TempFFS: Enabling Delayed Allocation for Existing File Systems

Chih-Wen Hsiao and Ting-Chang Huang
Institute of Computer Science and Engineering
National Chiao Tung University
kenn@os.nctu.edu.tw,
tchuang@os.nctu.edu.tw

Da-Wei Chang
Department of Computer Science and Information Engineering
National Cheng Kung University
davidchang@csie.ncku.edu.tw

Abstract

Based on the previous observation that many files are short-lived, some advanced file systems delay the block allocation of their files until the file data needs to be written back to disk. Such delayed allocation feature can reduce the IO traffic as well as the degree of file fragmentation, and thus results a better IO performance. However, a file system can enjoy the benefit of delayed allocation only when it explicitly implements the feature.

In this paper, we design and implement a temporary-file file system called TempFFS to allow all the existing file systems to enjoy the benefit of delayed allocation. Based on the concept of stackable file system, TempFFS places all newly-created files in memory and transfers these files to their original file systems as needed. We implement TempFFS in the Linux kernel. Performance results show that, with the help of TempFFS, both the disk IO traffic and the degree of file fragmentation can be largely reduced, resulting a performance improvement of 34% to 69% under a file system benchmark, Postmark.

Keywords: File Systems, Delayed Allocation, File Fragmentation, Disk IO

1. Introduction

Disk performance has become the major bottleneck of most computer systems for a long time. Following the Moore's Law, the performance of processors improves by about 60% per year. However, the performance improvement of disks is only about 10%, resulting in an increasing performance gap between processors and disks.

To improve system performance, disk IO should be avoided as far as possible. For example, most operating systems use buffer cache to reduce the number of disk IO operations. If a data block is cached, disk access can be avoided. Moreover, when disk access is needed, it should be done efficiently. Specifically, the accessed blocks should be as contiguous as possible (i.e., less fragmented) in order to minimize the disk seek time, which dominates the disk access latency.

Delayed allocation [3] improves the system performance by reducing both the disk IO traffic and the degree of file fragmentation. The main idea of delayed allocation is to delay the block allocation of a newly-created file until the file data needs to be written to disk. According to the

previous studies [13][16][19], many files are short-lived, meaning that they are deleted soon after their creation. Allocating disk space for these files, which involves disk IO operations for reading the file system metadata (e.g. block allocation map), is unnecessary. Moreover, frequent creation/deletion of these short-lived files can cause a file system become fragmented, leading to IO performance degradation due to longer seek time. Some advanced file systems such as XFS [18], Resier4 [12], and Ext4 [7] support delayed allocation. The limitation of file system specific implementations is that each file system should explicitly implement delayed allocation so as to enjoy the benefit.

In this paper, we design and implement a Temporary-File File System (TempFFS) to apply delayed allocation simultaneously on all existing file systems. Different from file system specific implementations that maintain newly-created files on their own, TempFFS maintains newly-created files for all the file systems. Upon memory pressure or *sync* operations, the files are transferred to their original file systems and block allocation of these files takes place. Therefore, an existing file system can enjoy the benefit of delayed allocation without any code modifications. We have implemented TempFFS by modifying a RAM based file system in the Linux kernel. According to the performance evaluation, a traditional Linux file system, ext2, can have a performance improvement of 34% to 69% with the help of TempFFS.

The remainder of this paper is organized as follows. Section 2 describes the related work. Section 3 presents the design and implementation details of TempFFS. The performance results are shown in Section 4. Finally, we give conclusions in Section 5.

2. Related Work

In this Section, we describe some research efforts related to improving file system performance. To reduce disk IO operations, most file systems do not write updated data immediately to disk but cache the data in memory and propagate it to disk later. This is called *delayed write* [2][9]. Delayed write can group several write operations into a single disk IO operation. Moreover, if the delay is longer than the lifetime of a file, no further disk IO is needed. Rio [4] provides a persistent file cache so as to delay IO writes until memory pressure. Although delayed write can reduce disk IO traffic, they cannot reduce the degree of file fragmentation because a disk block is still allocated during the time when new file data is written to the file cache.

Some advanced file systems such as XFS [18], Resier4 [12] and Ext4 [7] support *delayed allocation* [3]. They delay the disk block allocation of a newly-created file until the data is needed to be flushed back to the disk due to memory pressure or *sync* operations. However, the delayed allocation feature is not shared among all file systems. Only the file systems that implement the feature can benefit from it.

Log-Structured file systems [8][14][15][17] treat a file system as a log and write data and metadata updates into the end of the log. File system changes are buffered in the cache and then written into the disk sequentially in single disk IO operation. Therefore, it can improve the performance of write operations by eliminating costly seek and rotation delays. File system performance can also be improved by using higher performance disks or disk arrays [10]. Moreover, using non-volatile as disk cache [1][5] has also

been investigated.

