File System Logging Versus Clustering: A Performance Comparison

Margo Seltzer, Keith A. Smith
Harvard University

Hari Balakrishnan, Jacqueline Chang, Sara McMains, Venkata Padmanabhan
University of California, Berkeley

Abstract

The Log-structured File System (LFS), introduced in 1991 [8], has received much attention for its potential order-of-magnitude improvement in file system performance. Early research results [9] showed that small file performance could scale with processor speed and that cleaning costs could be kept low, allowing LFS to write at an effective bandwidth of 62 to 83% of the maximum. Later work showed that the presence of synchronous disk operations could degrade performance by as much as 62% and that cleaning overhead could become prohibitive in transaction processing workloads, reducing performance by as much as 40% [10]. The same work showed that the addition of clustered reads and writes in the Berkeley Fast File System [6] (FFS) made it competitive with LFS in large-file handling and software development environments as approximated by the Andrew benchmark [4].

These seemingly inconsistent results have caused confusion in the file system research community. This paper presents a detailed performance comparison of the 4.4BSD Log-structured File System and the 4.4BSD Fast File System. Ignoring cleaner overhead, our results show that the order-of-magnitude improvement in performance claimed for LFS applies only to meta-data intensive activities, specifically the creation of files one-kilobyte or less and deletion of files 64 kilobytes or less.

For small files, both systems provide comparable read performance, but LFS offers superior performance on writes. For large files (one megabyte and larger), the performance of the two file systems is comparable. When FFS is tuned for writing, its large-file write performance is approximately 15% better than LFS, but its read performance is 25% worse. When FFS is optimized for reading, its large-file read and write performance is comparable to LFS.

Both LFS and FFS can suffer performance degradation, due to cleaning and disk fragmentation respectively. We find that active FFS file systems function at approximately 85-95% of their maximum performance after two to three years. We examine LFS cleaner performance in a transaction processing environment and find that cleaner overhead reduces LFS performance by more than 33% when the disk is 50% full.

1 Introduction

The challenge in building high performance file systems is in using the disk system efficiently. Since large caches reduce read traffic but do little to reduce write traffic, the critical issue is write performance. Achieving efficient writes to disk implies writing data in large, contiguous units. The central idea in log-structured file systems is that, by aggressively caching data and applying database logging techniques, all disk writes can be made sequential.

The early work in log-structured file systems focused on how to build such file systems. The key issues were providing efficient reads in a file system that was written sequentially and maintaining large contiguous regions on disk. The seminal work on log-structured file systems [9] showed how conventional file system structures could be implemented in an LFS and how the combination of a segmented log and a cleaner process (garbage collector) could be used to maintain large, contiguous regions of disk space. The work's main focus was on the design of log-structured file systems and on the derivation of efficient algorithms for segment cleaning. The performance results reported long-term cleaning summaries (e.g. number of segments cleaned and average utilization of cleaned segments) and micro-benchmarks that demonstrated the strengths of LFS.

The paper by Seltzer et al. [10] discussed design modifications necessary to incorporate LFS into a BSD framework. The performance analysis presented there focused on areas not covered by Rosenblum and Ousterhout, with an emphasis on workloads that stressed the cleaning capabilities of LFS. It concluded that the clustering modifications to FFS made it competitive with LFS in reading and writing large files and in software development environments (as characterized by the Andrew benchmark), cleaning overhead in LFS degraded transaction processing performance by as much as 40%, and the general applicability of LFS and its competitiveness with FFS warranted further investigation. This paper is part of that further investigation, analyzing the performance of LFS and FFS, and focusing on the areas that pose the greatest challenges to each system. We focus on four main issues:

- validating the BSD-LFS implementation by comparing its performance to that of Sprite-LFS,
- the interaction of file size and performance for sequential access,
- the impact of disk fullness on cleaning overhead in a transaction processing workload, and
- the impact of free space fragmentation on FFS performance.

In Section 2 we compare the BSD implementation of LFS to the Sprite implementation of LFS to validate that we are measuring a representative implementation of a log-structured file system. In Section 3 we examine performance as a function of file size. In Section 4 we examine the performance of the two file systems in a transaction processing environment, with special attention given to LFS cleaning and its performance as a function of disk fullness. In Section 5 we discuss the effect of disk fragmentation on FFS performance. Section 6 summarizes this study.

1.1 Overview of FFS

The BSD Fast File System can be described in terms of its *bitmap*, which keeps track of the free space, and its *cylinder groups*, which correspond to collections of cylinders and provide for regions of allocation and clustering. Information is arranged on disk in terms of three units: blocks, partial blocks called *fragments*, and contiguous ranges of blocks called *clusters*. In principle, placing related files and their inodes in the same cylinder group provides for a high degree of locality, and allocating blocks contiguously in clusters provides the opportunity for large, sequential reads and writes. In practice, there are two potential problems with FFS. First, operations that affect the file system meta-data (e.g. creating and deleting files) require a large number of I/O operations, many of which are synchronous. For example, it takes potentially six distinct I/O operations to create a new one-kilobyte file (the inode of the new file, the directory containing the new file, the directory's inode, the data of the new file, the inode bitmap, and the block bitmap), the first two of which are synchronous. The second potential problem is that the FFS block allocator does not take any special measures to preserve large free extents. File systems that have been in use for a long time may become sufficiently fragmented that it is impossible to find large clusters of blocks. The challenges facing FFS can be summarized as reducing the number and synchronous behavior of writes and avoiding file system fragmentation.

1.2 Overview of LFS

In contrast to FFS, LFS avoids both the multiple-write and synchronous write problems by batching large numbers of writes into large, contiguous writes. However, it must maintain large contiguous regions of free space on disk, called *segments*. LFS uses a generational garbage collector [5], called the *cleaner*, to regenerate large free extents. If there is available disk space, the cleaner can always coalesce that free space to produce clean segments. The cleaner can be run during idle periods so as not to interfere with normal file access; however, during periods of high activity it may also be necessary to run the cleaner concurrently with normal file accesses. Depending on the file access patterns, cleaning can potentially be very expensive and can degrade system performance. Thus for LFS the key issue is the cost of cleaning.

