Amortized Linux Ext3 File System with Fast Writing after Editing for WinXP-Based Multimedia Application*

Seung-Wan JUNG†(a), Member, Young Jin NAM††, and Dae-Wha SEO†, Nonmembers

SUMMARY Recently, the need for multimedia devices, such as mobile phones, digital TV, PMP, digital camcorders, digital cameras has increased. These devices provide various services for multimedia file manipulation, allowing multimedia contents playback, multimedia file editing, etc. Additionally, digital TV provides a recorded multimedia file copy to a portable USB disk. However, Linux Ext3 file system, as employed by these devices, has a lot of drawbacks, as it required a considerable amount of time and disk I/Os to store large-size edited multimedia files, and it is hard to access for typical PC users. Therefore, in this paper a design and implementation of an amortized Ext3 with FWAE (Fast Writing-After-Editing) for WinXP-based multimedia applications is described. The FWAE is a fast and efficient multimedia file editing/storing technique for the Ext3 that exploits inode block pointer re-setting and shared data blocks by simply modifying metadata information. Individual experiments in this research show that the amortized Ext3 with FWAE for WinXP not only dramatically improves written performance of the Ext3 by 16 times on average with various types of edited multimedia files but also notably reduces the amount of consumed disk space through data block sharing. Also, it provides ease and comfort to use for typical PC users unfamiliar with Linux OS.

key words: Ext3 file system, multimedia application, data sharing, multimedia editing

1. Introduction

Recently, demand for high-performance multimedia devices has been ever-growing, as the quality of multimedia contents has continuously increased. For example, recoding a multimedia content of higher quality consumes more disk capacity, implying a bigger file size [1]. Recently, digital TV and digital camcorders have begun to provide high-definition (HD) and Full HD images in real time. Normal digital TV sets provide 720 x 480 resolutions, whereas the full HD digital TV sets support 1920 x 1080 resolutions. Simple calculation reveals that a full HD multimedia content consumes 9 Gbytes of storage capacity every hour [2], [3], when a multimedia device records the content.

Typical multimedia devices are equipped with a microcontroller including a high-performance decoding engine, high-quality multimedia input/output devices, and data storage devices. With the persistent advance in hard disk technologies, the hard disk has currently become more commonplace, as compared with the flash memory, because of its cost-effectiveness; that is, the hard disk-based storage devices are being widely used for various multimedia devices from PVR to digital camcorders, portable media players, and digital TV sets [4], [5].

Multimedia devices, such as digital TV sets and PVR [6] devices employ the Linux Ext3 file system [7]–[9] on top of the hard disk-based storage devices. Note that the Ext3 has been a default file system in the Linux kernel because of its useful features including journaling for fast recovery, large-scale file size support, etc. In addition, the Linux Ext3 is under GPL license, implying that all the underlying source codes can be used, freely modified, and redistributed by anyone [10].

Recently, digital TV sets have provided a file copy function that recorded a multimedia file copy to a portable storage device through USB interface, in order to allow users to play multimedia files in other multimedia devices such as the PC and other kinds of digital TV. However, it has a drawback that users do not watch multimedia files copied from digital TV in the PC. Because the OS is different between the PC and the digital TV, also the file system is different between the PC and the portable storage device. Note that the portable storage device should be formatted as an Ext3 file system to be able to copy multimedia files from digital TV because the most of digital TV sets employ the Linux Ext3 file system [11]. Therefore, a PC-employing Windows OS is not able to read the portable storage device. It is read in a PC-employing Linux OS, but typical PC users are not familiar with Linux OS. To manipulate the storage device-copied multimedia file from digital TV in a Windows OS familiar with typical PC users, a specific type of software is needed.

Sometimes, users of multimedia devices would like to edit recorded multimedia contents stored in their storage devices, in order to reserve more storage space or to meet certain types of their individual preferences. Note that, in this paper the editing means only deleting the portion of multimedia contents. For example, the editing function provides users to edit recorded streams to remove advertisement and uninterested portions. A typical sequence of editing multimedia contents includes the following three steps: 1) marking a set of removing parts, 2) deciding whether to store the edited contents as a new file or to overwrite them onto the original file, and 3) storing the edited contents sequentially while skipping the removed parts. Specifically, let the last
step be defined as Writing-After-Editing (shortly WAE).

It has been known that the time required to complete the WAE operation becomes considerably high, in relation to how much the multimedia file (contents) size increases [12]. To prove this, a series of preliminary experiments are conducted to measure the performance of the WAE operation by using commercial PVR software [13], which runs on top of a Windows PC and uses an external USB HDD formatted with the Ext3 file system. Figure 1 shows the required time for the WAE operation with increase of the edited multimedia file size. Note that, for the WAE operation, the conventional Ext3 initially reads all the edited data into the main memory sequentially and subsequently writes the data into the disk. Surprisingly, it is observed that the WAE operation with an edited multimedia file of 8 GB size takes more than 800 seconds (13 minutes).

