EROFS: A Compression-friendly Readonly File System for Resource-scarce Devices

Xiang Gao, *Huawei Technologies Co., Ltd.;* Mingkai Dong, *Shanghai Jiao Tong University;*
Xie Miao, Wei Du, and Chao Yu, *Huawei Technologies Co., Ltd.;.*
Haibo Chen, *Shanghai Jiao Tong University / Huawei Technologies Co., Ltd.*

https://www.usenix.org/conference/atc19/presentation/gao

This paper is included in the Proceedings of the
2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the
2019 USENIX Annual Technical Conference
is sponsored by USENIX.

EROFS: A Compression-friendly Readonly File System for Resource-scarce Devices

Xiang Gao[1], Mingkai Dong[2], Xie Miao[1], Wei Du[1], Chao Yu[1], and Haibo Chen[2,1]

[1]Huawei Technologies Co., Ltd.
[2]Shanghai Jiao Tong University

Abstract

Smartphones usually have limited storage and runtime memory. Compressed read-only file systems can dramatically decrease the storage used by read-only system resources. However, existing compressed read-only file systems use fixed-sized input compression, which causes significant I/O amplification and unnecessary computation. They also consume excessive runtime memory during decompression and deteriorate the performance when the runtime memory is scarce. In this paper, we describe EROFS[1], a new compression-friendly read-only file system that leverages fixed-sized output compression and memory-efficient decompression to achieve high performance with little extra memory overhead. We also report our experience of deploying EROFS on tens of millions of smartphones. Evaluation results show that EROFS outperforms existing compressed read-only file systems with various micro-benchmarks and reduces the boot time of real-world applications by up to 22.9% while nearly *halving* the storage usage.

1 Introduction

Low-end smartphones with relatively low price are still prevalent in the market [18, 25], especially in developing countries. At a price, such devices are usually equipped with limited resources in both capacity and performance. For example, a low-end Android smartphone may have 1-2GB runtime memory and 8-16GB slow eMMC storage [19, 21, 22]. Even worse, the Android operating system itself can consume more than 3GB in storage, leaving scarce storage space available to users [46]. Even for high-end smartphones, the increasing storage and runtime memory consumption of popular or resident apps usually render a device resource-scarce for both user-initiated and system-initiated operations.

File systems with compression support, or compressed file systems, can be used to release more space to the users by transparently compressing/decompressing file data upon accesses. However, such file systems usually consume more resources and yield notably worse performance during compression/decompression. Thus they are not suitable for resource-limited devices, especially smartphones, on which user experience has the top priority.

Fortunately, for partitions with read-only data, such as the */system*, */vendor* and */odm* partitions of Android, the file system can be made read-only to boost the performance by simplifying the structures and designs for file changes. However, existing compressed read-only file systems, such as Squashfs [10, 11], usually cause notable degradation on access performance and incur extra memory usage during decompression. One key issue is that such file systems use fixed-sized input compression, in which file data is divided into fixed-sized chunks (e.g., 128KB) and each chunk is compressed individually. The fixed-sized input compression incurs significant read amplification and excessively unnecessary computations (§2.2). Even worse, they usually require a huge amount of runtime memory, which is scarce on low-end smartphones or heavily-used high-end smartphones (§2.2).

To save the storage space and retain high performance with low memory overhead, we design and implement EROFS, an enhanced read-only file system with compression support. EROFS introduces the fixed-sized output compression, which compresses file data to multiple fixed-sized blocks, to significantly mitigate the read amplification problem and reduce unnecessary computations as much as possible. By exploiting the characteristics of compression algorithms (such as LZ4), EROFS designs different memory-efficient decompression schemes to reduce extra memory usage during the decompression. EROFS also adopts a set of optimizations that carefully ensure guaranteed user experience.

The main contributions of this paper include:

- A study of existing compressed file systems which reveals the performance issues on resource-hungry devices (§2).
- A fixed-sized output compression scheme that signifi-

[1] Short for Enhanced Read-Only File System. It has been upstreamed to Linux 4.19 as a major feature and integrated into Huawei's Smartphone Operating System (called EMUI) as a top feature (`https://consumer.huawei.com/en/emui/`) of version 9.1.

cantly mitigates the read amplification issue (§3.1).

- A set of novel decompression schemes for both memory-efficiency and high performance (§3.3).
- An evaluation of EROFS against other file systems to validate the effectiveness of EROFS (§5) and a study on the deployment experience of EROFS on tens of millions of smartphones (§6).

2 Background and Motivation

2.1 Low user-perceived storage space

Smartphones are usually resource-scarce due to the cost constraint. Meanwhile, the space occupied by the Android operating system is constantly increasing. Fig. 1 shows the */system* partition size in stock Android factory images [6] for different Android versions. The sparse image strips off all zero blocks and thus only contains all effective data; while the raw image is the actual space consumed once stored into the devices. From the figure, we can see the data size of the */system* partition increases from 184MB in Android 2.3.6 to 1.9GB in Android 9.0.0. Besides the trend of increasing the effective data size, we can also see a large number of zero blocks in Android 7 and 8, which also consume large space. For Android 9, the zero blocks are significantly less, which is due to the support of data block deduplication [20] in the ext4 file system. Besides the */system* partition shown in Fig. 1, there are other space-consuming partitions for Android such as */vendor*, */oem* and */odm* [8]. As reported in previous work [46], the space used by the whole Android system itself is increasing and far larger than what we show here. For example, Android 6.0.0 consumes 3.17GB storage after a factory-reset [46].

Meanwhile, the storage consumption of Android applications also keeps growing. As reported by Google Play, by early 2017, the average app size has quintupled compared with that at the time Google starts its Android application marketplace [45]. As a result, the storage capacity of low-end smartphones available for users is rather small. Further, many top apps for smartphones tend to consume a huge amount of memory, leaving only a small amount of memory for system-initiated operations even on a high-end smartphone.

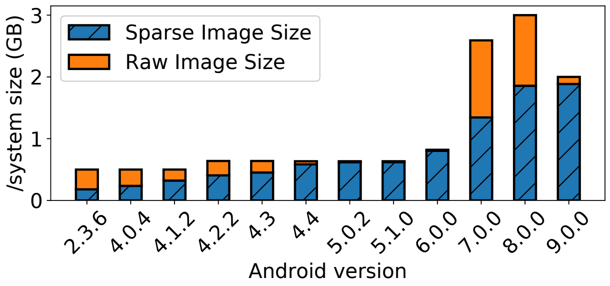


Fig. 1: Android /system partition sizes

Compressed file systems. One intuitive approach to unleashing more spaces for users is adopting compressed file systems, which exposes standard file interfaces to the applications but transparently compress and decompress file data during file writes and reads.

Btrfs [2] is a modern B-tree file system with compression support. When compression is enabled, the file data is divided into multiple 128KB chunks and compressed separately. Each of the compressed chunks will be stored in an extent, which is a run of contiguous blocks that store data sequentially. The locations of these extents are recorded as indexes in the B-tree structures. To read the file data, the corresponding extents are read from the storage and the whole chunks are decompressed. To update a file, the new data is compressed and written to new extents, and then the indexes are updated. To read the file data, Btrfs reads the corresponding extents from the storage and decompresses the whole chunks. To update a file, Btrfs compresses the new data, writes it to new extents, and updates the indexes.

Btrfs is a general-purpose file system, so its internal structures must consider efficient data modifications and cannot be aggressively optimized for compression. Furthermore, compression is not the only metric. The memory consumption during decompression should also be constrained.

For devices like smartphones, performance and responsiveness are important key metrics that cannot be compromised. Hence, with the burden of efficient data modification, Btrfs can hardly satisfy the requirements of both performance and compression efficiency, as we will show later in the evaluation (§5).

Compressed read-only file systems. Considering the access patterns of partitions in Android, we find that system resources are rarely modified once the Android operating system is installed. We can thus use compressed read-only file systems on read-only partitions to reduce the space consumption for system resources while retaining the performance. Unlike compressed read-write file systems which are complicated by data modifications, compressed read-only file systems exclude data updates by design, which exposes more opportunities for higher compression ratio and faster data reads.

