Deduplication TAR Scheme Using User-Level File System

Young-Woong KO†a, Member, Min-Ja KIM††, Nonmember, Jeong-Gun LEE†, and Chuck YOO†††, Members

SUMMARY
In this paper, we propose a new user-level file system to support block relocation by modifying the file allocation table without actual data copying. The key idea of the proposed system is to provide the block insertion and deletion function for file manipulation. This approach can be used very effectively for block-aligned file modification applications such as a compress utility and a TAR archival system. To show the usefulness of the proposed file system, we adapted the new functionality to TAR application by modifying TAR file to support an efficient sub-file management scheme. Experiment results show that the proposed system can significantly reduce the file I/O overhead and improve the I/O performance of a file system.

key words: user-level file system, TAR, file allocation table, block management

1. Introduction

Applications using block-aligned data I/O can benefit from efficient sub-file management scheme. First, in Winzip compress tools, sub files are concatenated into one compressed file. If a user try to add or remove a sub-file in the compressed file, then the application usually has to rewrite entire data blocks in an old file and causes massive disk I/O. Second, TAR archive file contains lots of sub-files and it also suffers from sub-file insertion and deletion. Third, sub-file management scheme is also useful for multimedia editing. In digital video editing, it can access any frame, and use a cut-and-paste method, similar to the ease of cutting and pasting text in a word processor. In multimedia editing, by relocating blocks of digital video data, we can reduce disk I/O.

What happen if we delete or insert several blocks into a file with block-aligned manner? Suppose there is a file with 40 KByte size and the block size of the file system is 4 KByte. The total number of data blocks for storing the file will be ten. When an application inserts a data block with 4 KByte size on the file position 0, the application has to overwrite the old file. Although there are same blocks on the old file, the file system rewrites all the blocks to the new file. This traditional file modification approach causes massive disk I/O requests when we modify a file even though the file modification happens in block-aligned manner. The block-aligned file I/O is very frequent in lots of applications including compress and archive applications. However, traditional file system does not support block I/O for a file because it considers a file as a stream of byte.

There were several research results to tackle this problem. Raw disk I/O [1] is a good alternative for providing user-level direct block management, which is widely used in DBMS to improve the I/O performance. A raw device is a special kind of block device file that allows accessing a storage device directly, bypassing the operating system buffer. However, standard file manipulation cannot be used on raw devices and there are many administrative inconveniences. Data deduplication schemes [2]–[4] are another possible solutions to overcome massive file I/O by using fingerprint scheme such as MD5, SHA1, or SHA256. One of the well-known data deduplication approach is DEXT3 [5] which supports block-level inline deduplication layer for EXT3 file system. DEXT3 identifies the possibility of writing redundant data to the disk by ensuring duplicated blocks on the file system. But, data deduplication scheme targets for duplicated contents between files, therefore, it is difficult to use data deduplication approach for performance enhancement scheme of modifying a file.

In this paper, we propose a user-level file system to support block relocation by modifying the file allocation table without actual data copying. The key idea of the proposed system is to provide the block insertion and deletion function for file manipulation. To support efficient file modification, we propose a novel file system by modifying FAT architecture called UFFS (User-level FAT File System). We adopted the FAT architecture as a user-level file system because FAT is very simple and easy to implement. Furthermore, a user-level file system is portable and easy to adapt to various operating systems. We can avoid complicated kernel-internal API and harder debugging situations, crash and reboot.

2. System Architecture of User-Level File System

The FAT file system is one of the most simple types of a file system. It consists of boot sector, file allocation table and data blocks (clusters) to store files and directories. The metadata of a file is stored in directory blocks which are the array of file records. In the file record, cluster number references a first block of a file and next blocks could be found through file allocation table by using it as linked-list. The file allocation table is a kind of small map for data blocks...
and each entry value indicates block reference to next block of the file. Zero value of the entry means that the block is not used and special value (0xFFF) indicates end of file. In this example, a first cluster number of foo file is 100 and the file occupies 6 clusters, 24 KByte (suppose cluster size is 4 KByte). If we insert 4 KByte data on file offset 4096 then 6 clusters will be rewritten to the file system. Therefore total 24 KBytes disk writing work will be happen. When we delete 8 KByte size region of a foo file on offset 8196, 3 clusters will be written to the file system.

