Tuning the Ext4 Filesystem Performance for Android-Based Smartphones

Hyeong-Jun Kim and Jin-Soo Kim

School of Information and Communication Engineering
Sungkyunkwan University (SKKU), Suwon 440-746, Korea
{kimhj0514,jinsookim}@skku.edu

Abstract. The storage performance plays an important role in today's smartphones. However, most file systems have been optimized for hard disk drives and general filesystem workloads. This paper aims at improving the performance of the Ext4 filesystem, a de facto filesystem in Android-based smartphones, by taking into account the filesystem workloads in the Android platform and the characteristics of the underlying NAND flash-based storage device. We have considered five tuning parameters of the Ext4 filesystem. Our evaluation on a real Android-based smartphone shows that the new Ext4 filesystem, where all the tuning parameters are applied, improves the Postmark performance by up to 13% compared to the original Ext4 filesystem.

Keywords: Ext4, Filesystem performance, Android, NAND flash-based storage.

1 Introduction

Recently, the demand for Android-based smartphones is increasing rapidly. The market share of Android-based smartphones is reported to be 43.4% in the second quarter of 2011, up from just 17.2% in the same quarter of the previous year [1]. Among all smartphone operating systems, Android is the most fast-growing OS in terms of the market share during the past year. With this growth, the features and capabilities offered by Android-based smartphones are getting more and more sophisticated.

The storage performance plays an important role in ensuring better user experience in today's smartphones. This is because the applications running on smartphones are getting complex and storage can be a performance bottleneck in these applications, as in conventional desktop or notebook computers. For example, the Internet browser needs to store a large number of small product images while browsing e-commerce sites. High-end game software available in smartphones has a visible delay to read in more than tens of megabytes of game data when it is started. Therefore, maximizing the storage performance is one of the most challenging tasks facing system designers.

In this paper, we primarily focus on the filesystem performance of Android-based smartphones. More specifically, we strive to improve the performance of the Ext4 filesystem which serves as a *de facto* filesystem in most of Android-based smartphones. Tuning the Ext4 filesystem performance has been approached in two ways. First, we tune the Ext4 filesystem by analyzing the filesystem workload characteristics of Android-based smartphones. According to our analysis, we find that about 50% of the files are less than or equal to 4KB in size. This suggests that it is important to minimize the metadata overhead caused by reading and writing small files. Second, the Ext4 filesystem has been extensively optimized for hard disk drives (HDDs), but most of smartphones use NAND flash-based storage such as eMMCs (Embedded MultiMediaCards) [2]. Since eMMCs exhibit performance characteristics quite different from HDDs, we need to revisit policies and mechanisms of the Ext4 filesystem for eMMCs. Our evaluation on a real Android-based smartphones shows that the carefully tuned Ext4 filesystem improves the read and write throughput of the Postmark benchmark by 13% and 11%, respectively, compared to the performance with default options.

The rest of the paper is organized as follows. The next section discusses the related work. Section 3 briefly reviews some of features of the Ext4 filesystem. Section 4 presents the performance evaluation results obtained on a real Android-based smartphone for each tuning parameter. Finally, section 5 concludes the paper.

2 Background

Early Android-based smartphones used YAFFS2 [3] as the default filesystem. However, since the official release of the Android 2.3 platform codenamed Gingerbread, the default filesystem has been changed to Ext4. One of the main reasons is that YAFFS2 has a single-threaded design which might be a bottleneck in the upcoming multi-core environment [4].

The Ext4 filesystem has been designed and optimized mainly for HDDs. The seek time and the rotational delay caused by mechanical arms and rotating platters make the random I/O performance of HDDs fall far short of their sequential I/O performance. Many filesystem policies and mechanisms of Ext4 have been devised to overcome these performance characteristics of HDDs. One notable example is the *block group*. The Ext4 filesystem divides the entire disk space into a number of block groups, and tries to allocate data blocks belonging to a file into the same block group. This is an effort to minimize the head movement when reading the file, by placing the related data at physically close locations.