Squashfs [11] is a widely-used compressed read-only file system in Linux with many features and moderate performance. It supports several compression algorithms, and the chunk (i.e., compression input) size can be chosen from 4KB to 1MB. In Squashfs, metadata can be compressed, and inodes and directories are stored more compactly. File data is compressed chunk by chunk, and the compressed data blocks are stored sequentially. The compressed sizes of each original data chunk are stored in a list within the inode. These sizes are used to locate the position of compressed blocks during decompression.

2.2 Deficiency of existing readonly file systems

Compressed read-only file systems are designed to minimize storage usage. However, applying existing compressed read-only file systems on resource-scarce smartphones can induce significant overhead on both performance and memory consumption. For example, we first tried to use Squashfs for the read-only partitions on Android. While the system boots successfully with Squashfs, booting the camera application requires tens of seconds even with light background workloads.

Why is there such a huge performance slowdown? We conducted a detailed study of Squashfs with default configuration using microbenchmarks and uncover that the performance degradation mainly originates from two parts. The first one is I/O amplification. We used FIO [23] to evaluate the basic performance of Squashfs. When Android sequentially reads 16MB from the 1GB enwik9 [40] file stored in Squashfs, the actually issued I/O is 7.25MB. While the number looks decent regarding compression, Squashfs issues 165.27MB I/O reads when Android reads 16MB randomly. Moreover, when Android reads the first 4KB of every 128KB, reading 16M file data issued as much as 203.91MB I/O read. The difference suggests that when Squashfs reads some data that is not decompressed and cached before, the size of data requested is significantly amplified.

The second reason is extra memory consumption. The total memory consumption after sequentially reading the 1GB enwik9 file on Squashfs is about 1.35GB, which suggests that decompression in Squashfs requires a significant amount of temporary memory compared to the size of the original data needed. This causes high pressure to Android since memory is a key factor for user experience given that Android and its apps already consume a large amount of memory. On one hand, allocating memory during decompression may trigger memory swapping, which involves victim selections and I/Os with high cost. On the other hand, consuming much extra memory during decompression affects other components or applications by dropping their cached data or swapping out useful memory pages.

We further analyzed the design and implementation of Squashfs and found the following two defects.

Fixed-sized input compression. Existing file systems compress original data in a fixed-sized chunk, generating variable-sized compressed data. As shown in Fig. 2(a), Squashfs takes a fixed-sized data (e.g., a 128KB chunk) as the input of a single invocation of the compression algorithm. The compression algorithm then generates the compressed data whose size depends on the content of the input data. The compressed data of one file is usually compacted in the original data order, to reduce wasted space in the first and the last blocks of each compressed chunk.

Such a compression approach appears decent but has a notable deficiency due to amplified I/O and wasted computation. For example, in Fig. 2(b), the application wants to get the first byte of the 128KB chunk. To satisfy the application's request, the Squashfs has to read all compressed data from block 1 to block 7. Considering the minimal requested block size of the underlying storage devices is 4KB, the I/O is amplified 7 times! This is because the file system must read *all* related compressed blocks, even if the number of compressed blocks is very large. Even worse, even if not all data stored in the first block and the last block are useful for the decompression, they must be read from the storage altogether. In the example, the shadowed parts of block 1 and block 7 in Fig. 2(b) contribute nothing to the decompression but have to be read from the storage. Besides, the decompression process for useless data also causes huge CPU wastes that lead to high performance interference of other running apps (such as the Camera mentioned before).

One possible mitigation would be reducing the input chunk size to 4KB in Squashfs. While this might alleviate the I/O amplification, this non-trivially reduces the compression ratio and incurs higher CPU utilization, as we will show in section 5.

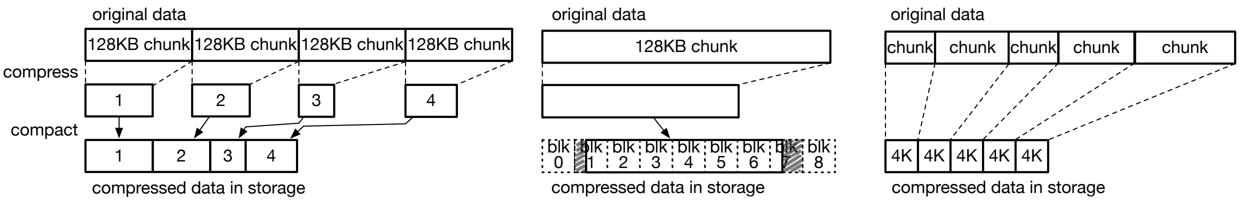
Massive memory consumption and data movements. The other defect we found is that Squashfs requires massive temporary memory during the decompression. Upon file read requests, Squashfs will first look up the metadata to get the number of related compressed blocks. It then allocates memory (e.g., the *buffer_head* structure) for each of the compressed blocks, and issues I/O reads to fetch the compressed blocks from the storage to the allocated *buffer_head*s. Since the buffers in *buffer_head*s of adjacent compressed blocks might not have continuous virtual addresses, Squashfs has to copy data in the *buffer_head*s of all compressed blocks to a single continuous buffer. Then, the compression algorithm decompresses all original data and puts them in a temporary output buffer. Finally, Squashfs copies the original data from the temporary output buffer to the corresponding page cache pages.

From the above routine, two pre-allocated temporary buffers are used and an array of *buffer_head*s are dynamically allocated for the decompression. The number of *buffer_head* needs to be large enough to store all compressed blocks. However, allocating such a large amount of memory can cause severe performance degradation under a low-memory situation.

In addition to extra memory allocation, there are two data movements during decompression: from the *buffer_head*s to the temporary input buffer, and from the temporary output buffer to the page cache. These two data movements also cause performance overhead since, most of the time, the compression/decompression algorithm is bottlenecked by memory accesses.

The above two defects in Squashfs reveal two challenges when designing a compressed read-only file system for resource-scarce smartphones.

- How to reduce I/O amplification during the decompression without sacrificing the compression ratio?
- How to reduce memory consumption during the decompression to prevent performance degradation?



(a) Fixed-sized input **(b) Fixed-sized input issues** **(c) Fixed-sized output**

Fig. 2: Compression approaches

3 EROFS:Enhanced Compressed File System

This section presents the design of EROFS, a compression-friendly readonly file system which overcomes the deficiency of prior systems. The key design of EROFS includes fixed-sized output compression, cached I/O and in-place I/O, and memory-efficient decompression.

3.1 Fixed-sized output compression

To overcome the read amplification incurred by the fixed-sized input compression, EROFS adopts a different compression approach: fixed-sized output compression.

To generate fixed-sized output, EROFS compresses the file data using a sliding window, whose size is a fixed value and can be configured during image generation. The compression algorithm is invoked multiple times until all file data is compressed. For example, with a 1MB sliding window, EROFS feeds the compression algorithm with 1MB original data at a time. The algorithm then compresses the original data as much as possible until all 1MB data is consumed or the consumed data can generate exactly 4KB compressed data. The remaining original data is combined with more data, forming another 1MB original data for the next invocation of compression. Fig. 2(c) depicts the fixed-sized output compression, in which variable-sized original data is compressed to 4KB blocks.

There are several benefits of using fixed-sized output compression compared to the fixed-sized input one. First, as what we will show in the evaluation (§5.3), it has better compression ratio under the same compression unit size. This is reasonable since the fixed-sized output compression can compress data as much as possible until the output limit is reached, while the fixed-sized input compression can only compress a fixed size of data at a time. Second, during the decompression, only the compressed blocks that contain the requested data will be read and processed. In the previous example where a single original block is requested, at most two compressed blocks will be read and decompressed. Third, as we will show later in §3.3, the fixed-sized output compression makes it possible to do in-place decompression, which further reduces the memory consumption in EROFS.

