comp, FPC-N, and FPC. All results are normalized to Baseline. Compared with Baseline, FPC greatly reduced the SQLite write volume by 47.5%, indicating that foreground compression was highly effective in terms of write stress reduction. In particular, during the 30-minute execution of Facebook, FPC reduced the SQLite write volume from 123.1 MB to 67.5 MB through foreground compression. FPC also outperformed Comp thanks to the use of a larger compression window on SQLite files and the storage of compressed file blocks having random file offsets in the same physical block. The improvement of FPC upon FPC-N depends is highly subject to the application scenario. Under the FF workload, FPC reduced the write volume by 5.3% by appending compressed node blocks of *.db-journal files to their associated compressed data blocks. Overall, the reduction in SQLite write volume contributed to a 26.1% reduction in the entire system write volume (from 3044.1 MB to 2250.9 MB). This large reduction is beneficial to the flash storage lifespan because the wear degree in flash memory is proportional to the volume of inbound write traffic.

Write Latency: Figure 11(b) shows that FPC reduced the write latency of SQLite files by 7.1% on average compared with Baseline. In other words, the benefit of a reduced write I/O count was larger than the cost of data compression. The only exception is Google Earth, whose SQLite files contain a large amount of incompressible multimedia data. Fortunately, many of the incompressible data were not selected for compression thanks to the compression ratio sampling technique mentioned in 3.2.2. By contrast, Comp suffered from the highest write latency because it compressed all incoming data, including those incompressible ones. To observe how high is the time overhead of data compression on the CPU, we measured the CPU utilization of all methods using the `TOP` utility during the trace replay. In our experiment, data compression was affiliated with a specific core. We observed that the average utilization of the involved core was between 3% and 8% under FPC, while that was between 1% and 2% under Baseline. In other words, data compression of FPC only marginally increased the CPU utilization.

Energy Consumption: Energy consumption is a critical

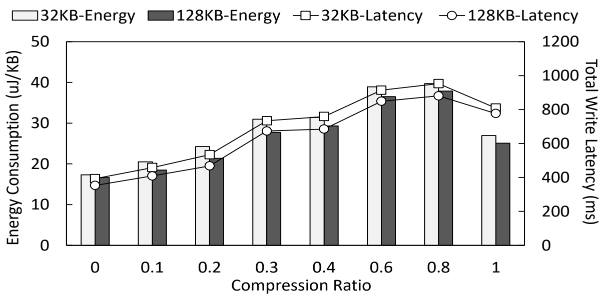


Figure 12: Sensitive study of write latency and energy consumption as compression ratio improves. Data are not compressed for compression ratio of 1.

concern for battery-powered mobile devices [31]. File compression takes extra CPU cycles but reduces flash storage writes. Although file compression induced an extra energy overhead, Figure 11(c) shows that FPC still achieved a lower energy consumption for most of the applications compared with Baseline. In particular, the energy-saving was larger when SQLite files were highly compressible, which was a common case among all applications. In particular, the largest write volume reduction for Firefox also led to the largest energy saving. FPC slightly increased the energy consumption of Google Earth. This is again because Google Earth stored incompressible multimedia contents in SQLite files and a small portion of them underwent ineffective data compression.

SQLite Journaling: In our experiments, although the system default for SQLite journaling was WAL, we found that the five applications (in Table 2) operated 23 out of 33 SQLite files in DELETE mode. In practice, many applications explicitly specify a journal mode to override the system default.

A rollback journal in DELETE mode is write-only except during crash recovery. However, rollback journals in PERSIST mode were reused, and reusing a rollback journal required to read the journal header block. However, due to page caching, the read barely reached the storage during successive transactions. For example, RD read only two blocks from its journal files in a 30-minute session. A similar result was also reported in [22]. The observation confirmed that rollback journals are (almost) write only.

Effect of Compression Ratio: An experiment was conducted to understand the trade-off between the cost and benefit of compression as the compression ratio changes. We created a 100 MB file beforehand and then sequentially overwrote the entire file with large (128 KB) and small (32 KB) file writes. Each file write was followed by an `fsync()` operation. For each test, the data pattern was pre-generated to match the desired compression ratio. As shown in Figure 12, compared with the results without compression (compression ratio=1), the write latency and total energy consumption became lower when the compression ratios were not lower than 0.4 and 0.2, respectively. In other words, when data were highly compressible, compression benefited both write latency and energy consumption. The result indicates when data compress reasonably well, the benefit of our foreground compression design outweighs the cost of compression.