2 Validation of 4.4BSD-LFS

The system under study in this work is the 4.4BSD-Lite operating system with implementations of the fast file system (FFS) and the BSD log-structured file system (BSD-LFS), both of which have been improved since the study by Seltzer [10]. Specifically, the read-ahead code used by both FFS and LFS has been largely rewritten. Fragments have been added to LFS. The algorithm for writing blocks to disk in LFS has been modified so that files spanning multiple segments are written with lower-numbered blocks written first. (On a new file system, if N is the number of blocks per segment, blocks zero through N–1 are written to the first segment, N through $2N - 1$ to the next segment, and so on.) Finally, the LFS file writing algorithm has been modified so that files are written to segments in the order in which they entered the cache. This does not affect write performance, but improves read performance when files are read in their creation order.

To show that BSD-LFS is a faithful implementation of a log-structured file system, we have run the benchmarks described by Rosenblum and Ousterhout [9] and compared our results to those reported for Sprite-LFS. The two hardware configurations are shown in Table 1. In each benchmark presented in this section, we scale the Sprite measurements so that the performance of the critical components match those of the BSD configuration. Table 2 shows the relevant parameters and their scale factors.

In the following discussion, we refer to the two file systems Rosenblum studied: the Sprite Log-Structured File System (Sprite-LFS) and the default Sun file system without clustering in effect (Sun-FFS) [7], and the three file systems we have studied: the BSD Log-Structured File System (BSD-LFS), the BSD Fast File System with *maxcontig* of one so that clustering is not in effect (BSD-FFS-m1r2), and the BSD Fast File System with *maxcontig* of eight (BSD-FFS-m8r2). As in Rosenblum's tests, we use a 4 KB file system for the LFS implementation and an 8 KB file system for the FFS implementations. We deduced that the *rotdelay* parameter in the Sun-FFS file system was set to one disk block, indicating that the system was optimized for writing. Similarly, we have optimized the BSD-FFS file systems for writing, setting rotdelay equal to two disk blocks. The tests in Section 3 will examine the interaction of performance and *rotdelay* in more detail.

Benchmark Configurations		
	BSD	**Sprite**
CPU Parameters		
CPU	**SparcStation II**	**Sun 4/260**
Mhz	40	25 Mhz
SPEC int92	21.8	8.7
Disk Parameters		
Disk Type	DSP 3105	Wren IV
RPM	5400	3600
Sector Size	512 bytes	512 bytes
Sectors per Track	57	45
Cylinders	2568	1546
Tracks per Cylinder	14	9
Track Buffer	256 KB	32 KB
Average Seek	9.5 ms	17.5
Maximum Bus Bandwidth	2.3 MB/sec	1.3 MB/sec

Table 1: Benchmark configuration. The Sprite column describes the benchmark configuration used in the Rosenblum and Ousterhout study.

Parameter	Sprite	BSD	Scale
CPU (SPECint92)	8.7	21.8	2.5
Disk Bandwidth	1.4	2.5	1.8
Avg Access (I/Os per second)	39	67	1.7

Table 2: Scale factors for Sprite/BSD comparison. In order to validate the 4.4BSD implementation of LFS, we use these scale factors to compare the measurements on different systems. The average accesses per second is based on an average seek plus one-half rotation.

Rosenblum used two micro-benchmarks to evaluate the performance of Sprite-LFS. The first of these tests measures small-file performance, specifically, meta-data operations (creating and deleting files) and reading small files. Ten thousand one-kilobyte files are created, read in creation order, and deleted. Figure 1 shows the results of this benchmark for the five file systems.

According to Rosenblum, the create and delete tests for LFS are limited by processor speed, while Sun-FFS is limited by disk bandwidth. Our measurements for disk and CPU utilization show that LFS uses 64% of the CPU for creates and 73% for deletes. FFS uses 5-6% for both. Therefore, the LFS results for these tests are scaled by 2.5 (the CPU scale factor) and the Sun-FFS results are scaled by 1.7 (the average access scale factor).

Both log-structured file systems provide an order of magnitude performance improvement over the Fast File Systems for the create phase of the benchmark. Since the files are small, this aspect of the benchmark is dominated by the performance of meta-data operations, and LFS provides superior performance because it batches such operations.

The read test performance is limited by the disk bandwidth, so we scale both Sprite-LFS and Sun-FFS by the bandwidth scale factor (1.8). In both the Sprite benchmarks and the BSD benchmarks, the read performance of LFS and FFS are comparable. The BSD-LFS file system outperforms Sprite-LFS due to larger track buffers on the BSD disk. Once a block of inodes has been read from disk, all the files described by those inodes are resident in the track buffer. Since Sprite's track buffers are one-eighth the size of ours, they cannot cache all the file data.

In the delete test, both LFS systems outperform the FFS systems by an order of magnitude, again due to the meta-data updates, but BSD-LFS outperforms Sprite-LFS by a factor of two. The Sprite-LFS delete performance is baffling. Delete processing in LFS requires the following four steps.

1. Remove the directory entry.
2. Read the inode.
3. Update the appropriate segment usage information to reflect that one kilobyte is no longer in use.
4. Increment the version number in the inode map.

The only disk I/O required is reading the directories and the file inode blocks. As in the create case, we expect this test to be limited by the CPU. However, the CPU processing required for deletes is greater than that required for creates, and the amount of data written for deletes is less than that required for creates. The Sprite-LFS results indicate that the CPU was saturated during the create test, so we cannot explain why the Sprite-LFS delete performance exceeds the create performance. We expect the deletes to occur at close to memory speed with a single write at the end of the benchmark to write the empty directories, the inode map, and segment usage table. This is consistent with the BSD-LFS results.

