DFS: A De-fragmented File System

Woo Hyun Ahn, Kyungbaek Kim, Yongjin Choi and Daeyeon Park
Department of Electrical Engineering and Computer Science
Korea Advanced Institute of Science and Technology, Daejeon, Korea
{whahn, kbkim, yjchoi}@sslab.kaist.ac.kr, daeyeon@ee.kaist.ac.kr

Abstract

Small file accesses are still limited by disk head movement on modern disk drives with the high disk bandwidth. Small file performance can be improved by grouping and clustering, each of which places multiple files in a directory and places blocks of the same file on disks contiguously. These schemes make it possible for file systems to use large data transfers in accessing small files, reducing disk accesses. However, as file systems become aged, disks become too fragmented to support the grouping and clustering of small files. This fragmentation makes it difficult for file systems to take advantage of large data transfers, increasing disk I/Os. To offer a solution to this problem, we describe a De-fragmented File System (DFS). By using data cached in memory, DFS relocates and clusters data blocks of small fragmented files in a dynamic manner. Besides, DFS clusters related small files in the same directory at contiguous disk locations. Measurements of DFS implementation show that the techniques alleviate file fragmentation significantly and, in particular, performance for small file reads exceeds that of a traditional file system by 78%.

1 Introduction

In modern disk drives, the media bandwidth has been improved significantly to such an extent that its peak value is 40 MB/second. Unfortunately, small file (< 64 KB) accesses, the majority of file system usages [1][13], are limited by disk access latency (seek and rotational delay) rather than disk bandwidth since file systems spend a large amount of time in waiting for the disk head to reach its destination during small file accesses [7][3].

The Fast File System (FFS) [5] uses clustering [6][10] to increase performance for small file I/Os, It attempts to place logically sequential blocks of a file on physically contiguous disk blocks. When a new block is allocated, FFS determines the location of the last allocated block of its file and attempts to allocate the next contiguous disk block. When blocks of a file are clustered, multiple block transfers can be used to read/write the file, reducing separate disk I/Os and amortizing disk access latency. In particular, performance of small file accesses, which are sensitively affected by the number of disk I/Os, is enhanced by multiple block transfers if blocks of each small file are clustered. Besides, FFS divides the disk into cylinder groups, each of which is a set of consecutive cylinders. Cylinder groups are used to exploit locality: related data (e.g, files in a directory and blocks in a file) are located in the same cylinder to reduce disk seeks. Thus, FFS allocates logically sequential blocks of a file in the same cylinder group, and likewise allocates all of the files in a directory to the same cylinder group.

In the Co-located Fast File System (C-FFS) [3], small files in the same directory are allocated to contiguous disk locations, where they form a group. When a new block of a small file is allocated, C-FFS attempts to cluster it into an existing group associated with the same directory as that of the block. This succeeds if there is a free block outside of the group the size of which does not exceed a certain maximum size (e.g., 64 KB); otherwise, the new block cannot be added to the group and C-FFS creates a new group with one block. When a block is read, C-FFS finds the group holding the block and then reads all blocks in the group at one time by using the high disk bandwidth of modern disk drives. Then subsequent accesses of the prefetched files can be satisfied in memory without disk I/Os. Hence, C-FFS improves small file performance by an order of magnitude over traditional file systems in experiments on several potential applications.

A significant challenge for FFS and C-FFS is fragmentation of free and allocated space. As file systems become aged by many file creates/deletes, FFS cannot find clusters of free blocks in order to allocate blocks of a newly created file on the disk contiguously. This fragmentation of a file is defined as *IntrA-file Fragmentation* (IAF). Moreover, file system aging has a negative impact on explicit grouping of C-FFS. As files become created/deleted, some files in a group that has multiple files placed adjacently on the disk can be deleted, and thus fragmented free space is produced

1526-7539/02 $17.00 © 2002 IEEE

among the explicitly grouped files. This separation (or fragmentation) among the related files is defined as *IntEr-file Fragmentation* (IEF), reducing the number of files that can be fetched from disks at one time. To sum up, IAF and IEF prevent file systems from exploiting large data transfers, increasing the number of disk I/Os and large disk access latency. Specifically, the increase of disk I/Os seriously degrades small file performance. All data that suffer from either IEF or IAF can be called *fragmented data*.

In this paper, we propose a new file system called the *De-fragmented file system* (DFS), which relocates fragmented data in a dynamic manner so that the fragmented data can be placed contiguously on the disk. More specifically, DFS concentrates on the relocation of small files among the fragmented data to increase performance of small file reads. As related data in a directory become accessed together within a short period of time, they can be together contained in the file cache (or buffer cache). Accessing data on much fragmented disks causes fragmented data to be cached in memory. By using the fragmented data in the cache, DFS attempts to perform two techniques as follows: first, blocks within a small file are relocated in order to be physically contiguous with each other on the disk where FFS cannot cluster them due to a shortage of clusters of free blocks. Note that no additional disk access is required for fetching data from the disk since DFS relocates data cached in memory only. Second, DFS dynamically clusters small files that have not been adjacently placed with other related files on the disks. Hence, DFS alleviates fragmentation of both multiple files and blocks within the same file.

We implement DFS as a module of FFS in an OpenBSD operating system to measure its potential effectiveness. The rest of the paper is organized as follows: Section 2 and Section 3 explain the backgrounds and main ideas of our DFS respectively. Section 4 discusses some emergent issues for DFS implementation on an OpenBSD operating system. Section 5 presents our experimental methodology and results. Finally, we conclude in Section 6.