In this paper, design and implementation of an amortized Ext3 with FWAE technique is described for large-size edited multimedia files for a WinXP-based computer. It runs on a Windows XP user level because PC users can easily use it and gain access to it. In order to dramatically reduce the required time for the operation, the FWAE technique is described. It re-arranges the block pointers of an inode and makes the data block shared by as many files as needed, saving the required disk space. That is, FWAE technique does not modify data of file, but modifies metadata of file. And it creates new metadata called MSDB that manages shared data blocks among files. In this paper, we only deal with large sized multimedia files such as video files, not audio files.

The remainder of the paper is organized as follows: Section 2 summarizes the related work and background for the proposed file system. In Sect. 3, the proposed file system design is described. In Sect. 4, the procedure of FWAE_API and internal implementation are described. Subsequently, Sect. 5 compares the proposed file system with the conventional Linux Ext3 using various performance metrics. Finally, this paper concludes with Sect. 6.

2. Related Work and Background

In a storage area network where many physical storage devices are connected with each other over fiber channels or IP networks, a global file system (GFS) [14] and SANtopia [15] enable data blocks to be properly shared by different server systems. The core mission of those file systems is to efficiently guarantee data integrity while allowing different server systems to concurrently access the shared data blocks. Note that the GFS provides a device-level lock mechanism for the data integrity.

We describe problems of conventional file system such as Ext3 file system, NTFS [16] and FAT [17] when large-sized file are edited and rewritten. These file systems initially read all the edited data into the main memory sequentially and subsequently write the data into the disk. In this case, not only modified part is rewritten but also the unmodified part. The conventional file editing method is suitable for small-sized files in terms of algorithm simplicity, but it requires too much time and disk bandwidth for large-sized files.

In contrast to the Ext3 that uses a constant-sized (4 KB) data block, some file systems, such as a VBS file system [18], support the use of variable-sized data blocks. The VBS file system rewrites only modified data blocks when edited. As a result, if a user wishes to delete unnecessary parts from a file, they need only to change a block size. However, maintaining a variety of different-sized data blocks in a single file system is complicated. In XPRESS (eXtendible Portable Reliable Embedded Storage System) [19], the file system metadata, such as inodes, directories, and free space information, is handled by a transactional database that guarantees metadata consistency against various kinds of system failures. The XPRESS defines five databases to store the file system metadata: superblock DB, directory DB, inode DB, free space DB, and extents DB. The extents DB maps a given file offset into an actual blocks address. Therefore, it is possible for the XPRESS to implement fast WAE by simply modifying the mapping table inside the extent DB. However, a critical problem of this approach is that the disk-free space does not increase accordingly even if the unnecessary data were removed from the original file.

A typical disk layout of the Ext3 encompasses a boot sector and a set of block groups. Each block group consists of a superblock, a group descriptor table, block bitmap, inode bitmap, an inode table, and file data blocks. Figure 2 shows the disk layout and the block group layout of the Ext3. Note that a group descriptor table contains a list of group descriptors. Each group descriptor of 32-byte size includes a starting block address of block bitmap, a starting block address of inode bitmap, a starting block address of the inode table, a free block count, a free inode count, etc. Each block group maintains its own block bitmap and inode bitmap, where each bitmap occupies a single data block. Note that the size of data block is normally 4 KB in the Ext3.
Each bit of the bitmap represents whether or not the corresponding data block or inode is used.

The Ext3 represents each file or directory as an inode (also known as metadata). Each inode is assigned with a unique ID starting with 1. Inodes are stored in the inode table in a block group. Given an inode number \( \text{inode\_num} \), the block group number \( \text{block\_group\_num} \) can be easily identified in which the inode positions itself as follows: \( \text{block\_group\_num} = (\text{inode\_num} - 1) / \text{inodes\_per\_group} \), where \( \text{inodes\_per\_group} \) represents the number of inodes in a single block group. As shown in Fig. 3, each inode maintains a list of inode block pointers (shortly IBP), which are represented from \( \text{i\_block}[0] \) through \( \text{i\_block}[14] \). The first twelve fields \( (\text{i\_block}[0] \text{ through } \text{i\_block}[11]) \) directly point to the data blocks, while the other three fields \( (\text{i\_block}[12] \text{ through } \text{i\_block}[14]) \) indirectly point to the data blocks.

### 3. The Proposed File System

#### 3.1 The Overall Architecture

Figure 4 shows the overall architecture of the proposed file system, which adds to the amortized Ext3 an engine for fast writing after editing a multimedia file (briefly FWAE engine) and a new API for the engine (shortly FWAE API). The amortized Ext3 is implemented in Window XP user level based on Ext2Read [20] open source. It provides the file read/write function like the Linux Ext3. The amortized Ext3 file system APIs will be described in detail in the following subsections.

The key of the FWAE engine is to modify associated metadata and shared information, instead of copying data blocks from the original file to the edited one. To enable this, the FWAE engine encompasses two global data structures to maintain the shared information which are a shared block map and a shared block map pointer. It also contains two modules to manipulate the global data structures which are the IBPS (IBP setting) module and the MSDB (management for shared data block) module. The IBPS module re-arranges the order of data blocks for the edited multimedia file by simply modifying inode block pointers of an associate inode. In addition, the MSDB module is responsible for maintaining data block sharing between the original (multimedia) file and the edited (multimedia) file. The subsequent subsections will provide detailed descriptions on each module.