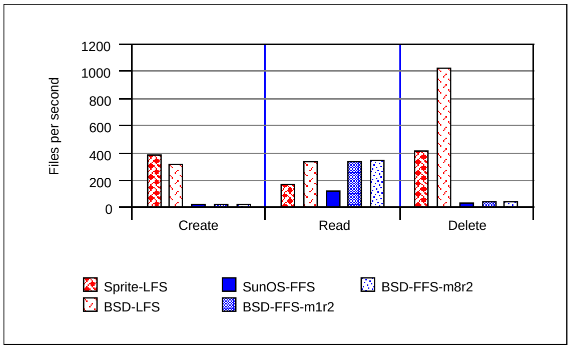

Figure 1. Validation of the BSD-LFS small file performance. The Sprite-LFS results have been scaled to compensate for the performance of the different processors and disks on the two systems. The LFS create and delete tests are scaled by the CPU scale factor, the FFS create and delete tests by the average access scale factor (1.7), and read tests by the bandwidth scale factor (1.8). The BSD-LFS create performance is approximately 25% worse than Sprite-LFS because BSD is not processor CPU bound, and the bandwidth scale factor is half the CPU scale factor. The BSD-LFS read performance dominates Sprite-LFS's due to larger tracks and track buffers. We cannot explain Sprite-LFS's delete performance.

Rosenblum's second benchmark evaluates the performance of large file I/O. This test consists of the following five passes through a 100 megabyte test file.

1. Create the file by sequentially writing 8 KB units.
2. Read the file sequentially in 8 KB units.
3. Write 100 KB of data randomly in 8 KB units.
4. Read 100 KB of data randomly in 8 KB units.
5. Re-read the file sequentially in 8 KB units.

Figure 2 summarizes the results for the five file systems. All the sequential benchmarks and the LFS random write test are scaled by the bandwidth scale factor (1.8) and the remaining random tests are scaled by average access time scale factor (1.7). BSD-LFS and Sprite-LFS display nearly identical performance for the write tests. On the read tests, BSD-LFS displays superior performance due to its aggressive read-ahead and clustered I/O. During a sequential read, the file is read in 64 KB clusters and read-ahead is invoked on clusters rather than individual blocks. The read-ahead algorithm improves the BSD-LFS re-read performance but degrades the random read performance. Although the file was written randomly, all the blocks within a segment are sorted. Since the eight kilobyte I/O requests consist of two logical blocks, read-ahead is always invoked on the second of

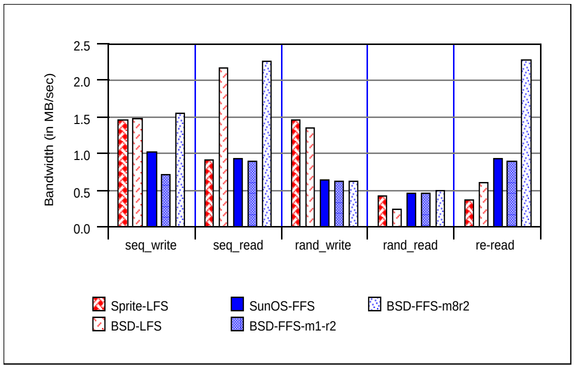

Figure 2. Validation of the BSD-LFS large file performance. The Sprite-LFS sequential read and write and random write results, and the Sun-FFS sequential read, write, and re-read results are scaled by the bandwidth scale factor (1.8). The random read and Sun-FFS random write results are scaled by the average access scale factor (1.7). The BSD-LFS implementation offers write performance equivalent to that of Sprite-LFS. BSD-LFS outperforms Sprite-LFS on sequential reads due to aggressive clustering and read-ahead. The read-ahead policy is also responsible for the degraded performance of random reads in BSD-LFS.

those blocks. Additionally, if logically sequential eight kilobyte units are written before the buffer cache is flushed to disk, the two eight kilobyte units (four blocks) will be allocated contiguously on disk. During the sequential re-read, BSD-LFS will read maximal-sized clusters based on disk layout.

In the random read case, the combination of read-ahead, the eight kilobyte I/O size, and the four kilobyte block size is a worst-case configuration for read-ahead. Two random eight kilobyte reads are actually four block reads to the file system. Assume that we are reading two random eight kilobyte units containing blocks 100, 101, 200, and 201. The read of block 100 is not sequential, so read-ahead is not triggered. However, when block 101 is read, the file system detects sequential access and triggers a read ahead operation on block 102. Unfortunately the next block read is block 200, requiring a seek. Since this is not a sequential access, read-ahead is not invoked on block 201. The end result is that we perform read-ahead when we do not want it (between eight kilobyte units) and we do not perform read-ahead when we do want it (after reading the first four kilobytes of an eight kilobyte unit). This phenomenon explains the low random read performance for BSD-LFS. The Sprite-LFS read numbers are consistent with an implementation that performs no read-ahead. Although the BSD-LFS random read performance is inferior to that of Sprite-LFS, none of the benchmarks analyzed in the remaining sections of this paper trigger the phenomenon described. The TPC-B benchmark discussed in Section 4 is the only benchmark that performs random I/O and its I/O size is equal to the file system block size, so the file system does not identify a pattern of sequential access.

Since Rosenblum did not set *maxcontig* high in Sun-FFS, we expect its performance to be comparable to BSD-FFS-m1r2 performance. This is consistent with all the results in Figure 2 except for the sequential write performance. To understand the performance for this test, we must deduce the *rotdelay* setting for the Rosenblum benchmarks. Typically, *rotdelay* is set to optimize write performance so that a disk revolution is not lost between successive contiguous blocks [6]. Sun-FFS obtains approximately 40% of the possible disk bandwidth when writing sequentially. This is consistent with a *rotdelay* of one block. In BSD-FFS, our disks were sufficiently faster that we had to use a *rotdelay* of two blocks to avoid missing a rotation on every write. This yields write performance of slightly less than one-third the maximal bandwidth, which is consistent with our measurements.