2 Related Works

The de-fragmentation concept of DFS resembles tools (e.g., disk reorganizer) on the Windows operating systems that de-fragment disks upon user commands. However, the disk reorganizer is different from DFS in the following points: first, the reorganizer should be executed during idle time when users do not use their computers. Otherwise, the overall performance of computer systems is degraded very seriously. Second, the reorganizer does not attempt to keep related files in the same directory together. It greedily reorganizes only fragmented files and free space without consideration of relationships among data in the same directory. Third, many of computer illiterates may not use the reorganizer because they do not recognize the necessity of the de-fragmentation. However, DFS implicitly reorganizes fragmented data regardless of user expertism.

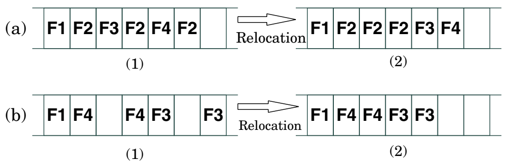


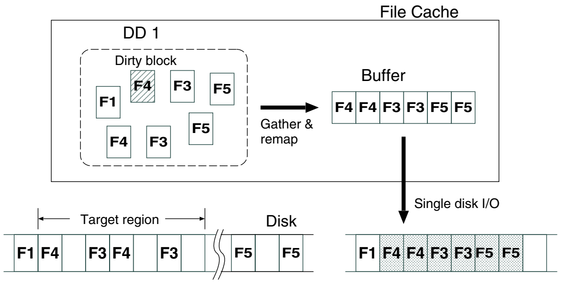
Figure 1. Two types of relocatable fragmented data. A file F_x is composed of blocks marked by F_x. For example, a file 'F2' has three blocks, as shown in Figure(a).

The log-structured File system (LFS) [8][9] increases synchronous write performance of all file system data. LFS re-maps all modified blocks into large, contiguous regions called *segments* on disks. Each segment is then written to disks with a single I/O operation. LFS is similar to DFS's mechanism in that data are gathered, re-mapped and stored at a single disk I/O, but it is different from DFS in that the goal of DFS is to reduce fragmentation of disk layouts and improve file read performance. However, there are some important problems with LFS. First, LFS performs a garbage collection process called *cleaning* to reserve empty segments for new modified data. However, the cleaning can significantly degrade file system performance in a busy system [9][10]. Second, LFS often fails to keep the contents of a directory together. For example, if all files in the directory are actively read but only some are actively written, the written files will move far away from the read-only ones. In such disk layout, the penalty of fetching the files from disk is much higher than in FFS and C-FFS.

3 A De-fragmented File System

3.1 Motivations

DFS is motivated by the fact that FFS and C-FFS take a kind of update-in-place approaches; once a new block has been placed in a given disk location, it does not move and all subsequent read/update requests for the block will be sent to that location. Unfortunately, once these file systems become fragmented, the performance penalty described above persists. Owing to this characteristic, FFS and C-FFS fail to make good use of the following opportunity; fragmented data that are in memory can be moved in a dynamic manner to reduce both IEF and IAF. Most operating systems implement file caching, which contains frequently accessed data in memory to avoid unnecessary disk accesses. As multiple files in the same directory are accessed within a short period of time due to the directory locality, most of the files can be cached in memory. Among data in the cache, there can be an amount of fragmented data, which traditional file



(a) Initial disk layout (b) Disk layout after grouping of DFS

Figure 2. Overview of DFS's de-fragmentation

systems have not dealt with efficiently enough to reduce the fragmentation. Hence, it is necessary to perform an efficient in-memory block management that can relocate disk locations of the fragmented data to solve both IEF and IAF. Note that the *relocatable fragmented data* should be cached in memory because the relocation of uncached data requires additional disk I/O to fetch them into memory.

Relocatable fragmented data are classified into two types where it is necessary to use relocation schemes appropriately, according to whether fragmentation of the data is incurred by related data or free space on disks. The first type has the following characteristic of the disk layout as shown in Figure 1(a)-(1): blocks of a file are not physically contiguous with each other due to interference with blocks of other related files that are currently cached in memory. Figure 1(a)-(2) illustrates how an efficient relocation scheme solves the IAF problem. This mechanism relocates not only blocks of the fragmented file, but also the neighboring data (see F_3 and F_4 in Figure 1(a)-(1)) that incur fragmentation of the file, placing blocks of the fragmented files at contiguous disk locations.

The characteristic of the second type is that blocks of fragmented data (or related files) are separated from each other by free blocks on disks (see Figure 1(b)-(1)). Figure 1(b)-(2) shows how careful relocation of the blocks can eliminate the IEF. When the second block of F_4 and two blocks of F_3 in Figure 1(b)-(1) are moved toward the first of F_4, the related files are grouped contiguously on the disk and, thus, their IEF problems can be solved.

3.2 Basic Concepts

DFS introduces two techniques as a general relocation framework to reduce fragmentation of small files[1] (2 blocks – 12 blocks): IntrA file De-fragmentation (IAD) and IntEr file De-fragmentation (IED) mechanisms. By using the high disk bandwidth of modern disk drives, they solve IAF and IEF problems of the two types of *relocatable fragmented data* in a dynamic manner. Whenever each dirty (or modified) block in the file cache using write back is flushed to the disk, DFS checks whether a fixed window called the *target region*, which starts from the disk location of the dirty block, has any type of relocatable fragmented data. If so, DFS applies IAD and IED to the fragmented data and, then, the data relocated by these techniques are written together with the dirty block at a single disk I/O. Otherwise, only the dirty block is flushed as with traditional file systems.