2.1 Design for UFFS Supporting Block Relocation

If the data deletion happens in block-aligned position and the data size is a multiple of block size then we can avoid actual data copying by applying metadata modification. In FAT file system, we can delete or insert a block by modifying cluster number on file allocation table as if we insert or delete an entry on linked-list. In this work, we provide the block relocation function for data modification on UFFS. The new file I/O function, bmodify (fd, buffer, offset, size, option) can relink the cluster number of file allocation table between offset and offset+size region. Figure 2 shows the overall architecture of the proposed User-level File System.

The proposed system creates a large-capacity file (10 GByte file in this implementation) on Linux EXT4 file system and logically divides the file into several regions including FAT, DIR, and data blocks. As can be seen Fig. 2, UFFS partition is a regular file on the Linux file system. Accordingly, we can fully control the UFFS file system with general file I/O operation of EXT4 file system. All of the functions in UFFS are very similar to general file system functions except UFFS_bmodify(). In UFFS, several blocks of data can be deleted or inserted to a file by specifying offset and size. For example in Fig. 3, we can insert 4 KByte data on foo file from file offset 4096 without actual data copying. First, the proposed system gets a free cluster (cluster number is 103) from UFFS file system then copies 4 KByte data from offset to cluster 103. Second, 100th cluster number in file allocation table is changed to 103 and 103 cluster number is changed to 101 as if we insert an entry into linked-list. Block deletion is also handled without actual data copying. If we want to delete two blocks from foo file on offset 8192, all we have to do is to modify cluster numbers and to delete two entries by marking 000. In this example, all the disk write works are only 1 block when we insert a block on foo file. Block deletion does not cause disk writing for data blocks.

2.2 TAR Application on UFFS

The original TAR file format was developed to write directly to sequential I/O devices for tape backup purposes. It is now commonly used to collect many files into one large file for distribution or archiving, while preserving file system information such as user and group permissions, dates, and directory structures. In the TAR file format, the entire sub-file is aligned with the disk block, which means that the data is written unaltered, except that the file length is rounded up to a multiple of 512 bytes (or the size of a disk block) and the extra space is filled with zeroes. In a traditional file system, if we delete a sub-file in a TAR file, then subsequent blocks will be rewritten to disk.

In this work, to show the usefulness of UFFS, we modified the TAR file format to support bmodify() function. TAR file can benefit from the bmodify() function when a sub-file
is added or deleted, because each sub-file is aligned along with block size by padding null data at the end of the sub-file. Actually, all the sub-files on a TAR file are aligned along with 4 KByte block-size, therefore, sub-file deletion and insertion can utilize the \texttt{bmodify()} function. Figure 4 shows the architecture of a TAR system.

3. Evaluation Results

In this experiment, we made several experiments to draw all of the aspects of the proposed system. Our hardware platform has a quad-core processor (Intel Xeon E5520 2.27GHz with 256 KB L2 Cache and 8MB L3 Cache), 4GB DDR3 RAM, 1TB Seagate hard disk with 7200 RPM and Gigabit ethernet. Because the experiment result with small sub-files shows very short time to evaluate, we made an 1 GByte TAR file that composed of 100 MByte sub-files. Experiment categories are three folds: creation time of a TAR file, sub-file management time on each file system and sub-file management time on different file position. To show the effectiveness of the proposed system, we tested on several platforms including Linux file system(EXT4), the proposed user-level file system with \texttt{bmodify()} function(UFFS\_BM) and user-level file system without \texttt{bmodify()} function(UFFS\_NBM). The difference between UFFS\_BM and UFFS\_NBM is to support file modification, therefore, file creation and file deletion processes are identical.

In the first experiment, we show the block I/O patterns when we delete a sub-file in a TAR file on a native file system. The grey region indicates the volume of disk reads and the line indicates the volume of disk writes. If we delete a sub-file inside a TAR file, the subsequent sub-files have to be rewritten to disk. Therefore, the closer a sub-file is to the beginning of the TAR file, the greater is the overhead cost of deleting it. Figures 5 (a) and 5 (b) show the volume of I/O operations when we delete a sub-file in the front section and another in the rear section of a TAR file, respectively. Note that the volume of disk writes is dramatically decreased when we delete a sub-file in the rear part of a TAR file.