The similarity in performance between Sprite-LFS and BSD-LFS demonstrates that BSD-LFS is an equivalently performing implementation of a log-structured file system. Its write performance is as good as Sprite's and it generally outperforms Sprite for reading due to the aggressive read-ahead and clustered I/O. Sun-FFS is a faithful implementation of the 4BSD FFS. The clustering modifications in 4.4BSD are a reimplementation of the modifications described by McVoy [7]. With these points established, we now compare the performance of BSD-LFS to that of the 4.4BSD FFS with these clustering enhancements. For the remainder of the paper, we use LFS to mean BSD-LFS and FFS to mean BSD-FFS.

3 Sequential Performance as a Function of File Size

Our first comparative benchmark examines the sequential read and write performance of the two file systems across a range of file sizes. The data set consists of 32 megabytes of data, decomposed into the appropriate number of files for the file size being measured. In the case of small files, where directory lookup time dominates all other processing overhead, the files are divided into subdirectories, each containing no more than 100 files. In the case of large files, we use either 32 MB or ten files, whichever generates more data. There are four phases to this benchmark:

- **Create**: All the files are created by issuing the minimal number of writes to create a file of the appropriate size. For all file sizes of four megabytes or less, this means that the test program does one I/O operation. For larger files, I/Os are issued in four-megabyte portions.
- **Read**: Each file is read in its creation order. The I/O operations are identical in size to those during the create test.
- **Write**: Each file is rewritten in its creation order.
- **Delete**: All the files are deleted.

The LFS measurements represent performance when there is no cleaner present and when the set of tests for each file size are run on a newly created file system. The FFS measurements represent empty file system performance; although a new file system is not created for each file size, all files from previous tests are deleted before the next data set is created. Both of these configurations yield the optimal operating conditions for the two file systems.

For all test results shown we provide the LFS performance and two FFS measurements. In both cases, *maxcontig* has been set to 64 kilobytes (the maximum transfer size supported for our controller), but the *rotdelay* parameter varies. On our system, a *rotdelay* of two blocks provides the optimal write performance and a *rotdelay* of zero produces the optimal read performance. While a second write cannot be written before the first write completes, the track buffer caches read data so that the *rotdelay* is unnecessary. We show results for both *rotdelay* settings.

All measurements shown have confidence intervals of plus or minus one percent of the reported number.

3.1 Create Performance

Figure 3 shows the performance for the create phase of the sequential benchmarks. Since LFS buffers a large number of writes before writing any data to disk, the file creates in this benchmark occur asynchronously. In contrast, each time a *creat* system call returns, FFS guarantees that the file has been created and will exist after a system crash. The journaling file systems [4] avoid the synchronous writes of FFS by logging all meta-data operations to an auxiliary data structure, replacing multiple, random, synchronous I/Os with a single sequential one. When files are small the create test measures the ability of the systems to perform meta-data operations; when files are large, the create test measures the write performance of the file systems.

As expected, the asynchronous creation and the sequential writes of LFS yield superior performance to FFS for small files. The order of magnitude performance improvement advertised for LFS is demonstrated during the create benchmark when the file size is one kilobyte. In the two kilobyte to one-half megabyte range, the superiority of LFS degrades from a factor of four at two kilobytes to a factor of two at 32 kilobytes to the same performance at one-half megabyte. When the created files are large (one-half megabyte or more) the performance of the two systems is comparable. The LFS and FFS-m8r0 systems use approximately 67% of the disk bandwidth, losing one rotation between each 64 kilobyte request. The FFS-m8r2 achieves approximately 75% of the disk bandwidth, losing only two blocks or two-thirds of one rotation between 64 kilobyte writes.

3.2 Read Performance

Figure 4 shows the read performance for both LFS and FFS as a function of the log of the file size. In the region between one and eight kilobytes both file systems show low bandwidth, but steady growth as the

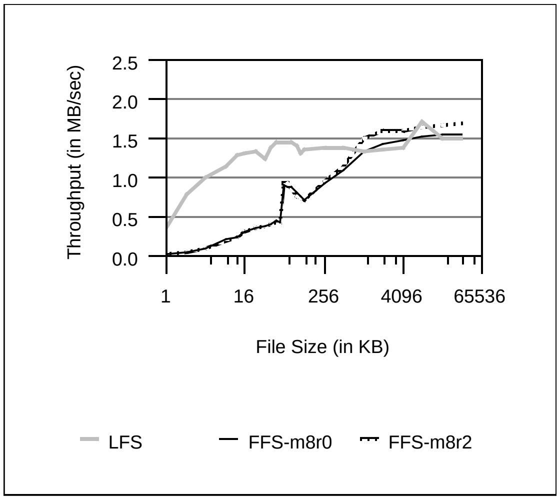

Figure 3. Create performance. When the speed of meta-data operations dominates (for small files less than a few blocks or 64 KB), LFS performance is anywhere from 4 to 10 times better than FFS. As the write bandwidth of the system becomes the limiting factor, the two systems perform comparably.