3. Design and Implementation

3.1 System Overview

As mentioned in the Introduction, instead of integrating the delayed allocation feature into a specific file system, we implement a RAM-based file system named TempFFS in order to apply the feature simultaneously on existing file systems such as ext3 and NTFS. Based on the concept of stackable file systems, TempFFS sits between Virtual File System (VFS) and file system implementations and is transparent to the latter. As shown in Figure 1, all new files are initially written to TempFFS and associated with their original file systems when they are created. TempFFS uses page cache as the file store, and the files are transferred, which is called *file transformation* in this paper, into their corresponding file systems upon memory pressure or *sync* operations. In this way, existing file systems can benefit from delayed allocation without code modifications.

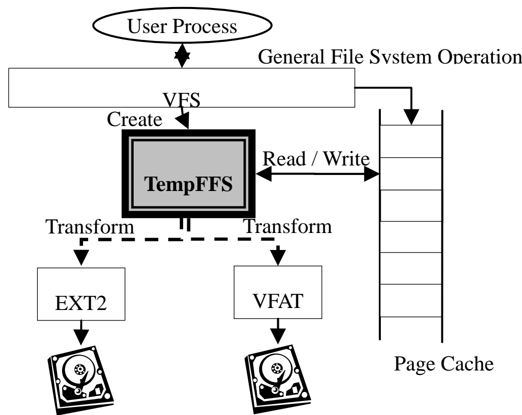

Figure1. Architecture of the TempFFS

3.2 File Transformation

TempFFS stores files in kernel memory, which cannot be paged out in traditional UNIX operating systems (including Linux). Upon memory pressure, an OS usually writes back the dirty pages that belong to the buffer cache or user processes to the storage device so as to release more memory space. In this situation, TempFFS checks if its size is larger than a specific threshold. If it is, TempFFS shrinks its size by evicting pages of the least recently used files. All the evicted files are transformed into their original file systems so that the corresponding data can be written back. In addition, we transform files whose sizes are larger than a specific threshold (currently, 1MB) due to the following two reasons. First, according to previous research [13][16][19], most short-lived files are small ones. Second, placing a huge file in TempFFS may cause the transformation of a large number of short-lived small files before they are deleted, reducing the benefit of delay allocation.

We manage the files in TempFFS in an LRU list. The number of pages that should be evicted from TempFFS, say N, is proportional to the number of pages in TempFFS. Specifically, N is calculated according to the following equation:

$$N = NR_WB * NR_TempFFS / NR_Dirty,$$

In the equation, *NR_WB* represents the target number of pages that need to be written back, which is determined by the operating system. *NR_TempFFS* represents the number of (dirty) pages in TempFFS, and *NR_Dirty* represents the number of dirty pages in the system. As shown in Figure 2, transforming a file involves the following three steps. First, the file create operation of the original file system is invoked to produce the metadata (inode) of the file. Second, several inode fields such as timing information, access rights and file size, are copied to the new inode. Third, a sequence of disk block allocation operations of the original file system are invoked for allocating the disk space for the file. In the figure, block 5, 6, and

7 are allocated, where block 5 is an indirect block. Because the operations are invoked consecutively, the resulting data blocks tend to be contiguous. After the allocation, the data is associated with the allocated blocks and the metadata in the TempFFS is deleted.

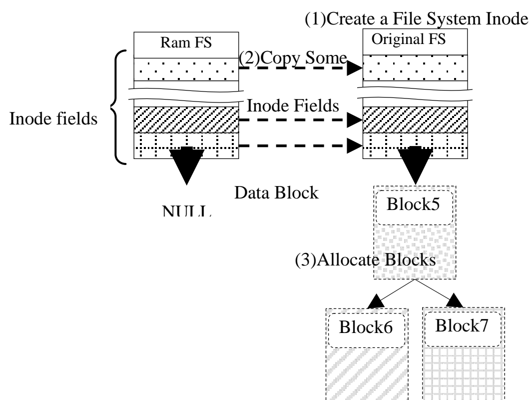

Figure 2. File Transformation

3.3 Implementation of TempFFS

For ease of implementation, the implementation of TempFFS was achieved by modifying the code of an existing RAM file system (i.e., the Resizable simple ram File System, RamFS [11]). As mentioned before, TempFFS sits between the VFS layer and the file system implementation layer. It intercepts the invocation of the VFS file creation function (i.e., *vfs_create()*), records the original file system for the created file, and directs the invocation to the file creation function of RamFS. After the creation, the file resides in RamFS and accesses to the file will through the file operations of RamFS.