Figure 2 shows how DFS's IAD and IED reduce fragmentation of files and free space. When target region determined by the position of a dirty block has IAF or IEF, DFS's de-fragmentation mechanisms are triggered. First of all, *relocatable* fragmented data, which include all data not only cached in memory but also contained in the same directory as the dirty block, are gathered into a buffer, which is an entry of the file cache. Note that the dirty block is the beginning of the buffer and then the fragmented data follow the dirty block in the buffer. Once the buffer gathering is completed, IAD and IED are applied to the fragmented data of the buffer in the following steps: (1) if there are any files (e.g., F_3 and F_4) with IAF, IAD clusters blocks of each file on the disk contiguously; (2) if there are any related files (e.g., F_3 and F_4) that are not contiguous with each other due to interference with free space, IED re-maps their disk locations to simply cluster them contiguously and compact the free space contiguously; (3) if the target region has contiguous free space either that is compacted by IED or that has existed before the de-fragmentation, the free space is used to relocate small fragmented files (F_5) not included in the target region. Finally, the relocating data contained in the buffer are stored to the original location of the dirty block at a single I/O. Despite these large data transfers, the incremental disk overhead is fairly small because accessing several blocks rather than just one requires a fairly small additional disk overhead in modern disk drives.

DFS is different from LFS in the following characteristics: DFS overwrites relocating data to the starting location of the target region involving the data, as shown in Figure 2. On the other hand, LFS always stores relocating data to free segments; otherwise, the relocating data might be overwritten to a disk region with valid data, which could be lost by the overwriting. In DFS, however, relocating data can be overwritten to the original locations without the loss of data. This is possible because the relocating data (F_3 – F_5) involve blocks (F_3 and F_4) on disk locations where the relocating data will be stored. Hence, DFS does not need the cleaning operation unlike LFS. If some of the blocks in the target region are not cached in memory, they cannot be contained in the relocating data; this will be described in Section 3.4.

In order for related data cached in memory to be relocated at contiguous disk locations, it is necessary to estab-

[1] Note that K. Smith and M. Seltzer [12] defines files less than 13 blocks as small files because accessing files with the sizes considers to be affected by disk head movement dominantly.

lish in-memory relations among related data at a directory unit. DFS defines a *De-fragmentation Domain* (DD), which presents a set of files that are contained not only in the same directory name space locality, but also in the cache. Each DD object is created in memory when any block of its directory is accessed for the first time. Whenever each data block is loaded into memory, the information of the block is registered at its DD object. The information of each block includes its in-memory address, its logical and physical block numbers, the in-memory inode pointer of the file involving the block. When DFS relocates fragmented data, the information is used to find which DD contains a data block with a specified disk location. Besides, each DD object manages a table called the *de-fragmentation table*, which contains only fragmented files among files cached in memory.

The maximum number of blocks that a buffer can gather for relocating is the same as that of a target region. However, if only some blocks in a target region that will be de-fragmented are clustered, the buffer size required for the clustering becomes less than the target region size. Though the target region has fewer blocks than its size, DFS always tries to gather and re-map as many blocks as a target region can maximally contain, writing the chunk to the disk. For example, there are four allocated blocks in the target region (see Figure 2), which has seven blocks including free space and data. To relocate as many blocks as possible during a single disk I/O, DFS additionally gathers and re-maps blocks of a fragmented file F_5 outside the target region.

3.3 De-fragmentation Decision Methodology

Whenever each dirty block is stored to the disk, DFS decides whether its target region should be de-fragmented or not. If data in the target region have IEF and IAF problems, DFS first considers it as a good candidate. For another candidate, DFS selects a target region with two or more free blocks [2] regardless of whether the target region has IEF and IAF. The free blocks are used to relocate and cluster blocks of small files (e.g., see F_5 in Figure 2) that not only exist outside the target region, but also have IAF. At this time, if the free blocks are fragmented, DFS makes them contiguous with each other in order to cluster blocks of the fragmented files at contiguous disk locations. For example, though the target region in Figure 2(a) has three free blocks, DFS contiguously compacts the fragmented free space through IED to relocate the fragmented file into the free space.

To find the amount of free space and IEF contained in each target region, DFS uses a block allocation bitmap, which file systems [5][2] use to check whether blocks on the disk are allocated. From the beginning of the location indicated by a dirty block, DFS checks as many bits as the number of blocks of the target region. Assume that bit value '1' represents the allocated state. If there are two or more '0's in the bits, DFS considers the target region as a candidate that should be de-fragmented because the situation presents the following two possibilities: first, data in the target are not placed adjacently due to interference with fragmented free space. Second, the target region has at least two free blocks that can be used to relocate any fragmented files outside the target region.

DFS maintains a *block fragmentation bitmap* with one bit per block to quickly find whether each target region has files with IAF. The bitmap only presents fragmentation of files cached in memory currently, not all files on the disk. By using the bitmap, DFS checks whether a block cached in memory is not physically contiguous with the next logical block of the file including the former block. The bit setting is performed whenever each block is loaded into memory from the disk. Each bit in the bitmap is set to value '1' if its corresponding block is not physically contiguous with the next logical block, which should be cached in memory. On the other hand, the value '0' indicates that either the block is not cached in memory currently, or is contiguous with its next logical block. If the bits corresponding to a target region have one or more '1's, DFS predicts that the target region has at least one fragmented file and selects it as a good candidate for de-fragmentation.

