Per-Block-Group Journaling for Improving Fsync Response Time

Yunji Kang and Dongkun Shin
College of Information and Communication Engineering
Sungkyunkwan University
Suwon, Korea
oso41@skku.edu, dongkun@skku.edu

Abstract—**The journaling is indispensable to the file systems of battery-backed consumer devices. It can cope with sudden power failures guaranteeing the file system consistency. However, the current compound transaction scheme can incur significantly long latencies for fsync system calls. This paper proposes a new more fine-grained journaling scheme called per-block-group journaling. Experiments show that the proposed scheme can reduce the average response time of fsync by up to 75%.**

Keywords—journaling; fsync; ext4; file system; storage

I. INTRODUCTION

Recent consumer devices such as smartphones, tablet PCs, and smart TVs have a large amount of NAND flash memory storage to store various multimedia data and applications. The storage device is accessed via a versatile file system such as ext4 which is a journaling file system supporting fast file system recovery [1]. The ext4 journal log is stored on a journal area reserved at storage. Ext4 groups many pending file system updates into a single *compound transaction* that is periodically committed by journaling thread to the journal area, instead of using each file system update as a separate transaction. Since the same structure is frequently updated in a short period of time, the compound transactions may have better performance than more fine-grained transactions [2]. The compound transaction is maintained in the transaction buffer of main memory until it is committed to the journal area.

By forcing journal updates before updating the original file system, the journaling can redo or undo any incomplete committed operations. The ext4 journaling supports three journaling modes: writeback mode, ordered mode, and data journaling mode. The ordered mode, which is the default option, journals only metadata. However, the ordered mode has an ordering constraint to guarantee file system consistency, where data writes to their original locations should be completed before the journal writes of the metadata. Therefore, the journal write latency will be long if the size of associated data is large.

The long latency of journal writes may not be a problem when the journal writes are invoked by a background journaling thread. However, it can be critical for user's *fsync* system call. The fsync system call for a file should commit all the file system updates relating to the file to a persistent storage device. Under the ordered mode journaling of ext4, the fsync system call wakes the journaling thread up on demand. Then, the journaling thread writes the compound transaction at the journal area. The compound transaction may include the metadata updates of other irrelevant files as well as the target file of the fsync (*fsynced* file). The fsync operations are frequently called when databases or xml configuration files are updated and the relevant data of fsync are generally small. However, due to the ordering constraint in journaling, the fsync operation should wait the flush completion of all the dirty pages related to the compound transaction. For example, consider the case when a background application of smartphone is coping a large multimedia file from the USB-connected host PC to its local storage. Then, the journal transaction buffer will have the metadata updates by the copy operation and there are many not-yet-flushed data in the page cache. At the moment, if another application updates a small file unrelated with the copy operation, and sends an fsync request, the response time will be long significantly.

The long latency problem of fsync results from the compound transaction that includes the metadata updates of irrelevant file operations. To solve the problem, we propose a *per-block-group (PBG) journaling* that extracts a block-group-level transaction from the compound transaction, and commits only the target block group's transaction in order to reduce the fsync latency.

II. PER-BLOCK-GROUP JOURNALING

Ext4 file system splits the storage space into a number of *block groups*. Each block group has its own block bitmap, inode bitmap, inode table, and data blocks. Only the group descriptor table (GDT) is shared among multiple block groups. The ext4 block allocator tries to allocate an inode in the same block group as the parent directory, and allocate data blocks in the same block group as the inode. If that group has no free inode or block, other block groups can be used. Therefore, most of the file operations modify the metadata of only one block group. If an inode is not located in the same block as its parent directory or its data blocks, there are metadata updates at multiple block groups.

Fig. 1 demonstrates the journaling operations of ext4. The transaction buffer in the main memory has the several metadata updates that are not committed. In this example, four inodes are updated and the corresponding six metadata blocks are included at the transaction list. Two block groups, BG 2 and BG 3, have their updated entries in the GDT block. While the data sizes of inode 1 and inode 2 are large, the data sizes of inode 3 and inode 4 are small. In the original ext4 journaling, when an application calls the fsync, all the metadata in the transaction buffer should be written at the journal area. In the ordered mode journaling, the metadata writes can be started after all the relevant dirty pages (D1, D2, D3, and D4) are written at the data area. Therefore, the fsync latency will be long.

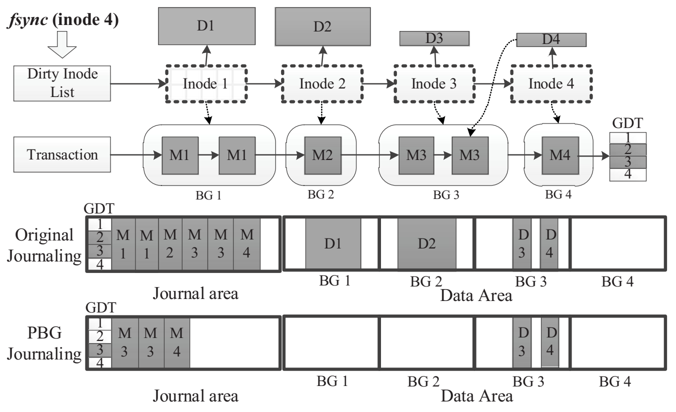

Fig. 1. Comparison between the original and PBG journalings.