#### 3.2 Amortized Ext3 File System API

First, the amortized Ext3 file system provides a set of file I/O functions that include auto mount, file read, file write and file delete. To store the edited contents as a new file or to overwrite them onto the original file, it is necessary to use the file write and file delete functions. Ext2fds support file write and file delete function. However, we do not use Ext2fds because of compatibility of our previous research [24]. For that reason, we added file write and file delete functions to Ext2Read open source software. Table 1 shows comparison of capabilities between the amortized Ext3 file system and other open source software to manipulate the Ext3 disk.
Table 1 Comparison of capabilities between the amortized Ext3 file system and other open source software to manipulate Ext3 disk.

|           | File read | File write | File delete | Auto mount | Journaling |
|-----------|-----------|------------|-------------|------------|------------|
| Amortized Ext3 (Win XP) | O | O | O | O | X |
| Ext3 (Linux) | O | O | O | O | O |
| Ext2Read (WinNT, XP) | O | X | X | O | X |
| Ext2fs [21] (Win NT) | O | X | X | O | X |
| Ext2fs [22] (MAC OS X) | O | O | X | O | X |
| Ext2fs [23] (Win XP) | O | O | O | O | O |

The amortized Ext3 file system API allows user-level multimedia applications to exploit file I/O, such as file read, file write and file delete. First, file read API provides to read assigned-sized file data from the designated file offset in the file: int AExt3_read(unsigned int ino, long long offset, void *buffer, int bufSize). The ino represents a file’s inode number wanted to read, and offset represents start position wanted to read in the file. The “buffer” functions to store read data. The bufSize represents size (bytes) of read data. Second, file write API provides to write assigned-sized file data to the designated file offset in the file: AExt3_write(unsigned int ino, long long offset, void *buffer, int bufSize). Finally, file delete API provides to delete file: int AExt3_delete(unsigned int ino).

3.3 FWAE_API Design

The FWAE API (function call) allows user-level multimedia applications to exploit the fast write feature combined the amortized Ext3 with FWAE Engine, in order to speed up the writing process after editing a large-size multimedia file. It can be provided as one of several user-level (external) API calls: int fwae(char[] src_path, char[] dest_path, struct EDIT_INFO *edit_info, int mode). The src_path represents an original file full path, and the dest_path represents a newly created file full path. The “edit_info” encompasses a set of editing segments, each of which corresponds to a consecutive data region. Note that an editing segment is represented with a pair of an offset and a size, as shown in Fig. 5. The mode represents one of these two writing modes: 1) writing the edited contents into the original file (overwrite mode), and 2) writing the edited contents into a new file (a new file mode). In addition, the existing file delete API should be modified accordingly, so that the data block sharing can be handled in a proper way.

3.4 MSDB (Management for Shared Data Block) Metadata

The FWAE engine requires extra metadata to maintain the information for the shared data blocks between different files. Let this metadata be called MSDB metadata. Recall that the MSDB metadata is created only when the edited contents of the original file are saved as a new file. Figure 6 represents the data structure of the MSDB metadata that consists of an edited file manager (EFM), a shared block map pointer (SBMP), and a shared block map (SBM).

For each data block, the SBM maintains the number of files that share an associated data block. A 1-byte field is allocated to each data block in order to retain the number of data block sharing. Recall that the block bitmap in the Ext3 keeps the information of data block usages in a block group, and it occupies a single data block where a single bit can be used to inform the usage of a data block. Since each byte of the SBM is associated with each bit of the block bitmap, the SBM consumes eight data blocks. The SBMP contains a data block address of each of the SBMs. Note that the i_reserved[1] field of an inode of the shared file points to the block address of the SBMP.

The EFM encompasses the number of files that share data blocks with each other. The number of the shared files is stored in the first four bytes. The EFM keeps a list of the associate inode numbers. It also maintains a list of block groups (actually their numbers), where data blocks of the original file reside. Recall that each block group has its own block bitmap of a single data block. The data block address of the EFM is stored in i_reserved[0] field in inode of shared files. Additionally, a SBM of eight data blocks was also introduced. As shown in Fig. 6, an associate SBM can be accessed from the block group number in the EFM through
the SBMP information. To access an associate 1-byte field in SBM, it calculates the location of a 1-byte field in SBM based on data block address. This procedure is similar to the procedure of the accessing a 1-bit-value in the Ext3 block bitmap except that a block bitmap is represented SBMs of eight data blocks.