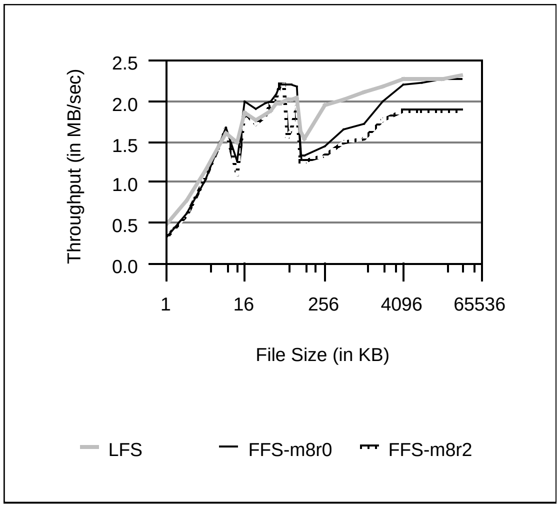

Figure 4. Read performance. For files of less than 64 KB, performance is comparable in all the file systems. At 64 KB, files are composed of multiple clusters and seek penalties rise. In the range between 64 KB and 2 MB, LFS performance dominates because FFS is seeking between cylinder groups to distribute data evenly.

transfer size between the disk and memory increases. Although the track buffer can hide disk latency, each file read results in a separate command to the device.

The dip in performance at sixteen kilobytes is due to the fact that two I/Os are required and read-ahead has not yet become operative. In the region between 8 KB and 64 KB, performance on both file systems improves as each approaches the 64 KB maximum I/O size. Files larger than 64 KB occupy more than one cluster and we see a performance dip when the next I/O operation is added.

FFS performance declines more steeply than LFS. There are two factors at work. FFS leaves a gap between clusters, the size of which is controlled by the *rotdelay* parameter. When *rotdelay* is non-zero and a new file is allocated, its blocks are placed in the gaps left by previously created files. Therefore, at file sizes greater than 64 KB, the files for FFS-m8r2 become increasingly fragmented. LFS does not suffer from this fragmentation since all of the blocks of a file are allocated contiguously. The ability to achieve the superior write performance of the FFS-m8r2 in the create test is precisely the limiting factor in the read case.

The second factor is the introduction of indirect blocks at files of 96 KB or larger. This drop is more pronounced for FFS since FFS begins allocation in a new cylinder group when the first indirect block is added. LFS continues allocating blocks sequentially.

There is a third parameter that can adversely affect FFS read performance in a test that creates many files. The FFS *cpc* (cylinders per cycle) parameter specifies the number of rotationally equivalent positions that will be considered in disk allocation. When a block is allocated, a preferred location is selected. If that location is unavailable, FFS will attempt to find up to *cpc* rotationally equivalent positions. This is exactly correct when allocating blocks to an existing file. However, when creating new files, the preferred block is set to the first block in a cylinder group. After the first file is created, that block is no longer available. Ideally, the next file should be allocated contiguously, but if *cpc* is non-zero, then the file will be allocated to a block that is rotationally equivalent to the first block in the cylinder group. Accessing these two blocks in succession requires a full rotation. Since the disk geometry of many of today’s disks (e.g. SCSI) is not exposed, determining an accurate value for *cpc* is virtually impossible. The numbers reported here use a *cpc* of zero.

For large files, both file systems approach the 2.3 MB/sec bandwidth achievable over our SCSI bus.

3.3 Overwrite Performance

In this test, the 32 MB of data are written to the files already created in the create phase of the benchmark. The results are shown in Figure 5. In this test, FFS need perform no allocation, since the blocks are reused, but LFS must mark blocks dead and create new segments.

For files smaller than a *cluster* (the maximum I/O size supported by the controller, typically 64 KB), the sequential layout of LFS dominates. The performance drop for both systems at 64 KB is due to the cluster size. LFS and FFS-m8r0 lose a disk rotation between each pair of files, because although the files are allocated contiguously on disk, both systems must issue two separate I/O requests to write them. While track buffers can hide this effect during reading, they cannot diminish the effect while writing. The *rotdelay* gap alleviates this problem, but it introduces the performance penalty of fragmenting files whose blocks are allocated in these gaps. The 4.4BSD file system has been enhanced with a new reallocation algorithm that coalesces such fragmented files when they are created. However, the long term effects of this policy have not been thoroughly investigated, so we did not enable this functionality during the test.

For large files (greater than 256 KB), the one-third rotation that FFS-m8r2 saves over FFS-m8r0 and LFS accounts for the 15% performance improvement.

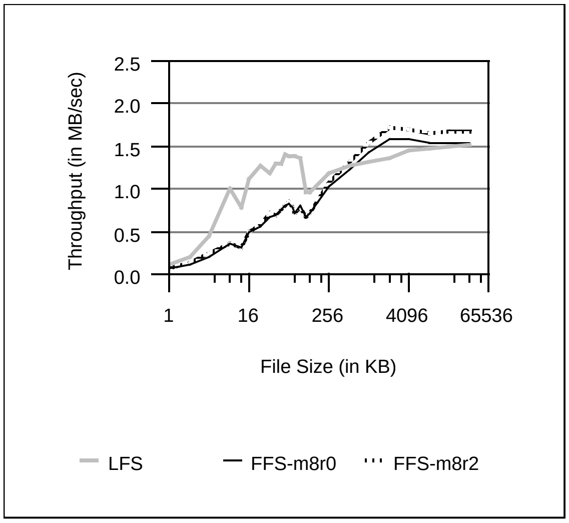

Figure 5. Overwrite performance. The main difference between the overwrite test and the create test is that FFS need not perform synchronous disk operations and LFS must invalidate dead blocks as they are overwritten. As a result, the performance of the two systems is closer with LFS dominating for files of up to 256 KB and FFS dominating for larger file sizes.

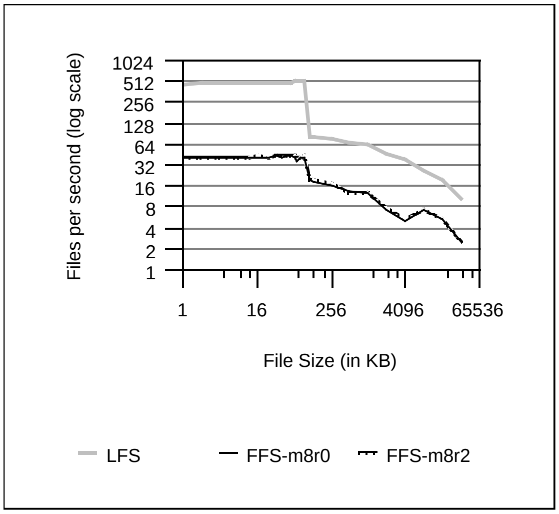

Figure 6. Delete performance. As was the case for creates, the delete performance is a measure of meta-data update performance and the asynchronous operation of LFS gives it an order of magnitude performance advantage over FFS. As the file size increases, the synchronous writes become less significant and LFS provides a factor of 3-4 better performance.