The proposed PBG journaling commits only the transactions relevant to the fsynced file. It extracts only the relevant metadata updates from the compound transaction in the transaction buffer. Since each block group has its own metadata not shared by other block groups (except GDT), the target metadata can be easily separated from the compound transaction. For example, when the journal transaction buffer has the metadata updates of inode 1, inode 2, and inode 3, if user modifies the file of inode 4, the transaction buffer has four block groups' metadata as shown in Fig. 1 as long as the journaling thread is not waken up. The data blocks for D4 are allocated at BG 3, and thus the block bitmap metadata of BG 3 (M3) is also updated. If an fsync is called for the file, the journaling thread is invoked on demand, and the PBG journaling is started instead of the normal journaling. The journaling thread first identifies the block groups relevant to the fsynced file. Since both M3 and M4 should be written to the journal area in order to commit the file system changes by the fsync, the relevant block groups are BG 3 and BG 4. Before the journal commit, the PBG journaling flushes all the dirty pages of inode 3 and inode 4. Compared to the original journaling, the PBG journaling can reduce the number of flushed dirty pages significantly as well as reduce the writes on the journal area. After the PBG journaling, only the committed metadata updates are removed from the transaction buffer, and the remaining transactions will be handled by the periodic journaling thread.

One hurdle in implementing the PBG journaling is the GDT that is shared by all block groups. In the example of Fig. 1, the compound transaction has an uncommitted GDT block, which has the modified entries for BG 2 and BG 3. Since the PBG journaling should commit only the relevant metadata, it should generate a partially updated GDT block by eliminating the irrelevant entries, and write the GDT block in the journal area as shown in Fig. 1. To generate the partially updated GDT block, the original GDT block committed by the previous journaling should be maintained in the main memory. By updating only the target entry from the original GDT block, the partially updated GDT block can be generated. Since the total number of GDT blocks is small, particularly in embedded systems, the memory overhead is negligible. There is no change by the PBG journaling in the recovery scheme.

III. Experiments

We evaluated the proposed technique with an Android-based smartphone device equipped with 32-GB eMMC storage. While an application writes a 2-GB file at the directory of /data/media, our test program writes 1-KB data to a file in another directory and generates an fsync after the modification at every 0.5 second. Fig. 2 compares the average response times of fsync calls under different journaling schemes. Two different sizes of flex groups are examined. The PBG journaling reduced the average response time of fsync by 75% and 52% when the flex group sizes are 1 and 16, respectively. Fig. 3 shows the response time variations of fsync calls. There are large response time fluctuations in the original journaling due to the undetermined amount of flushed dirty pages. The PBG journaling shows generally short latencies except only a few cases. If the fsync is called just when the normal periodic journaling has been already started, the fsync should follow the normal journaling policy. Optimizing the case is our future work.

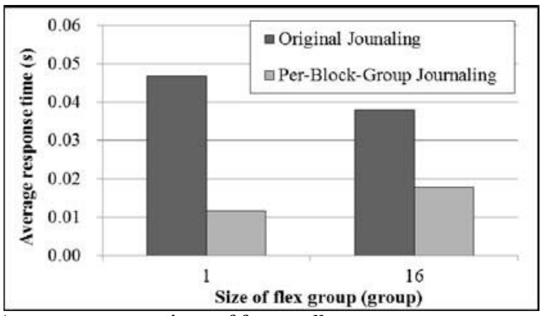

Fig. 2. Average response times of fsync calls.

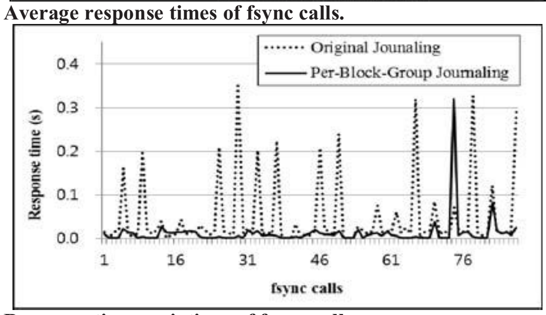

Fig. 3. Response time variations of fsync calls.

Acknowledgements

This work was supported by the Center for Integrated Smart Sensors funded by the Ministry of Science, ICT & Future Planning as Global Frontier Project (CISS-2011-0031863).

References

[1] Tweedie, S. Ext3, journaling filesystem. In Proceedings of the Ottowa Linux Symposium, 2000.

[2] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In Proc. of the 2005 USENIX, pages 105-120, 2005.