Upon memory pressure, TempFFS transforms the LRU files into their corresponding file systems. In addition, when the size of a specific file is larger than a threshold, TempFFS also transforms the file into the corresponding file system. After the transformation, accesses to the files will through the file operations of the original file systems of the file.

4. Performance Evaluations

4.1 Experimental Environment

In this section, we present the performance improvements of TempFFS by comparing the performance of the ext2 file system with and without the presence of TempFFS. Table 4.1 shows the experimental environment. We evaluate the performance of TempFFS under a popular file system benchmark, Postmark [6], which is a macro-benchmark that emulates the access pattern of an email server. During the execution, Postmark first creates a number of files as the initial file set. Then, it performs a specific number of transactions, which include file creation, deletion, reading, and appending. Finally, it deletes all the files.

Table 1. Evaluation Environment

Hardware	**CPU**	AMD Athlon 64 3000+
	Memory	1 GB DDR 400
	Disk	Maxtor 80G 7200 RPM
Software	**OS**	Linux 2.6.12
	Workloads	Postmark 1.5

4.2 Performance Results

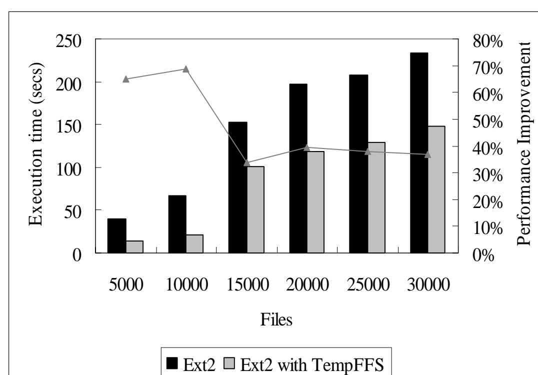

Figure 3. Performance Improvements with TempFFS

In this experiment, we measure the performance improvements of TempFFS under Postmark. In this experiment, 200k transactions were performed and the file size ranges from 512 bytes to 10 Kbytes. We measure the performance

under various numbers of files in the initial file set. As shown in Figure 3, TempFFS effectively improves the system performance. Specifically, the performance improvement ranges from 34% to 69%. This is because a number of files have been deleted before they are transformed to ext2, reducing both the I/O traffic and the degree of file fragmentation. We demonstrate this in the following experiments.

As mentioned before, Postmark deletes all the files at the end of its execution. Figure 4 shows the percentage of the number of files deleted in TempFFS and ext2, with the presence of TempFFS. As shown in the figure, at least 44% of the files are deleted in TempFFS. In the cases of 5000 and 10000 files, all files are deleted in TempFFS because the capacity of TempFFS is enough to contain all the files. When the number of files increases further, a number of files are transformed to ext2, because of memory pressure, and finally deleted in ext2. For each file deleted in TempFFS, all its file operations are done in memory and involve no disk IO.

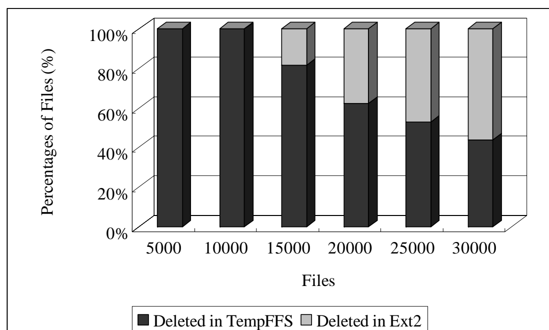

Figure 4. Percentages of Files Deleted in TempFFS

Figure 5 shows the reduction of IO traffic with the presence of TempFFS. In the cases of 5000 and 10000 files, nearly 100% of the IO traffic can be eliminated since almost all file operations are done in TempFFS. For the other cases, about 31% of the IO traffic can be eliminated.