3.4 Delete Performance

The final phase of this benchmark consists of the delete phase. Since this test writes little data, the results in Figure 6 are expressed in deleted files per second. Once again, the asynchronous behavior of LFS meta-data operations provides an order of magnitude performance improvement for small files. The sudden drop in performance occurs when indirect blocks are required. When a file exceeds its direct blocks, the indirect block must be retrieved from disk in order to free blocks (in FFS) or mark them dead (in LFS). Even for large file sizes, the asynchronous deletes of LFS provide three to four times the performance of FFS.

3.5 Benchmark Summary

To summarize these benchmarks, LFS offers an order of magnitude performance improvement in performing meta-data operations (creates and deletes) on small to medium-sized files. For deletes, LFS maintains its performance superiority at large file sizes, deleting at three to four times the rate of FFS.

The read, write, and create bandwidth for large files is comparable on both systems. FFS provides slightly better write performance when the rotdelay parameter is adjusted to avoid missing an entire rotation, and LFS provides slightly better performance reading since there is no effort to distribute the data across cylinder groups as is done in FFS.

For files smaller than the cluster size, read performance is comparable in both systems. LFS provides better write performance as its clustering of multiple small writes into large contiguous ones results in its using 43-65% of the available bandwidth.

As in all benchmarks, this one has many shortcomings. Both file systems are presented under optimal circumstances: all accesses are sequential, access order is identical to create order, the request stream is single-user, no cleaning is required for LFS, and FFS operates on an empty disk. The next section presents a more demanding workload, the TPC-B transaction processing benchmark.

4 Transaction Processing Performance

Although LFS was designed for a UNIX time-sharing workload, there has been speculation that the ability to convert small, random I/Os into large sequential ones would make it ideal for transaction processing [11]. Seltzer et al. measured a modified TPC-B implementation and found that the cleaning overhead severely limited its performance. The disk was 80% full in the benchmark configuration. In this section, we examine the performance of the same benchmark across a range of file system utilizations, since the file system utilization can affect cleaning cost. The benchmark configuration is identical to that described in Section 3, except that the file systems are configured with a four-kilobyte block size to match the block size of the database indexing structures.

The TPC-B benchmark simulates a check-cashing application [13]. The four files in the benchmark are described in Table 3. For each transaction, an account record is read randomly, its balance is updated, the balances in the associated teller and branch records are updated, and a history

File	Size	Description
account	237 MB	1,000,000 100-byte records
branch	44 KB	10 100-byte records
teller	44 KB	100 100-byte records
history	70 KB	Append-only; 50 bytes per transaction

Table 3: File specifications for a 10 TPS TPC-B database. Although our system is capable of supporting more than 10 TPS, we scaled the benchmark database to allow experimentation with disk utilization.

record is written. Although our implementation supports full concurrency control, we ran a single-user version of the test to minimize synchronization overhead and concentrate on disk behavior. All updates to the database are logged to a non-duplexed log residing on a separate disk.

The application maintains its own 4 MB block cache in user virtual memory. As the branch and teller files are rather small, they are always memory resident. In contrast, the account file is large, so only the internal pages of the B-tree index structure are likely to be memory resident. Therefore, the file system activity generated for each transaction is a random read of a block in the account file, followed by a random write from the user-level cache to the file system in order to make room for the newly retrieved account block. The newly retrieved account block is then dirtied and left in the cache until it is later evicted.

Three sets of results are shown in Figure 7. The top line indicates the performance of LFS in the absence of the cleaner. The performance was measured on the lowest utilization (48%) and projected across all utilizations since LFS will quickly run out of disk space if no cleaner is running. The second line on the graph shows FFS performance as a function of file system utilization. As expected, FFS shows no performance fluctuation as the disk becomes fuller. With the exception of the history file, every disk write in the benchmark merely overwrites an existing disk block, so there is no allocation and the fullness of the disk is irrelevant. Each data point represents the average of 100 iterations whose standard deviation was less than 1%.

In the absence of the cleaner, LFS provides approximately 50% better performance than FFS. The 50% performance difference can be attributed to LFS's ability to perform the random writes as sequential writes. In the LFS case, as dirty pages are evicted from the user-level buffer cache, they are copied into the file system cache. The dirty blocks remain in the cache until the number of dirty blocks exceeds a write threshold and LFS triggers a disk write. With the current system configuration, this triggering occurs when 115 blocks have accumulated (representing 115 transactions). These transactions progress at a rate limited by the time required to randomly read the account records from disk. To read, we must copy the page being read from the kernel into the user cache and must also evict a page from the user cache, copying it into the kernel. On our system, these two copies take approximately 1.8 ms. With a 9.5 ms average seek, a 5.5 ms average rotational delay, and a 1.6 ms transfer time, each random read requires 18.4 ms for a throughput of 54 transactions per second. Next, the segment must be flushed. The 115 data blocks are likely to have caused the 58 indirect blocks in the account file to be dirtied, so our segment contains 115 data blocks, 58 indirect blocks, one inode block, and one segment summary for a total of approximately 700 KB. Using the bandwidth numbers from Section 3.3, we can write the 700 KB at a rate of 1.3 MB/sec for a total time of 0.5 seconds. Therefore, processing 115 transactions requires $115 \times 18.4 + 500$ ms yielding 44 transactions per second, within 7% of our measurement.