Space Requirement: Typically, executable files require more storage space than SQLite files. Figure 13(a) shows the executable file size of each application with the three methods. With FPC, the total executable file size of all applications was noticeably reduced from 846 MB to 646 MB (reduced by 23.7%). Interestingly, the size reduction of Facebook with FPC was 24%, much better than the reduction 8% achieved by unconditional compression Comp. The reason for the greater reduction of FPC is that background compression BC used a larger compression window on read-critical file blocks and allowed the compressed data to be stored across the physical block boundaries. By contrast, Comp always used a 4KB compression window and did not allow the physical block straddling, incurring a poor space efficacy.

App Launching Time: The application launching time shown here is the value reported by the activity manager `am` called by an `adb` command from a remote PC [32]. As shown in Figure 13(b), compared with Baseline, FPC improved the launching time of applications (except EA and FF, to be explained later), and the reduction was 5.2% on average. By contrast, applications with Comp launched even slower than with Baseline because Comp incurred a high cost of decompression under small, random reads of executable files. FPC significantly outperformed Comp by 22.3% on average in terms of the launching time. This is because our BC method identified read-critical data and then compacted/compressed them into a few physical blocks, leading to a fewer number of block read operations to launch applications. As Figure 13(c) shows, while Comp and Baseline required comparable numbers of block read requests to launch an application, FPC produced much fewer block read requests. In particular, compared with Baseline, FPC reduced the total block read count from 469 to 386, thus speeding up launching LN by 14.8%.

FPC marginally increased the launching time of EA and FF (2 out of the 10 applications). As Figure 13(c) shows, EA required a small number of block reads to launch. We also discerned that the launching of EA was CPU-intensive, and therefore the launching of EA did not much benefit from a reduced I/O count. FF is another case, for which the launching involved sequential reads mostly. Nevertheless, in summary, FPC successfully reduced the I/O cost through read-critical data compression, and the reduction concealed the time overhead of decompression and accelerated application launching.

We use *Decompression Amplification Ratio* (DAR), which has been defined in Section 2.2, to show how much non-read-critical share the same physical block with read-critical data. Table 3 shows the DAR values of the 10 applications. The DAR values of 8 out of the 10 applications were large than 2, indicating that more than one half of a physical block was occupied by compressed blocks of non-read-critical data. The DAR values of Firefox (FF) were very low, explaining why our FPC did not improve the launching time of FF in Fig-

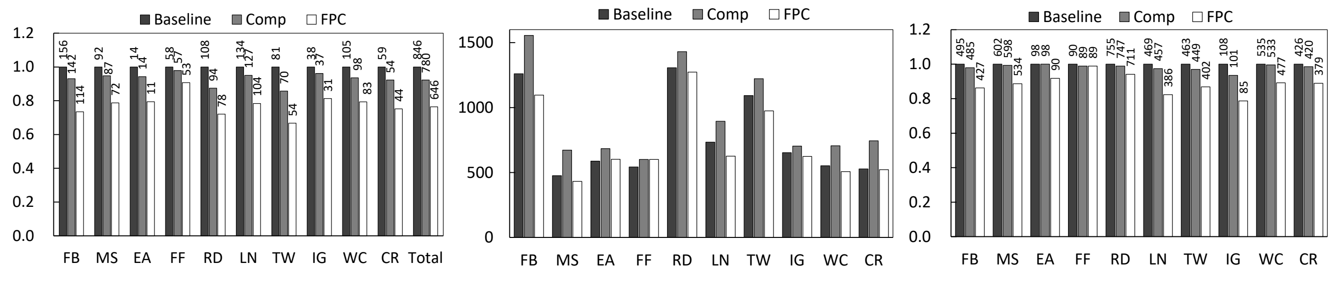


Figure 13: Results of (a) executable file sizes (unit: MB). (b) application launching time (unit: ms). (c) block read number during application launching. The values of block read numbers are marked above each bar.

ure 13(b). Overall, most of the profiled applications had high DAR values, and therefore we believe that small, random reads on executable files are a common problem of mobile applications. Provided that the launching process of an application is not CPU-intensive, the proposed FPC approach can help accelerate the launching process.

Table 3: DAR values of 10 mobile apps.