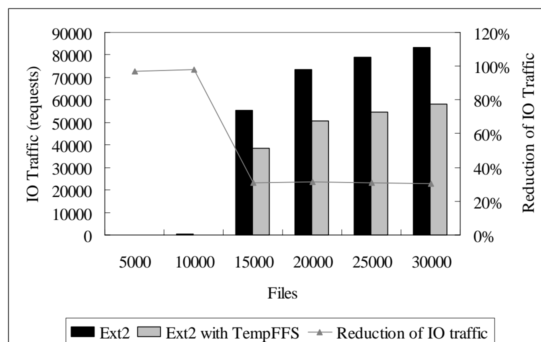

Figure 5. Reduction of IO Traffic with TempFFS

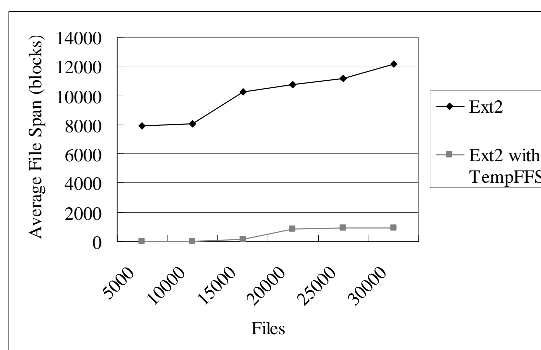

Figure 6. Average File Span

In the last experiment, we show that TempFFS can reduce the degree of file fragmentation, which is evaluated by using the *file span*, the distance between the first block and the last block of a file. During the execution of Postmark, we record the file span of each file upon the deletion of the file. Figure 6 shows the average file span of all the files. As shown in the figure, the degree of file fragmentation is largely reduced. Especially, in the cases of 5000 and 10000 files, the average file span is zero because all the files are deleted in TempFFS and do not have corresponding blocks on the disk.

5. Conclusions

Delayed allocation can reduce both disk IO traffic and degree of file fragmentation. In this paper, we design and implement TempFFS so as to allow existing file systems to enjoy the benefit of

delayed allocation.

Based on the concept of stackable file system, TempFFS sits between VFS and file system implementations. It intercepts VFS file creation operation so as to place the data of newly-created files temporarily in memory. Files are transformed into their original file systems when needed (i.e., memory pressure, sync, or exceeding file size limitation). We implemented TempFFS by modifying a RAM based file system in the Linux kernel. Performance results show that, an unmodified ext2 file system can have a performance improvement of 34% to 69% with the help of TempFFS.

References

[1] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer, "Non-Volatile Memory for Fast, Reliable File Systems", *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operation Systems*, October 1992

[2] P. M. Chen, "Optimizing Delay in Delayed-Write File Systems", *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, October 1994.

[3] M. Cao, T. Y. Tso, B. Pulavarty, S. Bhattacharya, "State of the Art: Where We Are with the Ext3 Filesystem", *Proceedings of 2005 Linux Symposium*, July 2005.

[4] P. Chen, W. T. Ng, G. Rajamani, C. M. Aycock and D. Lowell "The Rio File Cache: Surviving Operating System Crashes", *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 74-83, October 1996.

[5] J. W. Hsieh, T. W. Kuo, P. L. Wu, and Y. C. Huang, "Energy-Efficient and Performance-Enhanced Disks Using Flash-Memory Cache", *The International Symposium on Low Power Electronics and Design*, August 2007.

[6] J. Katcher, "Postmark: A New File System Benchmark", *Technical Report TR3022 Network Appliance Inc*, October 1997.

[7] A. Mathur, M. Cao and S. Bhattacharya, "The New Ext4 File System: Current Status and Future Plans", *Proceedings of the Linux Symposium*, June 2007.

[8] J. Matthews, D. Roselli, A. Costello, R. Wang, and T. Anderson, "Improving the Performance of Log-Structured File Systems with Adaptive Methods", *Proceedings Sixteenth ACM Symposium on Operating System Principles*, October 1997.

[9] D. A. Muntz, P. Honeyman and C. J. Antonelli, "Evaluating Delayed Write in a Multilevel Caching File System", *Proceedings of the IFIP/IEEE International Conference*, pp. 415-429, June 1996.

[10] D. A. Patterson, G. A. Gibson, and R. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 109-116, June 1988.

[11] "Ramfs", http://lwn.net/Articles/156098/.

[12] H. Reiser, "Reiser4 Transaction Design Document", *Technical Report, Namesys*, 2002.

[13] D. Roselli, J. Lorch, and T. Anderson, "A Comparison of File System Workloads", *Proceedings of the USENIX Annual Technical Conference*, pp. 41–54, June 2000.

[14] M. Rosenblum, and J. Ousterhout, "The LFS Storage Manager", *Proceedings of the Summer 1990 USENIX Technical Conference*, June 1990.

[15] M. Rosenblum, and J. Ousterhout, "The Design and Implementation of a Log-Structured File System", *ACM Transactions on Computer Systems,* Vol. 10, No. 1, pp. 26–52, February 1992.

[16] C. Ruemmler and J. Wilkes, "UNIX Disk Access Patterns", *Proceedings of the Winter 1993 USENIX Conference*, pp. 405–20, January 1993.

[17] M. Seltzer, K. Bostic, M. McKusick, and C. Staelin, "An Implementation of a Log-Structured File System for UNIX", *Proceedings of the 1993 USENIX Winter Technical Conference*, pp. 307-326, January 1993.

[18] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS File System", *Proceedings of the USENIX 1996Annual Technical Conference*, January 1996.

[19] W. Vogels, "File System Usage in Windows NT 4.0", *Proceedings of the 17th Symposium on Operating Systems Principles*, pp.93-109, December 1999.