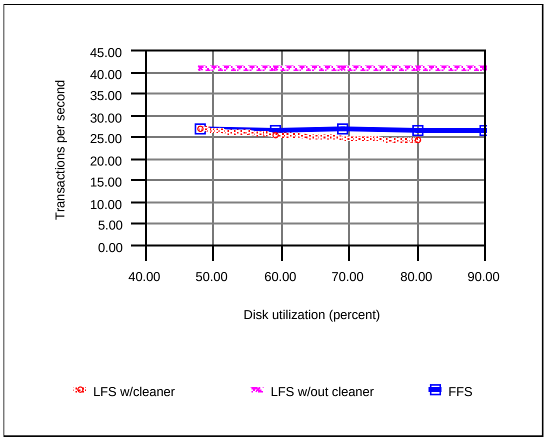

Figure 7. Transaction processing performance. While LFS can potentially provide dramatically improved performance, when the cleaner runs, its performance is comparable to FFS. The performance is largely independent of the disk utilization as the steady-state cleaning overhead is dominated by segment read time.

The calculation for FFS is much simpler: throughput is limited by the performance of the random reads and writes. Each random I/O requires a 9.5 ms seek, a 5.5 ms rotation, a 0.9 ms copy, and a 1.6 ms transfer for a total of 17.5 ms yielding throughput of 28.6 transactions per second, within 7% of our measurement.

The introduction of the cleaner changes the results substantially. At a file system utilization of 48%, LFS performance is comparable to FFS, yielding a performance degradation of 34% due to the cleaner. As the disk becomes more full, the impact increases to the approximately 41% degradation, as observed in [10]. The good news is that this performance is comparable to FFS, but the unfortunate result is that any performance advantage of LFS is already lost at a file system utilization of only 48%.

To understand the LFS performance, we must examine the operation of the cleaner and its interaction with the benchmark. In the steady state, each segment dirtied by LFS requires that the cleaner produce one clean segment. If we were insensitive to response time and wished only to clean most efficiently, we would run the benchmark until the disk filled, then clean, and then restart the benchmark. This should produce the best possible throughput in the presence of the cleaner.

As discussed earlier, LFS fills segments at a rate of 115 transactions per 700 KB or 168 transactions per segment. For simplicity, call the database 256 MB and the disk system 512 MB. This requires 256 segments, 43,000 transactions, or 1000 seconds at the "no-cleaning" LFS rate. After the 1000 seconds have elapsed, LFS must clean. If we wish to clean the entire disk, we must read all 512 segments and write 256 new ones. Let us assume, optimistically, that we can read segments at the full bus bandwidth (2.3 MB/sec) and write them at two-thirds of the disk bandwidth (1.7 MB/sec), missing a rotation between every 64 KB transfer. The cleaning process will take 223 seconds to read and 151 seconds to write for a total of 374 seconds. Therefore, at 50% utilization our best case throughput in the presence of the cleaner is 31.3 transactions per second. This is within 15% of our measured performance.

Unfortunately, LFS cannot clean at the optimal rate described above. First, the transaction response would be unacceptably slow while the cleaner stopped for six minutes to clean. Secondly, the calculations above assumed that the disk is read sequentially. Since the selection of segments is based on Rosenblum's cost-benefit algorithm [9], there is no guarantee that collections of segments being cleaned will be contiguous on disk. Thirdly, the history file in the benchmark grows by 50 bytes per transaction, so file system utilization increases during the test run. Finally, cleaning requires that multiple segments be cached in memory for processing, so we must limit the number of segments cleaned simultaneously.

Since LFS cannot clean at its maximal rate, it should clean at a rate that permits it to perform its segment reads and writes at near-optimal speed. At 50% utilization, we should be able to read two dirty segments and produce one clean segment. Reading one megabyte requires a random seek (9.5 ms) one-half rotation (5.5 ms) and a 1 MB transfer (435 ms) for a total of 450 ms per segment read. Rewriting the segment requires the same seek and rotation, but the transfer requires 588 ms for a total of 603 ms for the write or 1.5 seconds to clean the segment. In the steady state, this cleaning must be done for each 168 transactions. Our throughput without the cleaner is 41 transactions per second, so it takes 4.1 seconds to execute 168 transactions and 1.5 seconds to clean, yielding 5.6 seconds or 30.0 TPS. This is within 10% of our measured number.

It can be argued that LFS loses performance because it writes indirect blocks too frequently (approximately once every three seconds in our benchmarking environment). The current BSD-LFS write policy assumes that when the number of dirty buffers in the cache exceeds the write threshold (the point at which the kernel triggers a segment write), generating clean buffers is essential and it uses an aggressive policy of writing all the dirty buffers to disk. If the dirty indirect blocks were cached during this benchmark, the number of dirty data blocks that are allowed to accumulate in the cache would be reduced and segment writes would occur more frequently. While suboptimal for this benchmark, we believe that flushing indirect blocks with their data blocks is the correct default policy.

5 Effects of Free Space Fragmentation on FFS Performance

Both LFS and FFS rely on the allocation of contiguous disk blocks to achieve high levels of performance. Because the results in Section 3 were obtained from newly-created, empty file systems, there was no shortage of contiguous extents of free space. On real systems, in use for extended periods of time (months or years), the file system cannot expect to find such an ideal arrangement of free space. LFS and FFS deal with this reality in two different ways, both of which cause performance overhead for the respective file systems.

LFS relies on the cleaner process to garbage collect old segments, creating large regions of free space—clean segments. The cleaner imposes overhead on the overall performance of LFS. Section 4 discusses this overhead in the context of a transaction processing workload.

In contrast, FFS makes no assumptions about the layout of free space on the file system. FFS uses whatever free space is available on the disk, contiguous or not. In fact, the block allocation policy of FFS remained unchanged when clustering was added [7]. FFS may not allocate contiguous blocks to a file, even when contiguous free space is available. As with the LFS cleaner, this may adversely effect performance.