	FB	MS	EA	FF	RD	LN	TW	CR	IG	WC
DAR	3.8	3.1	2.5	1	2.1	2.6	2.7	3.2	3.7	1.4

6 Related Work

Host Level Compression: In-place update file systems with compression support, such as JFFS2 [11], NTFS [33], and compressed ext2 [34], could suffer from a low space efficacy if the new compressed data size was larger than the old version. Burrows et al. [19] proposed an on-line data compression approach by leveraging the out-of-place write behaviors of a log-structured file system to avoid the above problem. Compression techniques were also recommended in log-structured systems, e.g., enterprise storage system Purity [20] and the database engine Rose [35]. Btrfs [10] was a B-tree file system with compression support. It sequentially compressed incoming data in the fixed granularity upon file updates and then wrote the compressed blocks to a new extent. However, the above studies paid little attention to the compression optimization for small file writes, which could incur a poor compression efficacy for mobile systems.

To reduce read amplification for mobile applications, Zhang et al. [13] proposed a compression framework based on the FUSE file system which compressed read-only files only and required additional decompression hardware. A compressed read-only file system (EROFS) was introduced to save storage space and improve read performance [12]. It leveraged the fixed-sized output compression to reduce read amplification, but only sequentially compressed data without paying attention to the unfriendly random reads of mobile applications that could degrade the decompression efficacy. FPC exploited the compression-friendly structures of log-structured file systems. More importantly, FPC took advantage of the unique file access behaviors of mobile devices and addressed the impact of decompression on user-perceived latencies.

Device Level Compression: There are several solutions for device compression. Zhang et al. [36] proposed a device-side in-place delta compression technique to reduce write stress on SLC-mode flash blocks. Ji et al. [23] proposed a firmware-based compression that selectively compressed data in eMMC devices. Several enterprise storage system vendors including Nimble [37] and Pure Storage [38] had announced their compression-enabled enterprise storage devices. Nevertheless, unconditional device-side data compression is not aware of much useful host information, e.g., the critical reads associated with executable files, and hence it is difficult to optimize decompression latency and improve user experience. Another critical problem of device compression is that recent Android versions have been equipped with block encryption [39], which renders the encrypted data uncompressible.

7 Conclusion

This paper proposed FPC, a file access pattern guided compression framework, to reduce write stress and save storage space for mobile devices. First, the compression was performed at the foreground to selectively compress write-mostly, highly compressible files that produced many small data updates to reduce write stress. Second, background compression re-grouped and compressed critical blocks in executable files to reduce the application launching latency and improve space utilization. The proposed FPC approach was implemented on a real mobile device and experimental results showed both the write traffic and the executable file size was substantially reduced. FPC also reduced the application launching time.

Acknowledgement

We would like to thank our shepherd Youjip Won and the anonymous reviewers for their valuable comments and guidance. This work is partially supported by the Natural Science Foundation of Jiangsu Province (BK20200462), Ministry of Science and Technology of Taiwan (107-2628-E-009-002-MY3 and 109-2221-E-009-075), Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CityU 11204718 and 11218720), NSFC 62072177, and Shanghai S&T Project (20ZR1417200).

References

[1] Smartphone os market share. https://www.idc.com/promo/smartphone-market-share/, 2018.

[2] Umar Farooq and Zhijia Zhao. Runtimedroid: Restarting-free runtime change handling for android apps. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'18)*, pages 110–122. ACM, 2018.

[3] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting storage for smartphones. *ACM Transactions on Storage (TOS)*, 8(4), 2012.

[4] Wongun Lee, Keonwoo Lee, Hankeun Son, Wook-Hee Kim, Beomseok Nam, and Youjip Won. WALDIO: eliminating the filesystem journaling in resolving the journaling of journal anomaly. In *Proceedings of USENIX ATC*, pages 235–247, 2015.

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

[6] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O stack optimization for smartphones. In *Proceedings of ATC, 2013*, pages 309–320, 2013.

[7] Younghwan Go, Nitin Agrawal, Akshat Aranya, and Cristian Ungureanu. Reliable, consistent, and efficient data sync for mobile apps. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST 15)*, pages 359–372, 2015.

[8] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. Improving file system performance of mobile storage systems using a decoupled defragmenter. In *Proceedings of Annual Technical Conference (USENIX ATC 17)*, pages 759–771. USENIX Association, 2017.