3.2 Cached I/O and in-place I/O

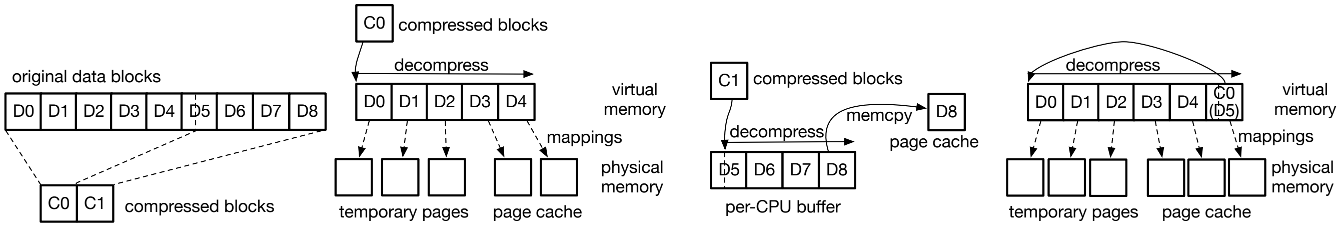
Before the actual decompression, EROFS needs space to store the compressed data retrieved from the storage. While this is costly for fixed-sized input compression due to excessive memory allocation and even page swapping, fixed-sized output compression would incur much less cost since EROFS clearly knows that each compression only retrieves up to two compressed blocks. There are two strategies for EROFS: cached I/O and in-place I/O. EROFS uses cached I/O for compressed blocks that will be partially decompressed. EROFS manages a special inode whose page cache stores compressed blocks indexed by the physical block number. Thus, for cached I/O, EROFS will allocate a page in the special inode's page cache to initiate the I/O request, so that the compressed data will be directly fetched to the allocated page by the storage driver.

For compressed blocks that will be completely decompressed, EROFS uses in-place I/O. On each file read, VFS will allocate pages in the page cache for the file system to put file data. For any one of these pages that contains no meaningful data before the decompression, we call it a *reusable page*. For in-place I/O, EROFS uses the last *reusable page* to initialize the I/O request.

Both I/O strategies are necessary. For cached I/O, partially decompressed blocks are cached in the special page cache, so that subsequent reads to the uncompressed part can use these blocks without invoking additional I/O requests. For blocks that are fully decompressed, they are unlikely to be used later since all decompressed data is stored in the page cache, which can serve subsequent reads without decompression. Thus, cached I/O vainly increases the memory spike due to page allocations for fully compressed blocks, while not contributing to subsequent file reads. In such cases, in-place I/O avoids unnecessary memory allocation, which relieves the memory pressure especially when there are many in-flight file read requests on different compressed blocks. Note that although it is possible to put the compressed block on the stack, it is not recommended to do so since the stack size is limited to be 16KB [14] and it is not easy to know how many bytes of the stack are still available.

3.3 Decompression

After loading compressed data into memory, we illustrate how EROFS decompresses data both fast and memory-efficiently. Examples in this section are based on Fig. 3(a) where the first five blocks (D0 to D4) and part of the block D5 are compressed to block C0, and the rest blocks are compressed to block C1. In this subsection, we only introduce



(a) Compression **(b) Vmap decompression** **(c) Buffer decompression** **(d) In-place decompression**

Fig. 3: Decompression

how a single compressed block is decompressed since, for read requests containing data in multiple compressed blocks, the compressed blocks are decompressed one by one similarly. For example, to read blocks D4 to D6 in Fig. 3(a), C0 is firstly decompressed to get D4 and the first part of D5; then C1 is decompressed to get the rest of D5 and D6.

Vmap decompression To get the data in block D3 and D4, EROFS first reads the compressed block C0 from the storage and stores it in the memory. Then EROFS will decompress it in following steps.

1. Find the largest needed block number that is stored in the compressed block (C0), which is the fifth block (D4) in the example. As an advantage, EROFS only needs to decompress the first five blocks (D0 to D4), rather than decompressing all original data blocks.
2. For each of the data blocks that need to be decompressed, find memory space to store them. In the example shown in Fig. 3(b), EROFS allocates three temporary physical pages to store D0, D1, and D2. For the requested two blocks, D3 and D4, EROFS reuses the two physical pages that have been allocated by VFS in the page cache.
3. Since the decompression algorithm requires continuous memory as the destination of decompression, EROFS maps physical pages prepared in the previous step into a continuous virtual memory area via the `vmap` interface.
4. If it's in-place I/O, in which case the compressed block (C0) is stored in the page cache page, EROFS also needs to copy the compressed data (C0) to a temporary per-CPU page so that the decompressed data won't overwrite the compressed data during the decompression.
5. Finally, the decompression algorithm is invoked, and data in the compressed block is extracted to the continuous memory area. After the compression, the three temporal physical pages and the virtual memory area can be reclaimed, and the requested data has already been written to the corresponding page cache pages.

Per-CPU buffer decompression The above decompression approach causes two problems. The first one is that it is still required to dynamically allocate physical memory pages, which increases the memory pressure on memory-constrained devices. The second problem is that using `vmap` and `vunmap` on each decompression is inefficient.

EROFS leverages per-CPU buffers to mitigate the problems when the decompressed data is less the four pages. As shown in Fig. 3(c), a four-page memory buffer is pre-allocated for each CPU as the per-CPU buffer. For decompression that extracts no more than four blocks of data, EROFS decompresses the data to the per-CPU buffer and then copy the requested data to the page cache pages. In the example demonstrated in Fig. 3(c), data in block D8 is requested. The compressed data in C1 is directly decompressed to the per-CPU buffer, and the content of D8 is copied to the page cache page via `memcpy`.

The length of the per-CPU buffer is empirically decided, but it can effectively eliminate memory allocations since the per-CPU buffer can be reused across different decompressions. The per-CPU buffer decompression is a cost-effective trade-off which mitigates issues in the vmap decompression while introducing extra memory copies.

Rolling decompression To avoid the overhead of `vmap` and `vunmap` and eliminate other dynamic page allocations, EROFS allocates a large virtual memory area[2] and 16 physical pages for each CPU.

Before each compression, EROFS uses the 16 physical pages, along with the physical pages of the page cache to fill in the VM area, so that step 2 and step 3 of the vmap decompression can be skipped.

EROFS uses LZ4 as the compression algorithm, which needs to look backward at no more than 64KB of the decompressed data [7]. Thus, for a compression that extracts more than 16 pages, EROFS can reuse the physical page mapped 16 virtual pages (i.e., 64KB) before. For example, in Fig. 4, the virtual addresses to store blocks D0 to D15 are backed by the 16 physical pages. The virtual page of D16 can be backed by the same physical page with D0 since each virtual address in D16 is 64KB away from the corresponding address in D0. D17 is backed in the same way by the physical page used by D1. D18, which is requested by the file read, uses the physical page of the page cache.

As a result, 16 physical pages are sufficient for any decompression cases by using such a rolling decompression.

[2] A virtual memory of 256 pages is sufficient for all the workloads we have met.

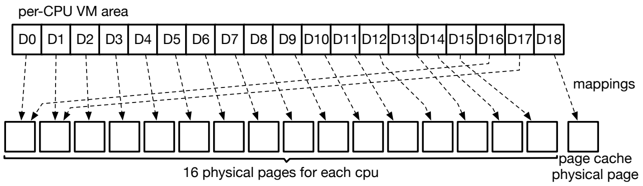


Fig. 4: An example of rolling decompression

In-place decompression In step 4 of the vmap decompression approach, the compressed data is moved to a temporary per-CPU page to avoid data not yet compressed from being overwritten by the compressed data (Fig. 5).

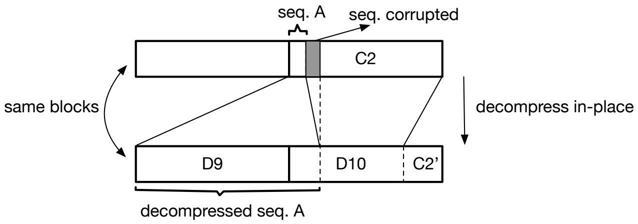


Fig. 5: An example compressed block (C0) that cannot be decompressed in-place. The decompressed data of sequence A corrupts the next sequence (the shadow part) which has not been decompressed.