On the other hand, NAND flash memory used as a storage medium in most smartphones has different characteristics. First, there is no seek time since NAND flash memory is a solid state storage device. The read latency is independent of the location of the data in a single NAND flash memory chip. Second, NAND flash memory has asymmetric operational latencies; the write latency is greater than the read latency by more than four times. Third, the previously written data should be erased to overwrite a new data in the same location. To make matters worse, the larger area (called an *erase block*) containing the previous data needs to be erased at once by a single erase operation.

Due to the presence of erase operations, the traditional disk-based file system cannot be used directly on top of NAND flash memory. Instead, NAND flash-based storage solutions such as SSDs (Solid State Drives) and eMMCs have internal firmware called FTL (Flash Translation Layer) [5], which hides the peculiarities of NAND flash memory and emulates the block device interface on NAND flash memory. With the help of FTL, unmodified disk-based file systems such as Ext4 can be used for eMMCs as well.

3 Ext4 File System

Ext4 is one of the most widely used journaling filesystems in Linux, developed as a successor to Ext3. The Ext4 filesystem splits the disk space into logical blocks to reduce management overhead and to increase throughput. The typical block size is 4KB. These blocks are grouped together to form a block group. Each block group consists of block bitmap, inode bitmap, inode table, and data blocks with optional backup of superblock and block group descriptor table. To minimize external fragmentation and disk seek time, the block allocator tries to allocate new blocks in the same block group for a new file [7, 8]. The key features of the Ext4 filesystem can be summarized as follows.

Extents. Extents are a new block mapping scheme introduced in Ext4. Instead of indirect pointers used in Ext2/3, a single extent identifies a set of blocks which are logically contiguous within the file and also on the underlying block device. Up to 4 extents are stored in the inode directly, while Htree is used when the number of extents exceeds four. Extents are useful to reduce the amount of block allocation information especially for large files.

Delayed Allocation. Ext4 uses a new block allocation mechanism called delayed allocation. Instead of allocating a new block during the write() operation, Ext4 postpones the allocation of a new block until it is flushed from the page cache. This is effective in reducing internal file fragmentation.

Multiblock Allocator. With delayed allocation, Ext4 can allocate a group of relevant blocks in a batch while Ext2/3 allocates one block at a time. The multiblock allocator minimizes the filesystem overhead, making better choice for allocating blocks.

Flex Block Group. The flex block group allows for more flexible placement of filesystem metadata. A number of block groups (16, by default) are combined into a flex block group, where block bitmap, inode bitmap, and inode tables for each block group are all allocated at the beginning of the corresponding flex block group. This increases the metadata locality and enables fast loading of metadata.

Large Inode. In Ext2, the default inode size was 128 bytes. In Ext4, the default inode size is increased to 256 bytes to accommodate several new fields for supporting nano second timestamps, inode versioning, and extended attributes. Extended attributes are used to associate special metadata (such as access control lists, etc.) with a file.

Journal Checksum. To improve reliability, Ext4 computes a CRC32 over all of the journal transaction and adds it to the journal commit block. Without the journal checksum, the recovery program has to detect the corruption in the journal by matching the transaction numbers of the header and the commit block. Moreover, the commit block should be written to disk after the header and all the metadata blocks are written to disk to ensure reliability. Using the journal checksum, corrupted transactions can be detected more easily and the commit block can be written before other metadata blocks.

4 Tuning the Ext4 Filesystem

4.1 Evaluation Methodology

Our evaluations have been performed on a commercial Android-based smartphone, Samsung GT-I9000, running the Linux kernel version 2.6.32.9. GT-I9000 is equipped with 8GB of internal eMMC storage (Sandisk iNand), which is divided into 2GB of /data partition and another 6GB of /mnt/sdcard partition. Since the /data partition is essential in operating the smartphone, we used the /mnt/sdcard partition for our evaluations after reducing its size to 2GB. Whenever we create a new Ext4 filesystem with different parameters, we aged the filesystem to model a realistic environment. Aging has been performed by repeating the creation and deletion of 256KB-sized files randomly until the half of the filesystem is filled with those files.