3.4 De-fragmentation Mechanisms

The de-fragmentation algorithm of DFS is invoked when the target region starting from a dirty block seems to be less than optimal according to the decision methodology in Section 3.3. As the first step of the algorithm, blocks within a target region are simply gathered in a buffer so as to be stored to the disk with a single disk I/O. For the buffer gathering, a buffer of the same size as that of the target region is assigned by the file cache manager. Then, DFS first puts the dirty block at the head of the buffer because lower-level disk drivers store data to the location indicated by the head block in a buffer. From the beginning of the next disk location, DFS looks up each disk block of the target region one by one to gather only blocks that are not only allocated on the disk, but also included in the same DD as that of the dirty block.

As seen in Figure 3(a), DFS gathers all data blocks of a target region into a buffer because they are included in the same DD object. IAD and IED are successively applied to the blocks gathered in the buffer to eliminate fragmentation. IAD first re-maps the blocks to place logically sequential blocks of each fragmented file on physically contiguous disk blocks. Additionally, since the file F_4 are related to the other files (F_1 – F_3), IED clusters it together with the

[2] Only files with at least two blocks can have IAF problem. A target region needs at least two free blocks so that one two block-sized file outside a target region can be relocated into it.

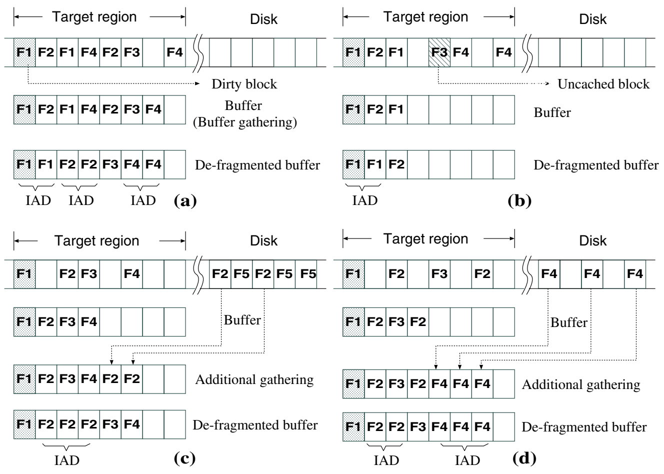


Figure 3. De-fragmentation mechanisms of DFS

related files. Then, the data in the buffer is written to the starting location of the target region at a single disk I/O.

Figure 3(b) shows how blocks not cached in memory have an impact on the buffer gathering. When one uncached block F_3 is encountered in process of the buffer gathering, the process is terminated and, then, the buffer contains only blocks that have been gathered currently, not including the uncached block. In this case, the number of blocks in the buffer is less than that of target region. DFS applies IAD and IED to the blocks gathered in the buffer in a manner similar to that shown in Figure 3(a). The reason that an uncached block should incur the termination is noteworthy. First, additional disk I/Os for fetching them into memory are required to gather uncached blocks. Second, no consideration of uncached blocks incurs their loss on the disk. For example, we assume that DFS sequentially gathers and remaps only five blocks (i.e., blocks of file F_1, F_2 and F_4) exclusive of the uncached block, storing the relocating data to the staring location of the target region. Unfortunately, one block (i.e., the second block of F_4) among the re-mapped blocks can have the same disk location as that of the uncached block. This overwriting incurs the loss of the uncached block. To prevent uncached blocks from being lost, the buffer gathering process is directly terminated when any uncached block is encountered.

Figure 3(c) shows how DFS relocates a file F_2 that has its blocks in the inside and outside of a target region. First of all, DFS gathers all blocks of a target region into a buffer in the same way as that of Figure 3(a). After this buffer gathering, DFS finds one file whose blocks exist in the inside and outside of the target region. If the buffer has enough free space to relocate the rest of the blocks outside the target region, they are gathered into the buffer additionally. These procedures are performed repeatedly for other files in the same DD as long as the buffer still has sufficient free space. After the additional buffer gathering, DFS re-maps all blocks in the buffer, storing to disks at a single disk I/O.

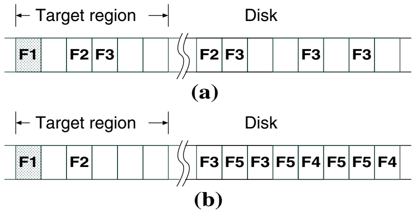


Figure 4. Optimal file selection of DFS

If a buffer has available free space after the buffer gathering, DFS additionally gathers small fragmented files outside the target region, as illustrated in Figure 3(d). For the candidate files that can be gathered into the buffer, DFS preferentially selects the smallest of the fragmented files (e.g., F_4) that exist outside the target region. These candidate files can be found from the current DD object (this mechanism will be explained in Section 4). After this additional buffer gathering, IAD and IED are applied to all data that have now been gathered in the buffer.

3.5 Methodology of Optimal File Selection

As shown in Figure 4, DFS should determine which files with IAF are relocated into a target region involving limited free space so as to increase the utilization of free space and small file access performance. As the basic file selection policy, DFS preferentially selects the smallest files because the performance of small file reads is more dominantly affected by separate disk I/Os. As shown in Figure 4(a), for example, each of file F_2 and F_3 in the target region has its blocks in the inside and outside of the target region, but all outside blocks of the files cannot be moved into the target region because the target region has fewer free blocks than the total number of the blocks to be relocated. According to the policy, DFS only relocates the second block of the smaller file, F_2, into the target region. However, the remaining blocks (the second and third blocks) of F_3 are not relocated because the target region does not have enough free space to include them after the relocation of F_2. If the two blocks of F_3 were relocated, the file would suffer from IAF again because the last block would be separated from the others. Additionally, the selection policy is applied when DFS selects several files outside a target region in Figure 4(b). Hence, only two block-sized files (F_3 and F_4) are relocated into the target region.