However, if such a situation will never happen for a compressed block, we can decompress it in-place to avoid the extra memory allocation and memory copies. EROFS simulates the decompression during *mkfs* and marks whether a compressed block can be decompressed in-place in the block index. During the decompression of a block that can be decompressed in-place, step 4 is skipped. In our tested workload enwik9 in §5, 99.6% compressed blocks can be decompressed in-place; thus, most blocks can benefit from the in-place decompression as long as they are retrieved by in-place I/O.

4 Implementation

We have implemented EROFS as a Linux file system and upstreamed the common part of EROFS to Linux kernel[3].

In the current implementation, we use 4KB as the fixed output size since it is the minimal unit of page management and storage data transfers, and thus I/O amplification can be minimized. We support LZ4 (v1.8.3) as the compression algorithm since it has the fastest decompression speed and good compression ratio in our case. Other compression algorithms, such as LZO, can be supported once they are modified to provide fixed-sized output compression interfaces. Only the file data is compressed in EROFS; metadata such as inode and directory entries is stored without compression.

Currently, EROFS is still under active development and new features are constantly shipped into the smartphones after a rigorous commercial testing process. Hence, we introduce two versions of EROFS: 1) the latest version with all features and optimizations presented in this paper; 2) the commercially-deployed version, which has all features and optimizations except the rolling decompression and the in-place decompression. The two versions are also different in decompression policies which we will illustrate in §4.2.

4.1 EROFS image layout

Fig. 6 shows the layout of an EROFS image. As in other file systems, a super block is located at the beginning of the image. Following the super block, metadata and data may be stored in a mixed style without constraints on the order.

In the current implementation, metadata and data of a file are stored together for better locality. For each file, as shown in Fig. 6, an inode is stored at the beginning, followed by blocks containing the extended attributes (i.e., xattrs) and the block index. Blocks for compressed or uncompressed file data (encoded blocks) are stored at the end of each file.

Since an inode can be placed anywhere in the image, the inode number is calculated from the position of an inode, so that the inode can be quickly located. Blocks for xattrs and the block index are omitted if a file contains no xattrs or is uncompressed. Further, xattrs, the block index and file data can also be inlined within an inode if possible, which reduces storage overhead and decreases the number of I/O requests since the inlined data/metadata is read along with the inode.

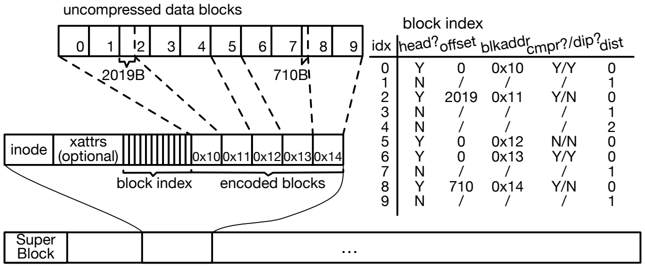
The block index is used to quickly locate the corresponding encoded block for read requests. Fig. 6 shows an example block index for a regular file containing ten blocks before compression. The block index is an array of 8B-length entries, each of which corresponds to a data block before compression. Each entry indicates whether the corresponding data block is the head block (the boolean `head` field in Fig. 6), which starts a new encoded block. If so, the encoded block address (`blkaddr`), the offset of the first byte in the new encoded block (`offset`), whether the encoded block is compressed (`cmpr`), and whether the block can be decompressed in-place (`dip`) are also stored. If not, there must be a head block before the uncompressed block, and the block number difference to the head block is recorded in `dist`.

For a read request to an uncompressed data block, EROFS gets the block index entry according to the requested block number. For a head block, EROFS decompresses data from the block at `blkaddr`, and if the `offset` is non-zero, EROFS may also need to decompress from the nearest encode block stored before the `blkaddr`. For a non-head block, EROFS calculates the location of the corresponding head block according to the stored `dist`, and starts to decompress until the requested block data is decompressed.

Some data blocks (e.g., block 5 in Fig. 6), which are larger after compression, are not compressed and directly stored as encoded blocks. For these cases, the corresponding `cmpr` fields are set to false (i.e., “N” in the figure).

[3] Since some optimizations are not yet upstreamed, we also make the latest version of the code available at https://github.com/erofs/atc19-erofs and https://github.com/erofs/atc19-mkfs.

Directories are stored similarly as the regular files, except that there is no block index, and the encoded blocks are used to store uncompressed directory entries. For better locality of random accesses in directories, EROFS puts all dirent headers (e.g., inode number, file type, and name length) at the beginning of directory entries part, and places filenames after those headers.



block index

idx	head?	offset	blkaddr	cmpr?/dip?	dist
0	Y	0	0x10	Y/Y	0
1	N	/	/	/	1
2	Y	2019	0x11	Y/N	0
3	N	/	/	/	1
4	N	/	/	/	2
5	Y	0	0x12	N/N	0
6	Y	0	0x13	Y/Y	0
7	N	/	/	/	1
8	Y	710	0x14	Y/N	0
9	N	/	/	/	1

Fig. 6: EROFS image layout and the block index

4.2 Decompression policy

Two versions of EROFS have different decompression policies. The commercially-deployed EROFS uses the per-CPU buffer decompression if less than four original data blocks are to be extracted; otherwise, the vmap decompression is used.

In the latest EROFS, all four decompression approaches are implemented. If there is no more than one data block to be extracted, the per-CPU buffer decompression is chosen. Otherwise, if a compressed block is retrieved using in-place I/O and can be decompressed in-place, EROFS employs the in-place decompression approach which avoids unnecessary memory allocations and memory copies. For other cases where the decompressed blocks can fit in the pre-allocated VM area, EROFS uses the rolling decompression since it beats the vmap decompression with less memory allocation overheads. For any other cases, the vmap decompression approach is adopted.

4.3 Optimizations

Index memory optimization It is possible that EROFS compresses hundreds of pages of original data into a single compressed block. In such a case, EROFS needs hundreds of pointers to keep track of where each page of the original data should be stored. These pointers can consume a large amount of memory. To address such a problem, EROFS tries to store the information with the help of *reusable pages*. If there are more than one VFS allocated pages are *reusable*, EROFS uses the last page to store the compressed data, and the other pages to store some of these pointers during the I/O. Before the actual decompression, these pointers are moved onto the stack, so that the *reusable pages* are free to store the decompressed data.

Scheduling optimization Decompression requires a relatively long time. Thus it is not suitable to be done within the interrupt context. In some file systems, such as Btrfs [2], when compressed data has been fetched to memory, a dedicated thread will be woken up to decompress the data. When the decompression is finished, the reader thread which issues the I/O will be woken up to get the decompressed data from the page cache. To reduce scheduling overhead, EROFS decompresses data in the reader thread, without dedicated threads for decompression. Thus once the compressed data has been fetched to memory, the reader thread will be directly woken up and start decompressing the data.

Cohort decompression Several requests can be in-flight simultaneously. If an original data block is requested on thread A and the corresponding compressed block is being decompressed by another thread named thread B, rather than decompressing the data by itself, thread A can wait for thread B to finish the decompression, and then directly read the decompressed data from the page cache. Such cooperation reuses the decompressed data and prevents a single data being decompressed multiple times.

Image patching Although EROFS is a compressed read-only file system, there are cases such as system upgrade or security patching where the data stored in EROFS needs to be updated. EROFS provides a feature called image patching, which supports partial data updates. Usually, modifying a single bit in the original file data might cause a huge amount of scattered modifications in the compressed data. Instead of modification in-place, image patching places updated data at the end of the EROFS image, and when the corresponding file data blocks are requested, the origin data blocks are firstly decompressed and then the updated data is applied to overwrite the decompressed data in memory. In this way, image patching prevents scattering of changes and supports partial data updates without re-compressing the while file system.

5 Evaluation

We have conducted a set of experiments to answer the following questions:

- How does compression affect the performance of file system read accesses?
- How much memory does EROFS consume during decompression?
- How does EROFS affect the boot time on real-world applications?