To investigate the different aspects of filesystem performance, we chose three benchmarks: IOzone [9], Flexible Filesystem Benchmark (FFSB) [10], and Postmark [11]. IOzone is configured to perform several microbenchmarks (sequential read, sequential write, random read, and random write) on a 32MB file with the 256KB record size for sequential tests and with the 4KB record size for random tests. FFSB is used to model the filesystem workload issued by multiple threads. We create 8 threads for each benchmark test. For large file read and write tests, each thread reads or writes 25MB files with the 256KB record size. In small file read and write tests, each thread reads or writes 40KB files with the 4KB record size. The performance metric we use is the amount of data read or written per second by each thread. Postmark performs 10000 transactions on 5000 files using a single thread. Each transaction is one of create, read, append, and delete operations and the file size is set to 4KB. These configurations are based on our analysis of five real Android-based smartphones; we find that about 50% of files in the /data partition are less than or equal to 4KB in size. In all benchmarks, we have cleared the buffer cache before each run.

4.2. The Effect of Each Tuning Parameter

Ext4 provides a number of tuning parameters. In this subsection, we study the effect of these parameters on the performance by changing only one or two parameters at a time. All the benchmark results are normalized to the performance of Ext4 with default options.

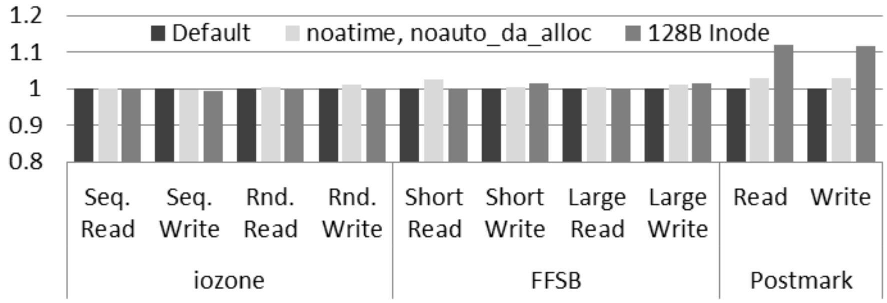

Fig. 1. The normalized performance when we enable noatime/noauto_da_alloc options and the inode size is reduced to 128 bytes

Noatime, noauto_da_alloc. The first tuning parameters we consider are noatime and noauto_da_alloc. When a file is read, Ext4 updates the last access time in the corresponding inode. The option noatime makes the Ext4 filesystem stop recording the last file access time, which eliminates inode updates. This option is useful especially when a number of small files are read. Ext4 tries to avoid the creation of zero-length files due to delayed allocation when a system crashes in a certain situation. The noauto_da_alloc disables this feature. In fact, the generation of zero-length files on a sudden power failure is POSIX-compliant and the problem should be avoided not by the file system, but by the application by calling fsync() before closing a file. Therefore, it is safe to use the noauto_da_alloc option as long as the application is correctly written. With noauto_da_alloc, Ext4 does not have to detect the problematic situation and utilizes delayed allocation more effectively.

Figure 1 depicts the normalized performance of Ext4 when noatime and noauto_da_alloc are enabled. The improved read performance in FFSB and Postmark is due to the noatime option. We can see that the use of noauto_da_alloc increases the write performance in IOzone, FFSB, and Postmark.

128-Byte Inode Size. The default inode size of Ext4 is increased from 128 bytes to 256 bytes. However, new fields added into the 256-byte inode, such as nano second timestamps and extended attributes, are not used in Android-based smartphones. If we decrease the inode size back to 128 bytes, the number of inodes stored in a single inode page is doubled. This is found very effective in reducing the metadata traffic, especially when a large number of files are accessed. As shown in Fig. 1, the performance of Postmark and FFSB has been improved by about 8% and by about 3%, respectively, with this change. The performance of IOzone has not been affected because IOzone deals with only a single file.