4. FWAE API Use Case and Procedures

4.1 FWAE API Use Case

Figure 7 shows the possible states of the shared files when calling various combinations of the FWAE and delete operations. Two different states exist: State 1 with no data sharing and State 2 with data sharing. The initial state of the original file begins with State 1. The file state remains in the State 1, as long as the FWAE operations with the overwrite mode are called. FWAE operation consists of two main functions such as IBP setting and manage_MSD. Note that such FWAE operations will be mainly handled by the IBP setting procedure in the IBPS module. When an FWAE operation with the new file mode is called, the file state is changed to the State 2 from the State 1. This transition requires the creation of the shared information (create_MSD) and the initialization for the data block sharing properly (manage_MSD(SHARE)). As long as the FWAE operations with the new file mode are called repeatedly, the file state remains at the State 2 with the creation of the inode metadata for the new file. When the FWAE operation with the overwrite mode is called at the State 2 (data sharing mode), the IBP setting procedure additionally needs to modify the shared information as well (manage_MSD(UPDATE)). Finally, the file delete operation for the shared file requires to modify the shared information accordingly (manage_MSD(UNSHARED)). When no shared files exist, the file state is changed into the State 1 from the State 2 by removing the shared information (delete_MSD). More detailed descriptions on each procedure will be given in next sections.

4.2 FWAE API Procedure and Internal Implementation

As previously described, the application can exploit the proposed FWAE operations through the user-level FWAE API. Recall that the FWAE API supports two different types of operations: the overwrite mode and the new file mode. Figure 8 shows the procedure for the two operations. It receives four inputs which are the source inode information (inode src), the destination inode information (inode dest) in case of the new file mode, the edit information (edit info) including the editing segment set, and one of the two modes. To begin, we describe the FWAE operation when the edited contents are overwritten into the original file. The FWAE operation firstly translates file’s full path into file’s inode number. Next, the FWAE operation calls the IBPS module described in Sect. 4.2.1 as soon as the FWAE API is called. In case of the new file mode, it additionally calls the manage_MSD with the SHARE mode, described in Sect. 4.2.2, to manage the shared metadata information (MSDB).

4.2.1 The IBP Setting (IBPS) Module

IBPS module is responsible for elaborately rearranging the contents of the i_block fields of the inode structure according to the edit_info information. Figure 9 shows the main procedure of the IBPS module. The procedure receives four inputs which are the source inode information (inode src) mapped into the original file, the destination inode information (inode dest) mapped into the new file in case of the new file mode, the edit information (edit info) including the editing segment set, and a writing mode. To begin, the overwrite mode requires allocation of a temporary destination inode that will be eventually copied into the source inode (lines 4–6). The core of the procedure is to re-arrange the inode block pointer information (i_block) through i_block[14]) of the source inode into those of the destination inode according to the information of the editing segments (lines 7–19). At this step, it re-configures the off-
set of the editing segments to prevent the loss of I-picture in the GOP (Group Of Pictures). If the loss of I-picture happen, the quality of the multimedia drops because the I-picture affects both P-pictures and B-pictures in the same GOP (lines 9–10).

Next, if the original file is not shared ($i_{reserved}[0] != \text{NULL}$), the IBPS module resets the block bitmap associated with data blocks of the $inode_{src}$ (data blocks of the $inode_{dest}$) (lines 22–26). That is, the IBPS module deallocates data blocks not belonging to the editing segments by updating the block bitmap associated with the data blocks. If the original file has been shared ($i_{reserved}[0] == \text{NULL}$), the IBPS module calls $manage\_MSDB(UPDATE)$ to manipulate data block sharing (lines 28–29). Finally, as mentioned before, the contents of the temporary inode ($inode_{dest}$) are copied into the original inode ($inode_{src}$). With the new file mode, the IBPS module needs to call the $create\_MSDB$ procedure in order to create the MSDB metadata if MSDB metadata previously is not created (line 34).

It can be observed that the edited file will be GOP (Group Of Picture)-aligned. The GOP is a group of successive pictures within a coded video stream. Each coded video stream consists of successive GOPs. From the pictures contained in it, the visible frames are generated. By performing the GOP-aligned editing, it is possible to increase the quality of multimedia contents because the frames are not damaged. The audio signal also is not damaged because it is packed in each frame [25]. To perform the GOP-aligned editing, it is necessary to reconfigure the offsets of editing segments. From selected offset, it read disk block forward until GOP header appear. If it finds GOP header, it modifies offset and size of editing segment. If FWAE is interworking ‘Avidemux’ [26] that is multimedia player supporting GOP-aligned file editing function, GOP-aligned editing does not need because Avidemux delivers the reconfigured offset to FWAE APIs.

Figure 10 shows an illustrative example of the operation of the IBPS module, where data 13–n have been removed from the original file. Recall that the inode maintains 12 direct block information and 3 indirect block information including indirect, double indirect, and triple indirect mapping. From this example, we can expect that the IBPS module will provide a FWAE operation by avoiding physical data copies from/to disks.

### 4.2.2 The MSDB (Management for Shared Data Block) Module

In this section, detailed descriptions will be given on when the metadata will be created and deleted, and how the contents of the metadata will be changed. The main task of the MSDB module is to create and properly manage the MSDB metadata.