5.1 Evaluation setup

By default, we conduct experiments on an ARM development board, HiKey 960, running Android 9 Pie with Linux kernel 4.14. The board is equipped with Kirin 960 (four Cortex-A73 big cores and four Cortex-A53 little cores), 3GB Hynix LPDDR4 memory and 32GB Samsung UFS storage. We also evaluate on two kinds of smartphones in some experiments. The low-end smartphones are equipped with MT6765 (eight Cortex-A53 cores), 2GB memory and 32GB eMMC storage. High-end smartphones run with Kirin 980 (four Cortex-A76 cores and four Cortex-A55 cores), 6GB memory and 64GB UFS storage.

For micro-benchmarks, we run FIO [23], a flexible I/O tester, on various file systems including EROFS, Squashfs, Btrfs, Ext4, and F2FS. We use the latest version of EROFS for micro-benchmark evaluation. Among these file systems, EROFS and Squashfs are designed to be compressed read-only file systems; Btrfs is a file system with compression support, but it is not a file system designed for read-only data; Ext4 is the default file system used by Android; F2FS is a file system designed for mobiles and is widely used in some smartphones.

EROFS is configured to use 4KB-sized output compression with LZ4. Squashfs is configured to use LZ4 with 4KB, 8KB, 16KB, and 128KB compression chunk sizes, indicated by Squashfs-4K, Squashfs-8K, Squashfs-16K, and Squashfs-128K, respectively. Btrfs is configured to run in read-only mode without data integrity checks for a fair comparison. The compression algorithm used by Btrfs is LZO, since Btrfs does not support LZ4. Both Ext4 and F2FS are used without compression in the experiments since they do not support it.

For real-world applications, we compare EROFS with Ext4, since Ext4 is now the default file system used by Android [17]. We use the commercial version for real-world evaluation since it takes time to ship the latest version to smartphones. We also tried to use Squashfs on Android. However, it costed too much CPU and memory resources, and when trying to run a camera application, the phone froze for tens of seconds before it finally failed.

5.2 Micro-benchmarks

We use FIO to show the basic I/O efficiency of different file systems. In this experiment, we use enwik9 [40] as the workload, which is the first 10^9 bytes of the English Wikipedia dump. We store the file in different file systems and read the file to test the file system read throughput. Each read is a 4KB buffered read. We test the throughput under three scenarios: sequential read, random read, and stride read. For the sequential read, we read the file 4KB by 4KB sequentially; thus the following reads are highly likely to hit in cache since the data is already loaded in the memory by previous decompression or prefetching (i.e., readahead). For the random read, we randomly read the whole file; thus the reads can hit in the cache if the data is already decompressed by previous reads. The last scenario is the stride read, in which we only read the first 4KB in every 128KB data. Since the largest compression chunk is 128KB, stride reads will not hit in cache[4]. We test stride reads to illustrate the worst-case performance for compressed file systems.

Before each test, the page cache is dropped to reduce interference. All tests are done at least ten times, and the average throughputs are reported. The max relative standard deviation is 17.3% for stride reads on A53 cores and 5.1% for the rest results. Fig. 7 shows the following results we observed.

Btrfs performs worst in all tests compared with EROFS and Squashfs-128K, since it is designed neither for compression nor for read-only data. On one hand, Btrfs does not take advantage of the read-only property and has to consider updates; thus it is outperformed by the compressed read-only file systems EROFS and Squashfs-128K. On the other hand, decompression in Btrfs incurs notable performance overhead compared to Ext4 and F2FS which do not need to decompress data during reads. This is reasonable since Btrfs is not designed to be a compressed read-only file system.

Btrfs performs better than other configurations of Squashfs for sequential reads, which is caused by its larger compression chunk (128KB). The advantages shrink in random reads where prefetching does not work; the advantage disappears in stride reads where decompressing more data than requested becomes the burden.

Overall, this result shows the inefficiency of using general file systems with compression support for read-only data and emphasizes the necessity of designing compressed read-only file systems.

As the size of compression input increases, the performance of Squashfs increases for random reads and sequential reads, but decreases for stride reads. The main reason for this phenomenon is the locality and cache. Since file systems have enough memory to cache file data in this experiment, all decompressed data will be cached and possibly be read in the future. Thus for random reads and sequential reads, the larger-sized data is decompressed, more future reads will hit the cache. That is basically the reason why the Squashfs throughputs grow as the compression chunk size increases.

Since both sequential and random reads will read the whole file, there is only a little performance difference, which is caused by the good locality and prefetching.

For stride reads, however, FIO only reads the first 4KB data for each 128KB data, which eradicates the benefits of memory cache since all the data decompressed but not requested will never be used in the future. Thus the more irrelevant data is read and decompressed, the more time and resource are wasted, yielding worse performance. That explains why the throughput drops with the increase of the compression chunk size for Squashfs.

EROFS performs best in most of the tests among file systems with compression support and sometimes outperforms file systems that do not compress data. For sequential reads, EROFS exhibits the best performance among compressed file systems. Most wins come from the design of fixed-sized output compression and the elimination of unnecessary memory allocations and data movements compared with Squashfs. For random reads, EROFS is outperformed by Squashfs-128K since the latter can decompress and cache the whole file during the test, while EROFS only benefits from cached

[4] In enwik9 and silesia.tar, no more than 128KB data is compressed to a single block in EROFS.

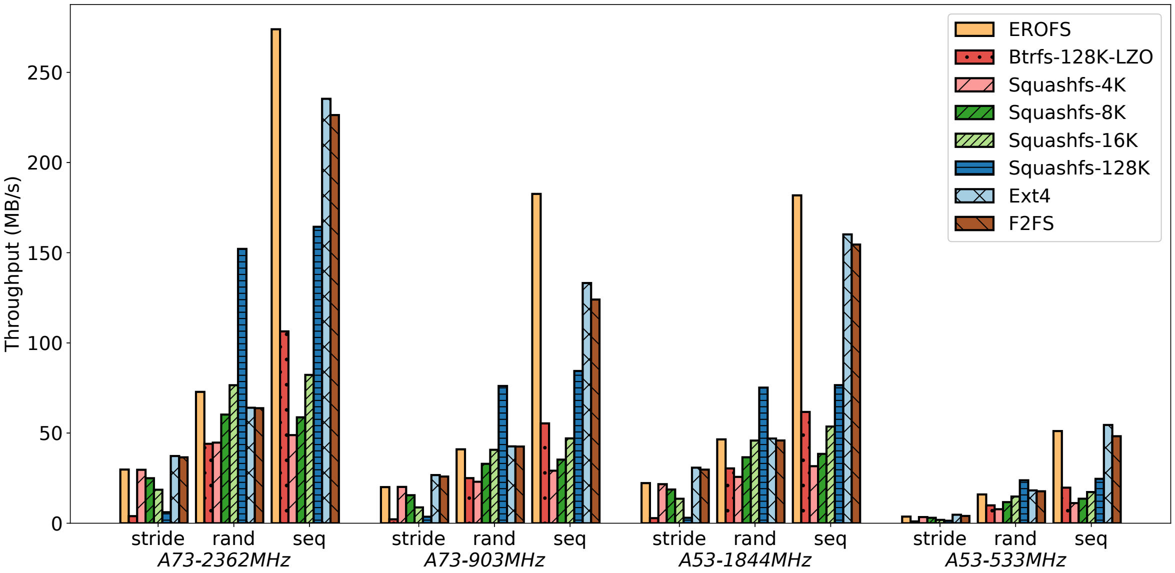


Fig. 7: FIO micro-benchmark results under three read patterns at four CPU frequencies

I/Os. However, EROFS still performs better than other compressed file systems. For stride reads, since the prefetching is barely useful, EROFS still yields the best throughput among compressed file systems, but the win is limited.

Compared with Ext4 and F2FS without compression support, EROFS always performs comparably with and even outperforms them (e.g., the sequential reads on A73 cores). The reason is that even if EROFS needs to decompress data, it reads much less data from the storage thanks to compression.