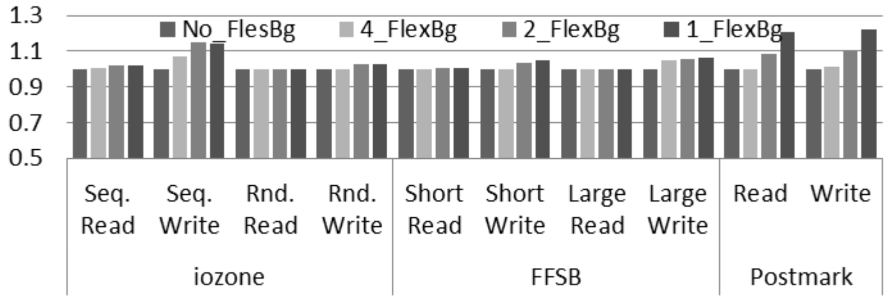

Fig. 2. The changes in the performance when we vary the number of flex block groups to 32, 8, 4, and 1 with 32 block groups. The results are normalized to the case of No_FlexBg.

The Flex Block Group Size. In Ext4, the default flex block group size is 16, i.e., all the filesystem metadata of 16 block groups are placed at the beginning of the associated flex block group. To investigate the impact of the flex block group size on the Ext4 performance, we create a new 4GB filesystem in the /mnt/sdcard partition. Fig. 2 compares the performance when we change the number of flex block groups to 32 (No_FlexBg), 4 (4_FlexBg), 2 (2_FlexBg), and 1 (1_FlexBg) in the same 4GB partition. Since the each block group size is 128KB with the 4KB block size, the partition has 32 block groups. Therefore, there are 1, 8, 16, and 32 block groups per flex block group for No_FlexBg, 4_FlexBg, 2_FlexBg, and 1_FlexBg, respectively. Having only one block group for each flex block group (No_FlexBg) is essentially equivalent to not using the flex block group feature. 2_FlexBg represents the default Ext4 configuration. In 1_FlexBg, we combine all 32 block groups in one flex block group.

As can be seen in Fig. 2, the overall performance has been improved by up to 20% (in Postmark), when we decrease the number of flex block groups. Usually, filesystem metadata, such as block bitmaps, inode bitmaps, and inode tables, are considered hot as they are frequently updated, while file data are relatively considered cold. In the NAND flash-based storage, separating hot data from cold data is known to be very important to increase the efficiency of flash memory management [12]. We believe placing the entire filesystem metadata at the very beginning of the partition as is done in 1_FlexBg, is desirable to the underlying NAND flash-based storage.

Journal Asynchronous Commit. The default behavior of Ext4 is to perform synchronous commit; the commit block can be written to disk after ensuring that the header and all metadata blocks are written to disk. However, with the journal checksum, Ext4 no longer has to wait until the header and metadata blocks are written to disk to write the commit block. This feature is called journal asynchronous commit.

The use of journal asynchronous commit can increase the bandwidth as it removes a barrier needed before writing the commit block in the synchronous commit scheme. Fig. 3 depicts the changes in the performance when we enable journal asynchronous commit. As expected, the write performance is improved by up to 7%.

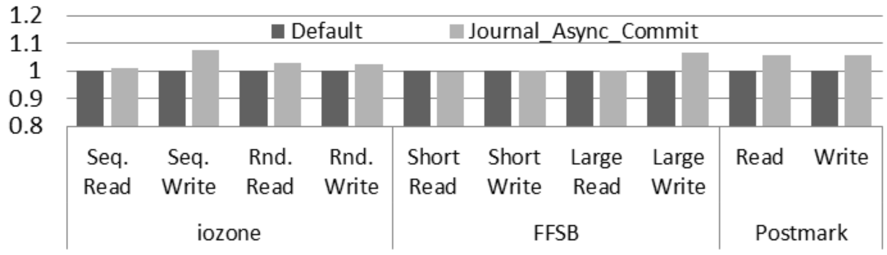

Fig. 3. The effect of enabling journal asynchronous commit

4.3 The Performance of the Tuned Ext4 Filesystem

Fig. 4 summarizes the tuning parameters investigated in this paper. Among them, noatime, noauto_da_alloc, and journal_async_commit are mount options, while the number of flex block groups and the inode size are parameters used during filesystem creation with mkfs. The use of noatime, single flex block group, and the smaller inode size contributes to reducing the metadata overhead in NAND flash-based storage. The journal asynchronous commit and the noauto_da_alloc options enhance the efficiency of filesystem operations.