Figure 11 shows the procedure of creating the MSDB metadata. The three metadata data structures are created only when the edited contents of the original file are saved as a new file (the new_file mode). First, the MSDB module
allocates data blocks for the EFM and fills the EFM with the values of the number of shared files and inode number of the shared files (lines 3–7). Second, it scans all of the data blocks in the original file \((\text{inode src})\) to register their associate group numbers to the EFM (lines 9–12). In addition, the block bitmap information in each block group is converted and stored into the SBM (line 17–18). Note that the number of data blocks for the SBM is the number of groups * 8 data blocks. Finally, the block addresses of the SBM and the EFM are respectively stored in \(i\) reserved[0] and \(i\) reserved[1] of the \(\text{inode src}\) and the \(\text{inode dest}\) (lines 21–24). Note that, \(i\) reserved[0-2] field is originally designed to store specific operating system information. However, it is not used for any purpose. Therefore, our proposed technique used this field to store block addresses of SBM and the EFM.

Figure 12 shows the procedure of deleting the MSDB metadata. The procedure receives three inputs which are the source inode information \((\text{inode src})\), the destination inode information \((\text{inode dest})\) and the \(m\) mode. The \(m\) mode is represented as one of three managing modes: 1) managing the MSDB metadata when edited contents are stored as a new file (SHARE mode), 2) managing the MSDB metadata when one of the shared files is deleted (UNSHARED mode) and 3) managing the MSDB metadata when the edited contents are overwritten into one of the shared files (UPDATE mode). The \(\text{inode dest}\) is inode information of the new file in case of the SHARE mode, it is NULL in case of the UNSHARED mode, and it is a temporary inode that has been created by the IBPS module in case of the UPDATE mode.

First, the procedure of managing the MSDB metadata in case of the SHARE mode is described. It reads the EFM and the SBMP associated with the deleting file and decreases the number of shared files field in the EFM (lines 3–4 and 6). Also, it adds the inode number of the new file into the list of inode numbers field in the EFM (lines 7–8). Next, the procedure finds the proper 1-byte value that is associated with the data blocks of the \(\text{inode dest}\) in SBM and decreases the 1-byte value associated with the data blocks of the \(\text{inode src}\) in the SBM (lines 29–33). If
Fig. 13 The procedure of managing the MSDB metadata 1) managing the MSDB when edited contents are stored as a new file (SHARE mode), 2) managing the MSDB when one of shared files is deleted (UNSHARED mode) and 3) managing the MSDB when the edited contents are overwritten into one of the shared files (UPDATE mode).

Fig. 14 The procedure of the modified file deletion operation.

the 1-byte value that represents the number of files sharing instances becomes zero, the procedure clears an associated bit value in the block bitmap (lines 33–37). If the number of shared files field in the EFM is equal to one (no data block sharing), it calls up the procedure of deleting the MSDB metadata. It deletes inode and directory entry associated with `inode_src`, because managing of the MSDB metadata (UNSHARED mode) is called when one of the shared files is deleted (lines 39–43). Otherwise, the procedure saves the modified EFM (lines 44–46).

Finally, the procedure of managing the MSDB metadata in case of the UPDATE mode is described. The procedure finds the proper 1-byte value and increases the 1-byte value associated with the data blocks of the `inode_dest` (lines 48–52). Note that this procedure is similar to the procedure of managing the MSDB metadata in case of the SHARE mode. Then, the procedure finds the proper 1-byte value and decreases the 1-byte value associated with the data blocks of the `inode_src` (lines 53–58). Note that this procedure is similar to the procedure of managing the MSDB metadata except for calling the procedure of deleting the MSDB metadata.

4.2.3 The Modified File Delete Operation

File deletion operation considers the procedure when the shared file is deleted in the FWAЕ technique. Accordingly, it is necessary for the modified file deletion operation to manage MSDB metadata when the shared file is deleted. That is, the procedure of the deleting shared file is added to the procedure of the deleting file in Ext3. But user-level file deletion API does not change. Figure 14 shows the procedure of the modified file deletion. The procedure considers whether the deleting file is the shared file or not. That is because procedure of the deleting file is different from whether the deleting file is the shared file or not. Therefore, it checks if the deleting file has already been shared by simply examining the value of `i_reserved[0]` in the inode of the deleting file. If `i_reserved[0]` is NULL, it deletes the relevant file using the same procedure as that of the deleting file in Ext3 (lines 3–5). If `i_reserved[0]` is not NULL, it calls the procedure of managing MSDB with unshared managing mode (lines 6–8).
5. Performance Evaluation

In this section, the performance of the proposed file system (the amortized Ext3 with FWAE) is evaluated during WAE operation, as compared with the Linux Ext3 file system. On the aspect of performance, our experiments show that the read/write speed of the amortized Ext3 is 1.4 times slower than read/write speed of Ext3 is implemented in Linux OS with kernel version 2.6.18. The experimental platform consists of 1.86 GHz * 2 Intel Core 2 processor and 1 GB of main memory. The USB disk used in this experiment is FUJITSU Calmeel MOON FKL-GP3 230 GB USB disk. File editing, file deleting and disk space utilization performance are evaluated between the proposed file system (amortized Ext3 + FWAE) and Ext3. Table 2 summarizes the environment of the experiment.