4 Implementation Issues

DFS techniques are implemented as a module of FFS on an OpenBSD operating system. This section describes the various issues that should be considered.

DD functionality required for small fragmented files

When a fragmented file with a specific size is needed to be relocated into a target region not including the file, DFS uses the de-fragmentation table to find it in a DD object. In FFS, a on-disk inode includes pointers to where the actual data blocks are stored. The pointers are classified into direct (12 entries) pointers for small files with 1 to 12 blocks and indirect pointers for files larger than 12 blocks. When a file is accessed, its on-disk inode is fetched into memory. This inode cached in memory is called *in-core inode*. When a file being fetched from disks is small as well as fragmented, its in-core inode is registered at the de-fragmentation table in the DD including the file. Note that the table has a hash chain of 11 entries indexed by file sizes because DFS is interested in optimizing small files of 2 to 12 blocks only. For example, when a fragmented file with 4 blocks is fetched into memory, its in-core inode is inserted into the hash entry indicated by its total number of blocks (e.g., 4) and linked with the others at the entry.

Clustering appropriate to a read-ahead algorithm

OpenBSD's FFS implements a history-based read-ahead (or prefetching) algorithm when reading files sequentially. The system maintains a "sequential count" of the last run of sequentially accessed blocks (if the last four accesses were for blocks 0, 2, 3 and 4, the sequential count is 2). When FFS concludes that the last accesses are sequential, it issues a new read-ahead of length l beginning with the first non-cached block, where l is the maximum of (a) powers of sequential count, (b) the number of contiguously allocated blocks remaining in the current clusters or (c) number '1'. For example, assume that a 4 block-sized file whose blocks are clustered altogether on the disk is accessed sequentially. When the second block is accessed, FFS decides that the sequential count is 1 and thus l is 2. Then FFS prefetches the other two blocks in a single disk read. However, if the file has 3 blocks, FFS only prefetches the last block though it can read two blocks at one time. For this file, FFS will perform as many disk accesses as the number of blocks of the file. Hence, multiple block transfers can be exploited to access only files larger than 3 blocks.

To increase efficiency of this prefetching, DFS selects fragmented files with 4 blocks as the smallest files that can be relocated inward into a target region (see Figure 3d). If a 4 block-sized file is available, DFS relocates it into the target region. Otherwise, DFS checks whether the hash of its DD object has the incremental block-sized files. DFS proceeds with this operation until either the checking arrives at the hash entry with 12 block-sized files or DFS has completely filled the free space with fragmented files. Still, if there is any free space in the target region, DFS additionally fills the free space with 2 or 3 block-sized files that are not only fragmented but also outside the target region. This makes it possible for a file with 2 or 3 blocks to be accessed with small seek and rotational latency.

Target region sizes and their effects on write performance

Target region sizes should be carefully determined so that multiple block writes do not have a negative effect on the overall performance of applications and flushing operations. DFS basically uses the target region size of 64 KB (or 16 blocks if one block size is 4 KB) for the following two reasons: first, a 64 KB access time is nearly the same as that of 4 KB in our experimental disk drive which was released in 1997. Second, a single write has been limited up to 64 KB in size because of limitations of computer hardware components (e.g., I/O bus and disk drive). Based on these facts, 64 KB is used as the target region size for DFS techniques. Additionally, we adopt 96 KB (or 24 blocks) as another target region size because writes of bulk data larger than 64 KB are expected not to incur significant overhead in modern disk drives such as the Quantum Atlas and the Seagate Cheetah series larger than 10,000 RPM.

DFS's techniques themselves amortize the disk overhead incurred by the writes of multiple blocks. In traditional file systems, dirty blocks in file caches are flushed to disks at separate disk I/Os, which reduce write performance during flushing operations. On the other hand, it is probable that DFS will gather several dirty blocks into a buffer when a target region is de-fragmented. Then the data in the buffer are stored at a single disk I/O, not at separate disk I/Os though the entire buffer contains several dirty blocks. Hence, DFS can write all of the modified data with fewer disk I/Os than that of traditional file systems, amortizing disk access latency added by large data writes.

Updating inodes on disks

DFS's techniques require modification of on-disk inodes associated with relocating data, increasing separate disk I/Os. As blocks of a small file are re-mapped, direct pointers in its inode should be modified to indicate the new locations. In OpenBSD's FFS, a metadata block for inodes has a 4 KB or an 8 KB size, containing 32 or 64 inodes of 128 Byte size respectively. When an inode is modified by applications, the whole of the block including the inode is stored to the disk because FFS writes metadata as well as file data at a block unit. In consideration of this feature, FFS locates inodes of related files in the same directory close to each other on the disk. This placement makes it possible for the related inodes to be contained in the same metadata block, reducing separate disk writes required to update modified inodes at the disk. In DFS based on FFS, inodes that include pointers changed under relocation can be contained in the same metadata block as those updated by applications. Therefore, updating the changed inodes on disks requires fairly small

Disk Parameters		File System Parameters	
Total Disk Space	6.4 GB	Size	2 GB
Rotational Speed	5400RPM	Block Size	4 KB
Sector Size	512 Bytes	Rotational Gap	0
Cylinders	13328	Cylinder Groups	283
Heads	15	Sectors per Track	63
Average Seek	9.5ms		

Table 1. Testing system configuration

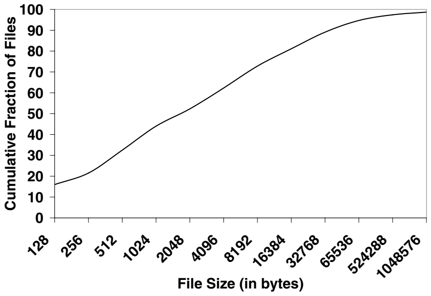


Figure 5. Distribution of file sizes

additional disk I/Os.