In the second experiment, we created an 1 GByte TAR file that has 10 sub-files on local file system(EXT4), UFFS\_BM and UFFS\_NBM. We measured the processing time when we deleted a sub-file in a TAR file. To prevent buffer cache effect, the proposed platform was rebooted after each experiment.

As illustrated in Fig. 6, a user-level file system needs additional metadata management overhead such as file allocation table management and directory entry creation on UFFS. In this experiment, overall processing time was consumed in writing data blocks to hard disk. As you can probably guess, the TAR file creation overhead in UFFS is slightly higher than that of EXT4. In this experiment, the file creation time between UFFS\_BM and UFFS\_NBM is the same as we noted earlier.

In the third experiment, we tested sub-file deletion and sub-file insertion on each platform. Because sub-file deletion time is varying with sub-file position in Ext4 file system, we fixed the file position in the middle of a file for comparison purpose. The TAR application running on EXT4 file system...
system, only supports sub-file append at the end of a file and does not support sub-file insertion in the middle of TAR file. Therefore, we only evaluated the processing time of sub-file insertion at the end of a file.

As we can see in Fig. 7, the file append work requires block writing overhead and metadata management overhead. Therefore the performance result of processing time is very similar between the file systems. In sub-file deletion in the middle of a file, UFFS_BM gives the best performance. As we expected, it takes much less time for sub-file deletion in UFFS_BM because the proposed system exploits bmodify() function for handling block-aligned file modification. However, without bmodify() function, user-level file system cause additional overhead for dealing with sub-file modification compared to original TAR file approach on EXT4 file system.

In the final experiment, we measured the performance result of sub-file deletion based on changes in sub-file position. We selected 2nd, 5th, 9th sub-files from a TAR file that had total 10 sub-files. As anyone would assume, when we delete a sub-file from front section (2nd sub-file in this experiment), lots of blocks will be rewritten to the hard disk. Figure 8 shows this result on EXT4 file system. As close as possible to the front section, the sub-file deletion time is increased and the performance gap is proportional to the sub-file position. In UFFS approach, it takes very short processing time for deleting sub-files in a TAR file. Furthermore, it is always the same, regardless of the location of the sub-file.

4. Conclusion

In this paper, we propose an efficient user-level file system that supports block-aligned file modification. If a user wants to insert or delete several blocks into a file then we can avoid unnecessary file I/O by utilizing existing data blocks in the old file. To support block-aligned file operation, we provide additional file operation, bmodify() that supports file allocation table management capability. Furthermore, to show the impact of the proposed system, we applied block-aligned file modification scheme to TAR file format. Traditional file systems do not support efficient block-aligned file modification, therefore, sub-file deletion overhead will be increased as a sub-file is placed in front of TAR file. In this work, we provide a new user-level file system to support efficient file modification minimizing disk writing. Experiment results show the performance enhancement of UFFS compared to traditional file systems.

Acknowledgments

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2012R1A1A2044694), and this work also was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No.2011-0029848).

References

[1] J.I. Shim and J.D. Lee, “Improving disk I/O performance by using raw disk for web proxy servers,” Proc. 2nd International Conference on Human.Society@internet, HSI’03, Berlin, Heidelberg, pp.613–618, 2003.
[2] X. Zhang, Z. Huo, J. Ma, and D. Meng, “Exploiting data deduplication to accelerate live virtual machine migration,” Proc. 2010 IEEE International Conference on Cluster Computing, CLUSTER ’10, pp.88–96, Washington, DC, USA., 2010.
[3] A.T. Clements, I. Ahmad, M. Vilayannur, and J. Li, “Decentralized deduplication in san cluster file systems,” Proc. 2009 Conference on USENIX Annual Technical Conference, USENIX’09, p.8, Berkeley, CA, USA., 2009.
[4] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble, “Sparse indexing: Large scale, inline deduplication using sampling and locality,” Proc. 7th Conference on File and Storage Technologies, FAST ’09, pp.111–123, Berkeley, CA, USA., 2009.
[5] V.S. Amar More and Z. Shaikh, “Dext3: Block level inline deduplication for ext3 file system,” Proc. 2012 Linux Symposium, pp.89–94, Ottawa, Canada, 2012.