[9] Sangwook Shane Hahn, Sungjin Lee, Inhyuk Yee, Donguk Ryu, and Jihong Kim. Fasttrack: Foreground app-aware I/O management for improving user experience of android smartphones. In *Proceedings of Annual Technical Conference (USENIX ATC 18)*, pages 15–28. USENIX Association, 2018.

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

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

[12] Xiang Gao, Mingkai Dong, Xie Miao, Wei Du, Chao Yu, and Haibo Chen. EROFS: A compression-friendly read-only file system for resource-scarce device. In *Proceedings of Annual Technical Conference (USENIX ATC)*. USENIX Association, 2019.

[13] 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.

[14] Kisung Lee and Youjip Won. Smart layers and dumb result: IO characterization of an android-based smartphone. In *Proceedings of the tenth ACM international conference on Embedded software (EMSOFT)*, pages 23–32. ACM, 2012.

[15] S. Bhattacharya A. Dilger A. Tomas A. Mathur, M. Cao and L. Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium*, pages 21–33. Citeseer, 2007.

[16] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[17] Ming-Chang Yang, Yuan-Hao Chang, Chei-Wei Tsao, and Chung-Yu Liu. Utilization-aware self-tuning design for TLC flash storage devices. *IEEE Trans. VLSI Syst.*, 24(10):3132–3144, 2016.

[18] Samsung Semiconductors. 3D TLC NAND to beat MLC as top flash storage. EETimes, 2015.

[19] Michael Burrows, Charles Jerian, Butler Lampson, and Timothy Mann. On-line data compression in a log-structured file system. In *ASPLOS*, pages 2–9. Citeseer, 1992.

[20] John Colgrove, John D Davis, John Hayes, Ethan L Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. Purity: Building fast, highly-available enterprise flash storage from commodity components. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1683–1694. ACM, 2015.

[21] M. Son, J. Ahn, and S. Yoo. Nonvolatile 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, 2018.

[22] Taeho Hwang, Myungsik Kim, Seongjin Lee, and Youjip Won. On the I/O characteristics of the mobile web browsers. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC'18)*, pages 964–966, 2018.

[23] Cheng Ji, Li-Pin Chang, Liang Shi, Congming Gao, Chao Wu, Yuangang Wang, and Chun Jason Xue. Lightweight data compression for mobile flash storage. *ACM Trans. Embed. Comput. Syst.*, (5s):183:1–183:18, 2017.

[24] Danny Harnik, Ronen Kat, Dmitry Sotnikov, Avishay Traeger, and Oded Margalit. To zip or not to zip: Effective resource usage for real-time compression. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST 13)*, pages 229–241, 2013.

[25] Yongsoo Joo, Junhee Ryu, Sangsoo Park, and Kang G Shin. Fast: Quick application launch on solid-state drives. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 259–272, 2011.

[26] Dmitry Garbar. How often should you update your mobile app? https://www.apptentive.com/blog/2018/12/27/how-often-should-you-update-your-mobile-app/, 2018.

[27] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and checking file system crash-consistency models. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 83–98, 2016.

[28] Hikey 960. https://www.96boards.org/product/hikey960/.

[29] LZO real-time data compression library. http://www.oberhumer.com/opensource/lzo/.

[30] Monsoon power monitor. http://www.msoon.com/LabEquipment/PowerMonitor/, 2016.

[31] Jayashree Mohan, Dhathri Purohith, Matthew Halpern, Vijay Chidambaram, and Vijay Janapa Reddi. Storage on your smartphone uses more energy than you think. In *Proceedings of HotStorage*. USENIX Association, 2017.

[32] Android debug bridge (adb). https://developer.android.com/studio/command-line/adb.html.

[33] Ntfs compressed files. http://www.ntfs.com/ntfs-compressed.htm.

[34] e2compr. http://e2compr.sourceforge.net/.

[35] Russell Sears, Mark Callaghan, and Eric Brewer. Rose: Compressed, log-structured replication. *Proceedings of the VLDB Endowment*, 1(1):526–537, 2008.

[36] 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 USENIX Conference on File and Storage Technologies (FAST 16)*, pages 111–124, 2016.

[37] Casl architecture in nimble storage. http://www.nimblestorage.com/products/architecture.

[38] Flashreduce data reduction in pure storage. http://www.purestorage.com/flash-array/flashreduce.html.

[39] Android Encryption. https://source.android.com/security/encryption/.