5 Performance Evaluation

This section reports measurements of our DFS implementation to show that it can dramatically improve small file read performance.

5.1 Experimental Setup

All experiments were performed on a PC with a 700 MHz Pentium processor and 128 MB of main memory. The disk on the system is a Quantum Fireball EX [4]. More parameters, along with our hardware and file system configuration, are summarized in Table 1. We used the OpenBSD operating system version 2.8 for all measurements in this paper.

5.2 Benchmark

For experiments, we wanted to make the testing disk as much fragmented as the file systems shown in the previous research [11]. To age the disk, we ran a file system benchmark on an empty disk. The aging benchmark executes sequences of file system operations, and, in particular, file creates and deletes are actively applied to our testing file systems to simulate the effects of a long period of use. Note that each create operation performs not only creation of a file, but also writing of data. A kind of issued requests, one of creates, read/writes and deletes, is determined by the probability distribution shown in Table 2, where the ratio of file operations was modeled according to previous study [1][13]. To determine file sizes of the issued requests, we scan our laboratory's file server. The scanned Linux file server supplies 30 GB storage for file systems. At the time of the examination, about 24 GB (80% of the total available) was being used for storage. Figure 5 shows that most files are small and the distribution is similar to that found in previous research [1][3]. To ensure that the benchmark has data access patterns matching directory namespace locality, the benchmark makes 50 sub-directories, each of which is selected by a Poisson distribution. The benchmark selects a random number within the range of 100 – 5000 to determine the number of file operations that will be performed in each sub-directory. When the file activities of a directory are completed, those of the next directory are started.

File requests	Before 75% utilization	After 75% utilization
Create	10%	10%
Delete	5%	10%
Read	65%	60%
Write	20%	20%

Table 2. Ratio of file operations according to the utilization of disk

To age the testing disk in a manner similar to that which occurs with real file system usages causing heavy fragmentation of disks, the benchmark increases the number of file deletes in progress of its execution as shown in Table 2. In real file system usages, users fill up their empty disks slowly, but usually do not clean files on the disks until the disk space is full extremely. When the disks become extremely full, users remove many files to clean up disk space and reserve free space at once. This cleaning causes the disks to be significantly fragmented. After the first cleaning, users again create many files and delete just as many files to reserve free space on occasion. These repeated creates/deletes make the disks more fragmented. To simulate this fragmentation process, the benchmark issues a much smaller number of delete requests than create requests to an empty disk to incur fragmentation slowly. However, we changed the ratio of file creates and deletes, making the disk much more significantly fragmented when the disk space became extremely full. We considered the disks to be considered extremely full when the utilization of the disks reaches 75%.

To simulate realistic access patterns and resource loads on our file systems, the benchmark issues 200 requests per second as in a previous study [13]. According to the study, many applications macroscopically issue requests at a constant interval of time. In real worlds, modified data during the file operations are stored to the disk by a sync (or flushing) daemon invoked every 30 seconds. However, if the benchmark continuously issued requests of file operations without the intervals of downtime, the testing file systems

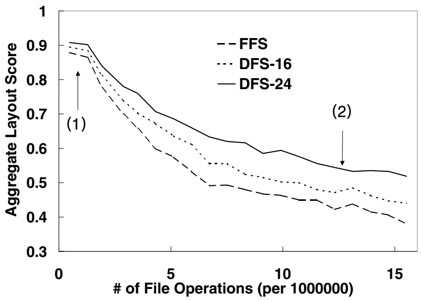


Figure 6. Aggregate disk layout scores according to the number of file operations

would be different from real file systems in loads on resources, behaviors of the file cache (e.g., data replacement and flushing operations), and the amount of data that will be flushed in every period. To minimize these differences, our benchmark issues the requests at a fixed interval of time.

5.3 Aggregate Disk Layout Scores

To verify improvement of DFS over FFS, we compared the degree of file fragmentation of DFS with that of FFS. As mentioned in Section 4, we implemented two DFSs each of which uses 16 (DFS-16) and 24 (DFS-24) blocks for its target region size, and then measured the effectiveness of DFS's IED and IAD according to the target region sizes. To quantify the amount of fragmentation of each file, we exploit a *layout score* used in earlier research [12]. The layout score for an individual file is the ratio of the block that is physically contiguous with the previous block of the same file. A file with a layout score of 1.00 is perfectly allocated; all of its blocks are allocated contiguously. A file with a layout score of 0.00 has no contiguously allocated blocks. To evaluate the fragmentation of all files on a file system, we compute the file system's "aggregate layout" score.

Figure 6 shows the aggregate layout scores of DFS-16, DFS-24 and FFS according to the number of file operations. The x-axis is the number of file operations issued by the benchmark. The aggregate layout scores are shown from the moment when 75% disk utilization is achieved. With a small number of file operations, the disks become slowly fragmented. However, as file operations are executed for a long period of time, the difference in disk layout score among the three file systems increases. The amount of fragmentation of free space and files increases in proportion to the number of file operations that have be executed. In such disk layout, both DFS-16 and DFS-24 actively relocate and cluster small files with IAF. This de-fragmentation makes both DFS-16 and DFS-24 suffer from less fragmentation than does FFS. Hence, each layout score of DFS-16 and DFS-24 outperforms that of FFS by 13% and 29%. Morever, DFS-16 and DFS-24 focus both IAD and IED on small files that are fragmented, but leave large files fragmented on the disks. This unconcern toward large files causes the layout scores of the disks to decrease in proportion to file operations.