| The proposed file system | Targets of comparison |
|--------------------------|-----------------------|
| OS                       | Windows XP service pack 2 |
| File system              | Amortized Ext3 with FWAE |
| CPU                      | 1.86GHz * 2 Intel Core 2 processor |
| Main Memory              | Samsung DDR 1GB |
| USB disk                 | FUJITSU Calmeel MOON FKL-GP3 230 GB |

5.1 Required Time Comparison When Storing Edited Contents

Figure 15 shows the results of comparison of the required times with various WAE operations for the proposed file system (Amortized Ext3 with FWAE), Amortized Ext3 (without FWAE) and the Ext3. Note that Ext3 reads edited contents into memory, and then rewrites them in disk. And WAE mechanism of amortized Ext3 is the same as that of Ext3. The proposed file system removes the deleting segments of the original file by using IBPS. The bigger the edited file size, the longer the time is required to store edited files for both the proposed file system and Ext3. But the difference between the required time in the proposed file system and the required time in Ext3 increases. The experiments for comparison of the required time are performed for various cases.

The required time of the proposed file system is at average 16 times less than that of Ext3 for all cases of the WAE operations. And the required time of the proposed file system is at average 22.4 times less than that of Amortized Ext3 for all cases of the WAE operation. Note that the amortized Ext3 is 1.4 times slower than read/write speed of Ext3, because Ext3 makes better optimization of metadata caching than amortized Ext3.

First, the result of comparison of the required time is
Fig. 16 The comparison of the required disk space, original file size is fixed (10 GB), new file mode (with MSDB creation).

Fig. 17 The comparison of the required time during file deletion operation, without deleting MSDB.

described, in the case when edited contents are overwritten in the original file and the original file does not have shared data blocks. As shown in Fig. 15 (a) and (b), the required time of the proposed file system is 14 times less than that of Ext3 and 20 times less than that of Amortized Ext3. Figure 15 (c) and (d) shows the result of comparing the required time, in the case when edited contents are stored in a new file and MSDB metadata is created. The required time of the proposed file system is similar to the results of the experiment shown in Fig. 15 (a) and (b). These results are the worst cases because the time needed to create MSDB metadata is as long as the time needed to scan block bitmaps associated with data blocks of the original file. The time to create MSDB metadata is directly proportional to the original file size regardless of the edited file size. On the other hand, Ext3 saves time because it does not need to delete the original file, as in the case when the edited contents are overwritten in the original file. The required time of the proposed file system is 13 times less than that of Ext3. Figure 15 (e) and (f) shows the result of comparison of the required time, in the case when the edited contents are stored in a new file and MSDB metadata is created. In this case, the WAE performance of the proposed file system is best of all, because it does not need to create MSDB metadata or to perform file deletion operation. The required time of the proposed file system is 23 times less than that of Ext3. The reason that the proposed file system has shown dramatically good performance for the WAE operations compared with Ext3, the proposed file system modifies only metadata without data I/O from the disk when the WAE operation performs.

5.2 The Comparison of the Required Disk Space

Figure 16 shows the result of the comparison of required disk space for the proposed file system and the Ext3, when the FWAE operation with new file mode (with MSDB creation) is called. In Ext3, the edited contents are written in free disk space in case of the WAE operation with new file mode. But the proposed file system uses shared data blocks for the original file and a new file. It requires extra disk space to store MSDB metadata, but it is only 0.03% of the original file size. In the proposed file system, the more files share data block, the better is the required disk space in comparison with the required disk space in Ext3. The proportion of the required disk space for the proposed file system and the Ext3 is described as (1).

\[ p = \text{The largest file size among } \sum_{i=1}^{n} \text{ size of } f_i \]  

In Eq. (1), \( F \) is a set of shared files and it is represented as follow: \( F = \{f_1, f_2, \ldots, f_n\} \). \( f_i \) is the shared files.

5.3 The Comparison of Required Time when Deleting File

Figure 17 shows the result of the comparison of required times with the proposed file system and Ext3, when the file is deleted. In Ext3, we estimate the required time when general file deletes. In the proposed file system, we estimate the required time when the shared file deletes by using the modified file deletion operation (without deleting MSDB). The modified file deletion operation in the proposed file system is 6.5 times faster than the file deletion operation in Ext3. The reason for the performance improvement is reduced disk seek time. Ext3 reads only direct pointer blocks and indirect pointer blocks associated with the deleting file when a file is deleted. Indirect pointer blocks stored the address of data blocks are distributed in disk space. If the file is large, it takes a longer amount of time to access indirect pointer blocks, because a large file has more data blocks. But in the proposed file system, a file newly created through writing a new file FWAE operation has continuous indirect pointer blocks, because the IBPS module sequentially allocates free blocks so as to use indirect pointer blocks stored in the addresses of data blocks. Thus, even if the file size is large, the proposed file system takes less time to access indirect pointer blocks.