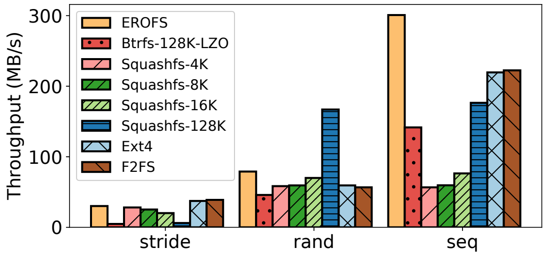
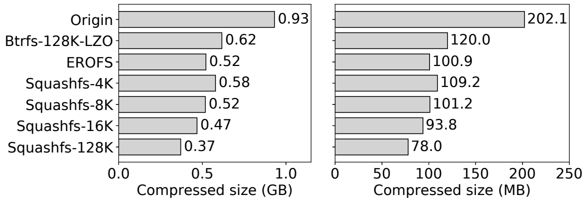


Fig. 8: FIO micro-benchmark results for silesia.tar

We also use the workload, *silesia.tar* [9], to conduct the same experiment. Silesia.tar is a tarball of the silesia compression corpus, which covers typical data types used nowadays. The results show the same trends as enwik9, so we only present the result for A73 cores at 2362MHz in Fig. 8.

5.3 Compression ratio and memory usage

We also evaluated the compression ratio and memory consumption during the decompression of each file system. We use both enwik9 and silesia.tar to present the compression ratio of different file systems. Fig. 9(a) and Fig. 9(b) show the number of bytes used on each file system for enwik9 and silesia.tar. The *origin* line in both figures represents the size of the uncompressed workload file, which is 953.67MB for enwik9 and 202.1MB for silesia.tar. Currently, EROFS only supports 4KB-sized output compression, but compared with Squashfs-4K, the compressed size is 10% and 9% smaller for the two workloads. The figure also matches the facts that the larger the compression unit, the better the compression ratio.



(a) enwik9 **(b) silesia.tar**

Fig. 9: Compressed size

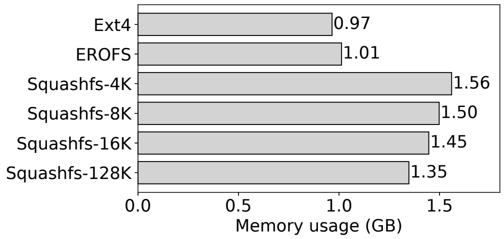


Fig. 10: Decompression memory usage

Fig. 10 shows the memory used after decompressing the enwik9 file. The test is conducted as follows: boot the machine, mount the file system, read the file stored in the file system, check the memory used, and then reboot.

Since the file is roughly 1GB, the remainders are either used by other parts of the operating system or temporarily

Table 1: I/O amount under different read patterns

I/O (MB)	seq-read	rand-read	stride-read
Requested	16.00	16.00	16.00
Squashfs-4K	10.65	26.19	26.23
Squashfs-8K	9.82	33.52	34.08
Squashfs-16K	9.05	46.42	48.32
Squashfs-128K	7.25	165.27	203.91
EROFS	10.14	26.12	25.93

Table 2: I/O patterns

I/O size	=4K	<=8K	<=16K	<128K	=128K	>128K
%	19.0	23.9	30.4	78.9	19.9	1.2

used by the file system. Besides EROFS and Squashfs, we also tested Ext4 as the baseline. From the figure, we can see that compared to Ext4, the memory overhead for four configurations of Squashfs ranges from 39.6% to 61.6%. However, memory used by EROFS is only slightly higher than the Ext4 (about 4.9%). The result shows that EROFS has much lower memory spikes than Squashfs and proves the effectiveness of memory-friendly decompression of EROFS.

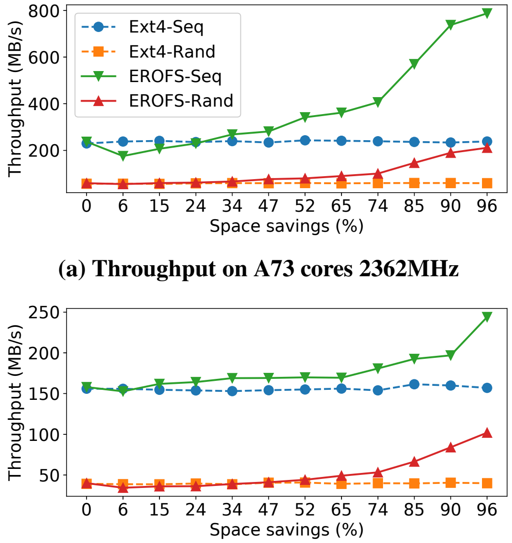
In the test, there is only one file to be decompressed, and we allocate abundant memory to ensure that no memory reclamation or swapping will happen during the decompression. However, in a real-world scenario where more files will be decompressed simultaneously, more memory will be needed by the decompression of Squashfs. Once the available memory is scarce, memory reclamation or swapping may happen, which is very expansive and affects not only the file systems, but also other components or applications in the whole system. Thus in real-world scenarios, the advantages of EROFS, which uses as little as memory during the decompression, will be more remarkable.

5.4 I/O amplification and I/O patterns

We reran tests mentioned in §2.2 on EROFS and different Squashfs configurations. Table 1 lists the actual I/O issued when reading 16MB file data under three read patterns. EROFS issued the least I/O for random reads and stride reads. Yet, since Squashfs-8K, Squashfs-16K and Squashfs-128K have a better compression ratio, they read less data than EROFS for sequential reads. In summary, EROFS reduces the I/O amplification for most cases compared with Squashfs, especially for random reads and stride reads.

We further identified the I/O pattern in a simulated real-world environment to illustrate how I/O amplification will affect real-world applications. We installed 100 apps and ran the Monkey tool [13] to randomly tap the screen once per second for 3 hours. We collected the I/O sizes passed to the `readpage` and `readpages` interfaces and show the proportion of different I/O sizes in Table 2. The result shows that there are quite a lot of I/Os (30.4%) with sizes no more than 16K, which we consider as random I/Os. The amount of random I/Os is reasonable since as the system keeps running for a long time, some pages in the applications' page cache are reclaimed due to memory shortage. The insignificant amount of random I/Os emphasizes the importance of EROFS's effort of reducing the I/O amplification.

5.5 Throughput and space savings



(a) Throughput on A73 cores 2362MHz

(b) Throughput on A73 cores 903MHz

Fig. 11: Throughput under different space savings

Fig. 11 depicts the throughput of EROFS and Ext4 under different space savings. The space savings is the value of space reduced by the compression divided by the size of original data; thus, a larger space savings indicates more space saved. For simplicity, we only show the results for big cores, since similar trends are presented on little cores.

For the test, we collected blocks in our compressed partitions and check their space savings. When we found a block that matches our expected space savings, the original data was decompressed and duplicated multiple times to form a roughly 512MB file. Then we stored the file in EROFS and read it to get the read throughput under its space savings. We also stored the file in Ext4 and got the corresponding read throughput for comparison.

Generally, the throughput of Ext4 remains stable during the test and the performance of EROFS increases together with space savings. EROFS achieves much better throughput than Ext4 when the space savings is high enough. In such cases, a single compressed block can be decompressed to dozens of blocks. Thus, the number of I/O requests is notably reduced, leading to higher performance. While when the space savings is low, the performance of EROFS is similar to Ext4 for random reads and worse than Ext4 for sequential reads. This is the result of the conjunction of I/O costs and decompression computation costs. For random reads, the I/O is more expensive than the decompression computation. While for sequential reads, due to prefetching, I/O is less costly and the computation cost dominates when the space savings is low.

5.6 Different decompression approaches and optimization

To illustrate the effect of different decompression approaches, we ran FIO sequential reads on the Kirin 980 smartphone with A76 cores at 2600MHz. The vmap decompression approach serves file reads at 726.5MB/s while the per-CPU buffer decompression yields throughput of 736.5MB/s. File data is read at 769.7MB/s in the latest EROFS with the rolling decompression and the in-place decompression added.

We also evaluated the effect of scheduling optimization in §4.3 with the same configuration. In the random read workload, the average throughput of EROFS without the scheduling optimization is 64.49MB/s, while with the optimization, the performance improves 9.5% to be 70.61MB/s.