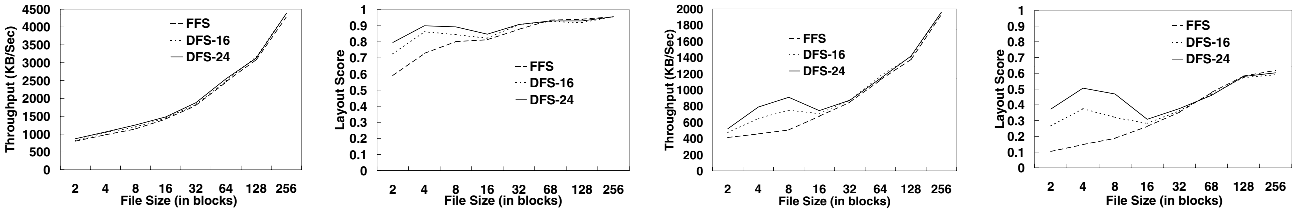
On the whole, DFS-24 outperforms DFS-16 in the layout score because larger target regions can relocate and cluster more files with IAF. On fragmented disks, the number of fragmented files that can be contained in a target region increases in proportion to the target region size. Also, the target region of DFS-24 can have more free blocks than that of DFS-16. With this larger free space, DFS-24 can relocate more files outside the target region. Thus, the effect of large target region makes it possible for DFS-24 to relocate more files at a single write than DFS-16 can.

5.4 Read Performance

We examined how file fragmentation reduced by DFS's techniques, i.e., IAD and IED, has a positive affect on file read performance. To examine this, we measured file read performance and aggregate layout score for files of a variety of sizes. These measurements were achieved at the points indicated by (1) and (2) in Figure 6, where points (1) and (2) present the disk layouts with small and large fragmentation, respectively. When the number of file operations arrives at each point, the benchmark unmounts each file system to flush all blocks cached in memory in order to exactly measure the time spent in reading only the data on the disks, not in the file cache. After the flushing, it mounts the file system again and reads all files in each directory, which is selected sequentially.

Figure 7(a) and 7(b) present read performance and layout score at point (1) of Figure 6. Figure 7(a) shows that DFS-16 and DFS-24 achieve slightly better performance of small file read than FFS by 5% and 8% respectively. This improvement results from the difference in the layout score among the three file systems, as seen in Figure 7(b). DFS-16 and DFS-24 have higher layout score than that of FFS by 23% and 30% respectively, and the improvement over FFS is especially apparent in the range of 2 to 12 blocks. However, the improvement is relatively small because the three file systems suffer from little fragmentation. Since a small number of file operations have been performed, the disks do not have many files with IAF. In such disk layouts, likewise, target regions triggered by dirty blocks do not have enough files with IAF so that DFS-16 and DFS-24 can relocate them actively. Therefore, this inactive relocation causes the difference in the layout score to be slight.

Figure 7(c) and 7(d) show read performance and layout score indicated by point (2) in Figure 6, where the disks become significantly fragmented. Figure 7(c) shows that DFS-16 and DFS-24 improve small file performance over FFS by 47% and 78%, respectively. These improvements stem from



(a) Read throughput at point (1) (b) Layout score at point (1) (c) Read throughput at point (2) (d) Layout score at point (2)

Figure 7. Read throughput and layout score on disks

a main reason: DFS-16 and DFS-24 prevent small files from being fragmented through active relocation of many such files while FFS suffers from serious file fragmentation. For the experimental evidence, Figure 7(d) shows that DFS-16 and DFS-24 reduce the amount of small file fragmentation by up to 150% and 260% compared to FFS respectively. Moreover, as expected, read performance of 4 to 12 block-sized files has a significant advantage over that of 2 or 3 block-sized files because of the OpenBSD's history-based read-ahead algorithm. However, files larger than 12 blocks still suffer from IAF problems because DFS-16 and DFS-24 do not attempt to relocate them on the disks contiguously.

As seen in Figure 7(d), both DFS-16 and DFS-24 have higher aggregate layout score for 4 to 8 block-sized files than 9 to 12 block-sized files. This is due to the file selection policy of DFS. That is, DFS preferentially selects 4 block-sized files as the smallest files that can be relocated into a target region if it has enough free space to contain them. Then DFS first uses up the free space with 4 to 8 block-sized files. Therefore, this makes it impossible for files larger than 8 blocks to be relocated into the target region. Besides, 4 to 8 block-sized files are created, read and written more actively than those with 9 to 12 blocks, as shown in the distribution of Figure 5. Due to this access pattern, the file cache, as well as target regions, contain more of the former than the latter. Thus, these behaviors cause DFS to relocate 4 to 8 block-sized files more actively.

An interesting result in Figure 7(c) and 7(d) is that DFS-16 and DFS-24 achieve better relative performance of 2 or 3 block-sized files over FFS by nearly 14% and 25%, respectively. As disks become fragmented, data become randomly scattered across the disks. Hence, blocks of files including small files as well as large ones can be placed across several cylinder groups or far away from each other on the same cylinder group. This makes it possible for file systems to access blocks of each file with more disk seeks and larger rotational delay. However, DFS-16 and DFS-24 relocate and cluster blocks of small fragmented files on the cylinder groups that each file was allocated. Moreover, they relocate related small files into the same cylinder group; otherwise, each of the files might be scattered across several cylinder groups. This relocation reduces the disk head movement required to successively read small files in the same directory.