5.4 The Evaluation of Drawbacks in the Proposed File System

We describe the merits of our proposed technique in the previous sections (Sect. 5.1 to Sect. 5.3). In this section, we evaluate the drawbacks caused by the file fragmentation
with the running FWAE. The proposed file system provides a FWAE operation by avoiding physical data copies from/to a disk. Therefore, it is possible to edit a multimedia file fast by modifying only its metadata. But, file fragmentation occurs when a single file has been broken into multiple pieces to delete only a deleting segment in a file. While disk file systems like the Ext3 attempt to keep individual files contiguous, this is not often possible without significant performance penalties [27]. File system fragmentation increases disk head movement or seeks, which are known to hinder throughput. Thus, a file read performance is estimated when the file fragmentation occurs, and the file fragmentation is not a serious problem to playback multimedia contents. The read performance of a general continuous file (defragmentation file) is 28 MB/s. Two fragmented files are used to estimate the read performance of fragmented files: 1) 64 MB sized file consists of 250 fragmented pieces (256 KB continuous data), 2) 1 GB sized file consists of 250 fragmented pieces (4 MB continuous data). The read performance of file 1) is 13.06 MB/s, and it is 46.6% of a general file’s read performance. The read performance of file 2) is 26.74 MB/s, and it is 97% of a general file’s read performance. It is observed that the fragmented file having the longer continuous data (file 2) has a better read performance than the fragmented file having the shorter continuous data (file 1). Also, the multimedia files used to the proposed file system have the longer continuous data area than file 1 or file 2. Note that the multimedia files used to the proposed file system have several MB or GB continuous data, though it is different from depending on the types of file editing. Thus, the multimedia files that were edited by using the proposed file system rarely affect the read performance penalties. Also, In the worst case, though the continuous data is shorter (file 1), the read performance of the fragmented file (13.06 MB/s) satisfies the playback performance of the HD multimedia contents (2.5 MB/s).

5.5 Limitations of the Amortized Ext3 with FWAE

In this section, we describe the limitations of the amortized Ext3 with FWAE operation. Firstly, we describe the compatibility of the amortized Ext3 with FWAE compared with Linux Ext3 file system. The amortized Ext3 with FWAE is partially compatible with Ext3 file system. The edited multimedia files using FWAE operation in the amortized Ext3 are readable in Ext3 file system, because the FWAE operation does not change file’s inode structure. For the same reasons, attributes of the edited multimedia files such as owner, permission, and etc. can be changeable in Ext3 file system. However, the modification of the file data shared with other files, is not possible because of the modification of the MSDB metadata created by the amortized Ext3 with FWAE. Also, disk volume, modified by the amortized Ext3 with FWAE, can be mounted in Linux kernel, because Ext3 metadata structure does not change. However, deleting multimedia files that were edited in the amortized Ext3 with FWAE and sharing data blocks causes problem. FWAE technique provides sharing data blocks between several files by managing MSDB metadata, but Linux Ext3 does not handle MSDB metadata. MSDB metadata exist only for FWAE technique. The Compatibility problem of the Amortized Ext3 with FWAE operation can be resolved by implementing FWAE technique in Linux Ext3.

Secondly, FWAE technique has the limited number of files sharing associated data block. As it was described in Sect. 3.4, for each data block, the SBM maintains the number of files that share an associated data block. A 1-byte field is allocated to each data block in order to retain the number of data block sharing. A 1 byte field represents maximum 255. Therefore, the number of files sharing associated to a data block limits maximum 255. However, it is sufficient to retain the number of data block sharing because the case that same multimedia file edit more than 255 can rarely occur.

Third, the amortized Ext3 with FWAE does not implement file lock mechanism. If two or more applications are called FWAE_API for the same file at the same time, it may occur problem. However, typical PC users do not run application such as media player at same time because its performance decreases due to frequent disk head movements. Therefore, we will implement file lock mechanism in future work.

Finally, the size of the shared file is limited to 16 GB in this version because the SBMP contains 1024 SBM block addresses and each SBM stores 4096 data blocks. Therefore, the maximum shared file size is 16 GB. However, it is possible for the SBMP to extend in the i_reserved field which is reserved. Then, the maximum shared file size is 32 GB.

Except the above directions, the amortized Ext3 with FWAE does not have any limitations.

6. Conclusion and Future Work

There is a growing user demand for multimedia devices, and such devices provide various services associated with multimedia file manipulation, including multimedia content playback, multimedia file editing, etc. Additionally, digital TV provides a recorded multimedia file copy to a portable USB disk. However, Linux Ext3 file system, as employed by these devices, has a lot of drawbacks, as it required a considerable amount of time and disk I/Os to store large-size edited multimedia files, and it is hard to access for typical PC users. In this paper, the design and implementation of an amortized Ext3 with FWAE technique is described for large-size edited multimedia files for a WinXP-based computer. It runs on a Windows XP user level because PC users can easily use it and gain access to it. These related experiments show that the proposed file system improves write performance on average by 16 times compared with Ext3 with various types of edited multimedia files and reduces consumed disk space dramatically through data block sharing. If the proposed file system is combined a multimedia application, it will be a fast and useful video editor as a Window XP user level application.
Recently, digital cameras that used a large-sized flash memory as storage devices enable users to record high-quality multimedia contents. Also, a digital camera provides a file editing function. The performance problems that happen when file editing is more serious in NAND flash memory that has a low read/write bandwidth in comparison with hard disk. Currently, the research for the proposed technique applies to YAFFS [28] which only used a NAND flash file system.