Fig. 5 depicts the performance of the tuned Ext4 filesystem where all the aforementioned features are applied. We can see that the proposed tuning method has benefit for all benchmarks we have tested. In particular, the performance of Postmark has been improved by 13% and by 11% for reads and writes, respectively, compared to the Ext4 performance with default options.

mkfs options	
Single Flex BG	-G 32
128B inode	-I 128
mount options	
noatime	-o noatime
noauto_da_alloc	-o noauto_da_alloc
Journal_async_commit	-o journal_async_commit

Fig. 4. Tuning parameters of Ext4 considered in this paper

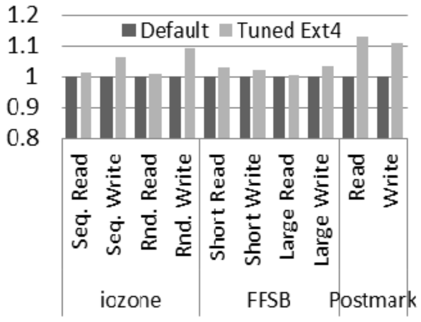

Fig. 5. The normalized performance of the tuned Ext4 filesystem where all the proposed features are applied

5 Conclusion

This paper aims at improving the performance of the Ext4 file system for Android-based smartphones. In tuning the Ext4 filesystem, it is important to understand the filesystem workload characteristics of Android-based smartphones. We have conducted an analysis of file systems used in five real Android-based smartphones and found that about half of the files require only one 4KB data block. This suggests that it is necessary to reduce the metadata overhead while reading and writing small files. In addition, we take into account of the characteristics of the underlying NAND flash-based storage device, namely eMMCs.

In this paper, we have considered five tuning parameters of the Ext4 filesystem: noatime, noauto_da_alloc, journal_async_commit, single flex block group, and the smaller inode size. These parameters are found to be very effective in decreasing the metadata overhead and enhancing the efficiency of the Ext4 filesystem operations. Our evaluation on a real Android-based smartphone shows that the proposed tuning method improves the performance of Postmark by up to 13% compared to the default Ext4 filesystem. We plan to extend our study to other storage software stacks such as I/O schedulers and virtual memory to further optimize the storage performance in Android-based smartphones.

Acknowledgements. This work was supported by Mid-career Researcher Program (No. 2010-0026511) and by Next-Generation Information Computing Development Program (No. 2010-0020730) through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science, and Technology.

References

1. Gartner, Inc., http://www.gartner.com/it/page.jsp?id=1764714
2. eMMC (Embedded MultiMediaCard), http://en.wikipedia.org/wiki/MultiMediaCard
3. YAFFS (Yet Another Flash File System), http://www.yaffs.net
4. Tso, T.: Android will be using ext4 starting with Gingerbread. The Linux Foundation, http://www.linuxfoundation.org/news-media/blogs/browse/2010/12/android-will-be-using-ext4-starting-gingerbread
5. Intel Corporation, Understanding the flash translation layer (FTL) specification, Application Note AP-684 (1998)
6. Btrfs, http://btrfs.wiki.kernel.org
7. Card, R., Ts'o, T., Tweedie, S.: Design and Implementation of the Second Extended Filesystem. In: First Dutch International Symposium on Linux (1994)
8. Mathur, A., Cao, M., Bhattacharya, S.: The new ext4 filesystem: current status and future plans. In: Ottawa Linux Symposium (2007)
9. IOZone, http://www.iozone.org
10. FFSB project, http://sourceforge.net/project/ffsb
11. Katcher, J.: Postmark a new filesystem benchmark, Network Appliances (2002)
12. Chaing, M.-L., Lee, P.C.H., Chang, R.-C.: Using data clustering to improve cleaning performance for flash memory. Software: Practice and Experience 29(3), 267–290 (1999)