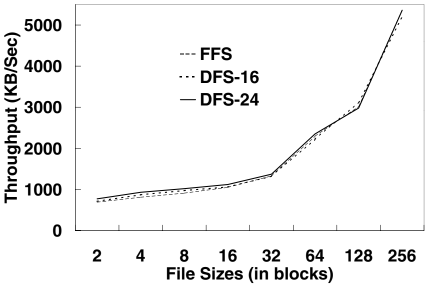


Figure 8. Write throughput

5.5 Write Performance

Figure 8 shows write performance at point (2) in Figure 6. When the number of file operations arrives at the point, the benchmark selects a directory randomly and, then, measures times spent in writing files in the directory. The writes are classified into "overwrite" and "read/write" according to whether data being written are cached in memory. Data being written can be currently cached in memory due to early accesses. Because our testing file systems use write back caching, the writes only modify the data in memory and, then, the modified data are delayed for some period of time. This is called the "overwrite", which spends much less time in writing data. However, if the data are not cached in memory, the file systems fetch them from disks to write them. Hence, this action is called the "read/writes". Because the data that will be written should be fetched, the writes additionally contains the time spent in reading the data.

DFS-16 and DFS-24 improve the write performance over FFS by nearly 7% and 12% respectively. The improvement is because files that are fetched during "read/write" operations suffer from less fragmentation in DFS-16 and DFS-24 than FFS. However, the difference in the performance among them is slight because the number of "read/write" operations is smaller than that of "overwrite". As seen in Table 2, the probability of writes is 20%, but is relatively

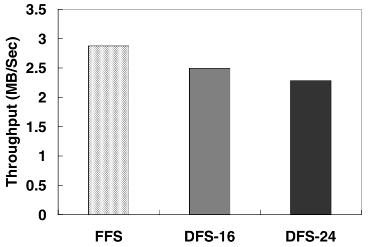


Figure 9. Flushing throughput

	# of relocated blocks	# of disk writes
DFS-16	9.2	1.3
DFS-24	15.4	2.4

Table 3. The average number of relocated blocks and additional metadata writes in flushing operations.

much smaller than the sum of create and read probabilities. Owing to this, it is probable that the benchmark will write files that have been cached in memory due to earlier reads as well as creates.

5.6 Overheads

To examine disk overhead added by DFS's techniques, we measured flushing performance for modified data contained in the file cache when the number of file operations arrived at point (2) in Figure 6. Figure 9 shows that DFS-16 and DFS-24 degrade the flushing performance under FFS by 11% and 17%, respectively. This is because DFS-16 and DFS-24 additionally write multiple blocks together with a dirty block at a single disk I/O. The second column of Table 3 presents how many blocks are stored along with each dirty block in DFS-16 and DFS-24. However, though DFS-16 and DFS-24 store much more data than that of FFS, the disk overhead does not look serious due to the following reasons: first, accessing multiple blocks rather than just one requires a small additional disk overhead in the testing disks. Second, DFS's mechanisms themselves amortize the incremental disk overhead.

Another part of the overhead is how many disk I/Os are addtionally required to update metadata under relocation. In Table 3, the third column shows the average number of disk writes required to update inodes in flushing a dirty block. However, relocated files have inter-file relationships due to the directory locality, and the file system tries to place inodes of the files to the same metadata block as many as possible. As seen in the table, the number of additional disk I/Os required for the updates is small though the inodes of several files are re-mapped. For this reason, the flushing overhead of DFS is not large as expected.

6 Conclusions

DFS gradually alleviates file fragmentation and improve performance for small file reads. It dynamically cluster not only related files that are not placed on disks contiguously, but also blocks of fragmented files. The measurements show that the techniques reduce file fragmentation by an order magnitude for a synthetic workload. Moreover, DFS exceeds FFS in read performance of small files by 78%. Despite this large improvement, penalties added by DFS's mechanisms are a fairly small.

References

[1] M. G. Baker, J. H. Hartmann, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterout. Measurement of a distributed file system. In *13thACM Symposium on Operating Systems Principles*, pages 198–212, Pacific Grove, CA, October 1991.

[2] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, 2001.

[3] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit grouping:exploiting disk bandwidth for small files. In *the 1997 USENIX Annual Technical Conference*, pages 1–17, Anaheim, CA, January 1997.

[4] Maxtor Corporation. http://www.maxtor.com/Quantum/products/archive/fireball-ex/fireball-ex-specs.htm, 1997.

[5] M. McKusick, W. Joy, and S. Leffler. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[6] L. McVoy and S. Kleiman. Extent-like performance from a UNIX file system. In *13th ACM Symposium on Operating Systems Principles*, pages 137–144, October 1991.

[7] E. Riedel, C. van Ingen, and J. Gray. A performance study of sequential I/O on Windows NT 4. In *Proceedings of the second USENIX Windows NT Symposium*, Seattle, Washington, August 1998.

[8] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):285–298, February 1992.

[9] M. Seltzer, K. Bostic, M. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *the Winter 1993 USENIX Conference*, San Diego, CA, January 1993.

[10] M. Seltzer, K. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: A performance comparison. In *the 1995 USENIX Technical Conference*, New Orleans, LA, January 1995.

[11] K. Smith and M. Seltzer. File layout and file system performance. Technical Report TR-35-94, Computer Science Department, Harvard University, 1994.

[12] K. A. Smith and M. Seltzer. File system aging — increasing the relevance of file system benchmarks. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 203–213, Seattle, WA, June 1997.

[13] W. Vogels. File system usage in Windows NT 4.0. In *17th ACM Symposium on Operating Systems Principles*, pages 93–109, Kiawah Island, SC, December 1999.