References

[1] Y. Ninomiya, “High definition television systems,” Proc. IEEE, vol.83, no.7, pp.1086–1093, July 1995.
[2] L. Zong and N. Bourbakis, “Digital video and digital TV: A comparison and the future directions,” Proc. 1999 International Conf. on Information Intelligence and Systems, pp.470–481, Nov. 1999.
[3] M. Sadiku and S. Nelatury, “High definition television in detail,” IEEE Potentials, vol.26, no.1, pp.31–35, Jan. 2007.
[4] S.Y. Lim, J.H. Choi, J.M. Seok, and H.K. Lee, “Advanced PVR architecture with segment-based time-shift,” International Conf. Consumer Electronics, pp.1–2, Jan. 2007.
[5] H. Jo, J. Kang, S. Park, J. Kim, and J. Lee, “FAB: Flash-aware buffer management policy for portable media players,” IEEE Trans. Consumer Electron., vol.52, no.2, pp.485–493, May 2006.
[6] SEMICO Research Corp., “Flash memory vs. Hard Disk drives - which will win?,” http://www.storagesearch.com/semico-art1.html.
[7] D. Bovet and M. Cesati, Understanding the Linux Kernel, 3rd ed., pp.738–774, O’Reilly, 2006.
[8] M. McKusick, W. Joy, S. Leffler, and R. Fabry, “A fast file system for UNIX,” ACM Trans. Comput. Syst., vol.2, no.3, pp.181–197, 1984.
[9] Whitepaper: Red hat’s new journaling file system: Ext3,” http://www.redhat.com/support/papers/redhat/ext3/ext3.pdf.
[10] Linux Online Inc., “GNU general public license,” http://www.linux.org.
[11] Electronic Times Internet, “A match made in heaven: Digital TV and Linux,” http://english.etnews.co.kr/news/detail.html?id=200803050007.
[12] T. Ts’o, “Planned extensions to the Linux Ext2/Ext3 file system,” Proc. 2002 USENIX Ann. Technical Conf., pp.235–243, 2002.
[13] LG Electronics, “Xcanvas Player ver2.0,” http://www.xcanvase.co.kr.
[14] K. Preslan, A. Barry, V. Brassow, G. Erickson, E. Nygaard, C. Sabel, S. Soltis, D. Teigland, and M. O’Keefe, “A 64-bit, shared disk file system for linux,” Proc. 16th IEEE Mass Storage Systems Symp., pp.22–41, March 1999.
[15] Y.K. Lee, S.W. Kim, G.B. Kim, and B.J. Shin, “Metadata management of the SANtopia file system,” Proc. 8th International Conf., pp.492–499, June 2001.
[16] Microsoft TechNet, “How NTFS Works,” http://technet.microsoft.com/en-us/library/cc781134(WS.10).aspx.
[17] Microsoft TechNet, “How FAT Works,” http://technet.microsoft.com/en-us/library/cc776720(WS.10).aspx.
[18] P. Corbett and D. Feitelson, “The vesta parallel file system,” Scalable High-Performance Computer Conference, May, 1994.
[19] J.Y. Hwang, J.K. Bae, A. Kirmesov, M.S. Jang, and H.Y. Kim, “A reliable and portable multimedia file system,” Proc. Linux Symp., July 2006.
[20] SourceForge, Inc., Ext2Read, http://sourceforge.net/projects/ext2read.
[21] SourceForge, Inc., ext2ifs, http://sourceforge.net/projects/freshmeat_ext2ifs/.
[22] SourceForge, Inc., ext2fs, http://sourceforge.net/projects/freshmeat_ext2fs/.
[23] Ext2fss, http://ext2fsd.sourceforge.net/projects/projects.htm.

Dae-Wha Seo received the B.S. degree in Kyungpook National University in 1981 and the M.S. and Ph.D. degrees in department of computer science from Korea Advanced Institute of Science and Technology in 1983 and 1993 respectively. He was a researcher at the Electronics and Telecommunications Research Institute from 1983 to 1995. He is now a professor at Kyungpook National University from 1995. His research interests include system software and realtime system. Dae-Hwa Seo is also a head of Center for Embedded Software Technology.

Seung-Wan Jung received the B.S. and M.S. degree in school of Electrical engineering and Computer science, Kyungpook National University, Daegu, Korea, in 2006 and 2008 respectively. He is Ph.D candidate of Kyungpook National University. He is current research interests include computer system architecture, embedded system, file system.

Young Jin Nam received the B.S. degree in Kyungpook National University in 1992 and the M.S. and Ph.D. degrees in department of computer science and engineering from Pohang university of Science and Technology in 1994 and 2004 respectively. He is now a professor at Daegu University from 2004. His research interests include low-power embedded software design and embedded wireless network.

Young Jin Nam received the B.S. degree in Kyungpook National University in 1992 and the M.S. and Ph.D. degrees in department of computer science and engineering from Pohang university of Science and Technology in 1994 and 2004 respectively. He is now a professor at Daegu University from 2004. His research interests include low-power embedded software design and embedded wireless network.