5.7 Real-world applications

For real-world applications, we ran modified Android 9 Pie on both low-end smartphones and high-end smartphones, whose hardware configurations are listed in §5.1. Android system partitions such as */system*, */vendor*, and */odm* are compressed with EROFS, and the space savings ranges from 30% to 35%. We tested the boot time of thirteen popular applications required by the production team. We compared application boot time on EROFS to those on Ext4 and present relative boot time in Table 3. On average, EROFS reduces the boot time by 5.0% for low-end smartphones and 2.3% for high-end smartphones compared with Ext4.

We also conducted the same test while running FIO as the background workload to simulate real-world scenarios. In the FIO workloads, four threads randomly read and write individual files with rate limited at 256KB/s for both reads and writes. The last two rows in Table 3 show the boot time with FIO workloads, where the reduction of boot time is 3.2% and 10.9% for low-end and high-end smartphones, respectively.

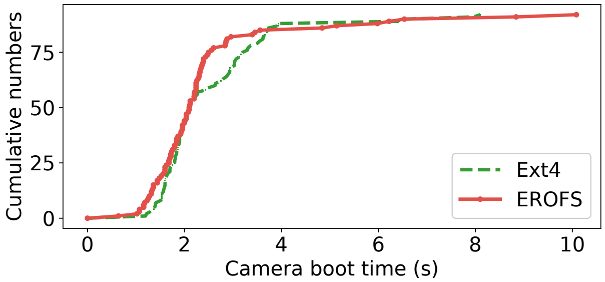


Fig. 12: Camera boot time

Other than the boot time of various applications, we also tested the boot time distribution of the camera application on the aforementioned high-end smartphones. To simulate the situation where memory is scarce, we ran a program in the background which continuously allocates memory and fills in garbage data. We waited until the program consumed all zram in the system before starting the experiment, and kept it running during the evaluation. In the experiment, we booted several applications in turn and recorded the boot time when it's the camera's turn to boot. We collected each time of 92 camera boots for both EROFS and Ext4, and present the cumulative distribution in Fig. 12. The camera application running on EROFS boots faster than on Ext4 for more than 90% of the boots, while the longest boot time on EROFS is worse than that on Ext4. We think the result is acceptable since EROFS saves the storage space while reduces the boot time in most of the cases.

6 Experience over deployment

EROFS has been deployed to tens of millions of smartphones. Here, we report some experiences during the deployment.

Optimizing for all cases, not only common cases. Deploying a new file system to replace an existing one is much harder than we first imagine. The reason is that a commercial product like a smartphone needs to retain the benefit and features of an existing file system. Hence, we need to carefully optimize EROFS for all cases instead of common cases to avoid performance degradation even in some rare cases.

For example, the performance of reading some files on EROFS is slightly worse than on Ext4. To optimize, we leave files with low compression ratio uncompressed for better performance. Further, we collect access frequencies of file blocks from anonymous beta users and store them in dedicated files. According to the frequency information, we pre-decompress the most frequently requested parts of compressed files and pin them in the memory to balance the storage consumption and the performance. As a result, the performance of EROFS can be as good as, and sometimes better than Ext4, while the storage consumption is significantly reduced.

Incomplete implementation leads to performance abnormalities and failures in real-world scenarios. We tested EROFS after the main functionalities have been implemented. However, several kinds of malfunctions happened during real-world tests.

One example is that after the smartphone runs on EROFS for days, several applications become extremely slow at times. Eventually, we found that the root cause is the missing implementation of page migration in EROFS. Page migration is invoked by the memory management subsystem to ask file systems to move their data somewhere else. Most of the time, the page migration will not be triggered and thus leaving it unimplemented is benign. However, when the memory is fragmented, which is the case when the bug happens, the functionality of page migration is crucial to the success of allocating contiguous memory in the system. The issue was solved after we implemented the page migration in EROFS.

Bottlenecks shift on different platforms. We developed

Table 3: Relative boot time of thirteen applications on low-end and high-end smartphones. Each number in the table is the average value of at least five boots. Negative numbers show the boot time reduction (in %) compared with Ext4, while positive numbers indicate the boot time is prolonged (in %). FIO workloads are running in the background for cases with 'w/ FIO' suffix.

App. #	1	2	3	4	5	6	7	8	9	10	11	12	13
Low-end	**-16.4**	-3.5	+4.2	-4.0	-7.5	-1.4	-6.8	+6.3	-2.2	**-18.4**	-3.3	-7.6	-4.5
High-end	-1.8	-0.7	-2.1	-1.8	**-12.3**	-3.7	+1.2	-8.0	-2.8	+0.7	+2.0	-2.7	+1.9
Low-end w/ FIO	-2.8	**-12.9**	-5.4	+3.9	-7.6	+3.7	+4.4	-2.6	+9.9	+4.0	**-11.1**	**-10.3**	**-15.1**
High-end w/ FIO	-4.6	**-14.1**	**-10.7**	**-19.3**	-7.0	**-11.0**	**-15.0**	+0.8	**-22.9**	-5.0	**-18.9**	-0.7	**-13.2**

early versions of EROFS on high-end smartphones where resources are abundant, and tuned it to use as fewer resources as possible. However, when we adopt the tuned EROFS to low-end smartphones, the performance is lower than we expected. This is surprising since we have already considered the limited resources and EROFS should work well. At last, the trouble-maker turned out to be the scheduler. Scheduling on low-end smartphones is much more costly and becomes the bottleneck of EROFS's decompression, which motivated us to introduce the scheduling optimization described in §4.3. Different platforms not only reflect resource limitation directly on the amount of memory available or how fast processors can run, but they can also shift the bottleneck of software.

7 Related Work

Other compressed file systems. Several other file systems support compression. AXFS [24] is a compressed read-only file system. It is designed to enable execute-in-place (XIP), which is not supported by common smartphone storage like eMMC or UFS. CramFS [3] is another compressed read-only file system, which is designed to be simple and space-efficient. However, it also has several limitations such as limited file size. Cramfs was once obsoleted by Squashfs in the Linux kernel [4] and then revived for XIP [5], which is not supported by common smartphone storage like eMMC or UFS. LeCramFS [27] extends CramFS for better read performance and memory efficiency on flash memory. However, the compression ratio is reduced, and LeCramFS generates much larger images.

JFFS2 [15], and UBIFS [12] are two file systems designed for flash memory. Although they support compression, they have to manage the wear-leveling, address translation, and garbage collection for NAND flash. Because all such features are already provided by the eMMC and UFS firmware, EROFS is much simpler and faster than JFFS2 and UBIFS.

Bcachefs [1] is a file system with an emphasis on reliability and robustness. ZFS [16] is a full-fledged file system designed by Sun Microsystems for Solaris. Although they both support compression, they have to consider updates on compressed files, which has a similar issue with Btrfs.

File system and storage for smartphones. File system and storage for smartphones have long been a hot topic. For example, Kim et al. [33] illustrated that storage can affect application performance on smartphones and proposed several approaches to mitigating the performance impact. Jeong et al. [31] uncovered and mitigated the journaling of journal (JOJ) anomaly by overhauling file systems on smartphones, which has generated several follow-up efforts [36, 38, 44]. They further identified Quasi-Asynchronous I/O in smartphone file systems and boosted them for responsiveness [28]. SmartIO [41] reduces the application delay by prioritizing reads over writes. MobiFS [43] is a memory-centric design for smartphone data storage that improves response time and energy consumption.

There has also been much work [29, 30, 34, 35, 37] providing benchmarking frameworks to evaluate file systems and storage on smartphones. Due to the emergence of non-volatile memory (NVM), recent researchers [26, 32, 39, 42, 47] also investigated how NVM can be used on smartphones.

8 Conclusion and Future Work

We introduce EROFS, a new compressed read-only file system designed for smartphones with limited resources. EROFS provides a comparable compression ratio while having much higher performance and less extra memory overhead compared to Squashfs. With fixed-sized output compression and fast and memory-efficient decompression, EROFS can store system code and resources with less storage usage and sometimes even better performance compared with file systems without compression support. Evaluation shows that apps on a system installed on EROFS can boot comparably or even faster compared with on Ext4. EROFS has been merged to the mainline Linux and has been deployed and used in tens of millions of smartphones. Currently, EROFS is still under active development, and we are continuously adding new features, such as deduplication, extended file statistics, fiemap, and EROFS-fuse in future versions of EROFS.

Acknowledgment

We thank our shepherd Ric Wheeler and the anonymous reviewers for the constructive comments, Guifu Li for helping prototype the `mkfs` utility, and Qiuyang Sun for his help with the early testing and evaluation. Haibo Chen is the corresponding author.

References

[1] Bcachefs. https://bcachefs.org.

[2] Btrfs: Main page. https://btrfs.wiki.kernel.org/index.php/Main_Page.

[3] Cramfs - cram a filesystem onto a small ROM. https://www.kernel.org/doc/Documentation/filesystems/cramfs.txt.

[4] Cramfs: mark as obsolete. https://lkml.org/lkml/2013/9/4/79.

[5] Cramfs refresh for embedded usage. https://lkml.org/lkml/2017/8/11/726.

[6] Factory images for Nexus and Pixel devices. https://developers.google.com/android/images.

[7] LZ4 block format description. https://github.com/lz4/lz4/blob/master/doc/lz4_Block_format.md.

[8] Partitions and images. https://source.android.com/devices/bootloader/partitions-images.

[9] Silesia compression corpus. http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia.

[10] SQUASHFS. http://squashfs.sourceforge.net.

[11] SQUASHFS 4.0 filesystem. https://www.kernel.org/doc/Documentation/filesystems/squashfs.txt.

[12] UBIFS file system. https://www.kernel.org/doc/Documentation/filesystems/ubifs.txt.

[13] UI/Application exerciser monkey. https://developer.android.com/studio/test/monkey.

[14] x86_64: expand kernel stack to 16K. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=6538b8ea886e472f4431db8ca1d60478f838d14b.

[15] JFFS2: The journalling flash file system, version 2. http://www.sourceware.org/jffs2/, 2003.

[16] ZFS: the last word in file systems. https://web.archive.org/web/20060428092023/http://www.sun.com/2004-0914/feature/, 2004.

[17] Saving data safely. https://android-developers.googleblog.com/2010/12/saving-data-safely.html, 2010.

[18] Android one was conceived with india in mind, says Google's Sundar Pichai. https://gadgets.ndtv.com/mobiles/news/googles-sundar-pichai-on-android-one-in-an-exclusive-chat-with-ndtvs-vikram-chandra-592062, 2014.

[19] HUAWEI Y3 2018. https://consumer.huawei.com/za/phones/y3-2018/specs/, 2018.

[20] Libext2fs: add EXT2_FLAG_SHARE_DUP to deduplicate data blocks. https://android-review.googlesource.com/c/platform/external/e2fsprogs/+/642333, 2018.

[21] Nokia 2.1 - long lasting entertainment. https://www.nokia.com/phones/en_int/nokia-2, 2018.

[22] Samsung unveils the Galaxy J2 core; an introductory smartphone packed with performance. https://news.samsung.com/global/samsung-unveils-the-galaxy-j2-core-an-introductory-smartphone-packed-with-performance, 2018.

[23] Axboe, J. Flexible I/O tester. https://github.com/axboe/fio.

[24] Benavides, T., Treon, J., Hulbert, J., and Chang, W. The enabling of an Execute-In-Place architecture to reduce the embedded system memory footprint and boot time. *JCP 3*, 1 (2008), 79–89.

[25] Brumley, J. Apple, Samsung continue to lose smartphone market share in shift toward more value. https://seekingalpha.com/article/4101007-apple-samsung-continue-lose-smartphone-market-share-shift-toward-value, 2017.

[26] Chen, R., Wang, Y., Hu, J., Liu, D., Shao, Z., and Guan, Y. Unified non-volatile memory and NAND flash memory architecture in smartphones. In *Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific* (2015), IEEE, pp. 340–345.

[27] Hyun, S., Bahn, H., and Koh, K. Lecramfs: an efficient compressed file system for flash-based portable consumer devices. *IEEE Transactions on Consumer Electronics 53* (2007).

[28] Jeong, D., Lee, Y., and Kim, J.-S. Boosting quasi-asynchronous I/O for better responsiveness in mobile devices. In *FAST* (2015), pp. 191–202.

[29] Jeong, S., Lee, K., Hwang, J., Lee, S., and Won, Y. AndroStep: Android storage performance analysis tool. In *Software Engineering (Workshops)* (2013), vol. 13, pp. 327–340.

[30] Jeong, S., Lee, K., Hwang, J., Lee, S., and Won, Y. Framework for analyzing android I/O stack behavior: from generating the workload to analyzing the trace. *Future Internet 5*, 4 (2013), 591–610.

[31] JEONG, S., LEE, K., LEE, S., SON, S., AND WON, Y. I/O stack optimization for smartphones. In *USENIX Annual Technical Conference* (2013), pp. 309–320.

[32] KANG, D. H., AND EOM, Y. I. FSLRU: a page cache algorithm for mobile devices with hybrid memory architecture. *IEEE Transactions on Consumer Electronics 62*, 2 (2016), 136–143.

[33] KIM, H., AGRAWAL, N., AND UNGUREANU, C. Revisiting storage for smartphones. *ACM Transactions on Storage (TOS) 8*, 4 (2012), 14.

[34] KIM, H., RYU, M., AND RAMACHANDRAN, U. What is a good buffer cache replacement scheme for mobile flash storage? In *ACM SIGMETRICS Performance Evaluation Review* (2012), vol. 40, ACM, pp. 235–246.

[35] KIM, J.-M., AND KIM, J.-S. Androbench: benchmarking the storage performance of android-based mobile devices. In *Frontiers in Computer Education*. Springer, 2012, pp. 667–674.

[36] KIM, W.-H., NAM, B., PARK, D., AND WON, Y. Resolving journaling of journal anomaly in android I/O: multi-version B-tree with lazy split.

[37] LEE, K., AND WON, Y. Smart layers and dumb result: IO characterization of an android-based smartphone. In *Proceedings of the tenth ACM international conference on Embedded software* (2012), ACM, pp. 23–32.

[38] LEE, W., LEE, K., SON, H., KIM, W.-H., NAM, B., AND WON, Y. WALDIO: eliminating the filesystem journaling in resolving the journaling of journal anomaly. Usenix.

[39] LUO, H., TIAN, L., AND JIANG, H. qNVRAM: quasi non-volatile RAM for low overhead persistency enforcement in smartphones. In *HotStorage* (2014).

[40] MAHONEY, M. About the test data. http://mattmahoney.net/dc/textdata.html, 2011.

[41] NGUYEN, D. T., ZHOU, G., XING, G., QI, X., HAO, Z., PENG, G., AND YANG, Q. Reducing smartphone application delay through read/write isolation. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services* (2015), ACM, pp. 287–300.

[42] PARK, H., BAEK, S., CHOI, J., LEE, D., AND NOH, S. H. Exploiting storage class memory to reduce energy consumption in mobile multimedia devices. In *Consumer Electronics (ICCE), 2010 Digest of Technical Papers International Conference on* (2010), IEEE, pp. 101–102.

[43] REN, J., LIANG, M. C.-J., WU, Y., AND MOSCIBRODA, T. Memory-centric data storage for mobile systems.

[44] SHEN, K., PARK, S., AND ZHU, M. Journaling of journal is (almost) free.

[45] TOLOMEI, S. Shrinking APKs, growing installs. https://medium.com/googleplaydev/shrinking-apks-growing-installs-5d3fcba23ce2.

[46] ZHANG, X., LI, J., WANG, H., XIONG, D., QU, J., SHIN, H., KIM, J. P., AND ZHANG, T. Realizing transparent OS/Apps compression in mobile devices at zero latency overhead. *IEEE Transactions on Computers 66*, 7 (2017), 1188–1199.

[47] ZHONG, K., WANG, T., ZHU, X., LONG, L., LIU, D., LIU, W., SHAO, Z., AND SHA, E. H.-M. Building high-performance smartphones via non-volatile memory: The swap approach. In *Proceedings of the 14th international conference on embedded software* (2014), ACM, p. 30.