The fragmentation of free space in an FFS may increase with time or with file system utilization. This fragmentation can degrade performance as an FFS ages. To assess this risk, we studied a collection of FFS file systems on the file servers of the Division of Applied Science at Harvard University over a period of nine months to examine how the performance of FFS file systems under real workloads differs from the performance of the empty FFS file systems typically used for benchmarking.

5.1 Data Collection

We collected data from fifty-one file systems on four file servers. All of the file servers were SparcStations running SunOS 4.1.3. Although the operating system is substantially different than the 4.4BSD operating system used elsewhere in this study, the file systems are nearly identical. (The BSD-FFS clustering enhancements were based on those originally implemented in SunOS [7].)

Our data collection consisted of daily *snapshots* recorded for every file system under study. A snapshot is a summary of a file system's meta-data, including information about the size and configuration of the file system, the age, size, and location of each file, and a map of the locations of free blocks on the file system.

In the interest of brevity, the presentation here is limited to six representative file systems. The remaining file systems in the study demonstrated behavior similar to one or more of these. The important attributes of these file systems are summarized in Table 4.

5.2 Data Analysis

A separate study performed extensive analysis of this snapshot data [12]. Examining this data in conjunction with the FFS block allocation algorithm provided a variety of interesting information characterizing the layout of FFS file systems. Some of the important results are summarized here.

An evaluation of the FFS block allocation algorithm showed that when a new file is created, FFS attempts to allocate space for it starting from the beginning of its cylinder group. The first free block in a cylinder group is almost always allocated as the first block of a new file. FFS attempts to extend the file according to the constraints of the *maxcontig* and *rotdelay* parameters. In practice, this means that as a file grows, it uses free blocks in order from the beginning of the cylinder group. File systems with a non-zero *rotdelay* may occasionally skip blocks as dictated by this parameter. All of the file systems studied here had a *rotdelay* of zero.

Because of this allocation pattern, free space within a cylinder group tends to be unevenly distributed, with most of the free space located at the end of the cylinder group. Furthermore, the free space near the beginning of a cylinder group is more fragmented than the free space near its end. This uneven distribution of free space, combined with the FFS block allocation algorithm, causes small multi-block files to be more fragmented than larger files. When a small file is created, FFS allocates space to it starting from the beginning of a cylinder group. This space is likely to be highly fragmented, resulting in a fragmented file. Space for larger files is also allocated from the beginning of a cylinder group, but a large file

Name	Server	Size (MB)	Age (months)	ncg	bpg	Descriptions
cnews	das-news	565	31	49	1539	News articles and software
glan5	virtual12	1705	24	89	2596	Installed software and sources
staff	das-news	353	30	31	1539	System Administrators' accounts
usr4	speed	953	17	85	1520	User accounts
usr6	endor	953	14	85	1520	Course accounts; 1st-year grad accounts
white	endor	450	14	40	1520	User accounts for theory

Table 4: File system summary for FFS fragmentation study. All file systems had an eight kilobyte block size, *maxcontig* of seven and *rotdelay* of 0. The Age column indicates the file system age as of November 1, 1994. Subtract nine months to obtain the age at the beginning of the study. The ncg and bpg columns indicate the number of cylinder groups and the number of file system blocks per cylinder group, respectively.

is more likely to be able to take advantage of the well clustered free space later in the cylinder group.

The data collected by the snapshots showed that only 37% of the two block files allocated on twelve different file systems were allocated to two consecutive blocks on the file system. In contrast, 80% of the blocks allocated to 256K (32 block) files were contiguous.

The amount of fragmentation in a file is of critical importance because it is one of the primary factors determining file system performance when reading or writing the file.

5.3 Performance Tests

Because the SunOS file systems we studied were in active use, it was not feasible to run benchmarks on them. Instead, the meta-data in the snapshots was used to reconstruct the file systems on the disk of a test machine. This not only allowed the file systems to be analyzed in an otherwise quiescent environment, but also made it easier to study the performance of comparable empty file systems. In the following discussion, the term “original file system” is used to refer to the actual file systems on the file servers, and “test file system” or “copied file system” is used for the file systems reproduced on the test machine. The benchmarks described in this section were run on a SparcStation I with 32 megabytes of memory, running 4.4BSD-Lite. The details of the system are summarized in Table 5.

In order to approximate the original file systems’ configurations as closely as possible, the test file systems were created using the same values of *rotdelay* and *maxcontig*. The test file systems were also configured with the same number of cylinder groups as the corresponding original file systems. When different sized cylinders on the original and test disks made it impossible to precisely duplicate the size of the cylinder groups, slightly larger cylinder groups were created on the test file system. The extra blocks in each cylinder group were marked as allocated, so the test file system could not use them.

FFS Fragmentation Benchmark Configuration	
CPU Parameters	
CPU	SparcStation I
Mhz	20
Disk Parameters	
Disk Type	Fujitsu M2694EXA
RPM	5400
Sector Size	512 bytes
Sectors per Track	68-111
Cylinders	1818
Tracks per Cylinder	15
Track Buffer	512 KB
Average Seek	9.5 ms

Table 5: Fragmentation benchmark configuration.

Because all of our file system benchmarks rely on the creation and use of new files, the most important characteristic of the original file systems to reproduce is the arrangement of free space. Other details, such as the precise mapping of allocated blocks to files, and the directory hierarchy, are less important, because they have little or no impact on the layout or the performance of newly created files. Therefore the only meta-data copied from the snapshots of the original file systems was the free space bitmap for each cylinder group. The resulting test file system contained the root directory and no other data, but could only utilize the same free blocks as the original file system.

The benchmarks from Section 3 were used to compare these reconstructed file systems to comparable empty file systems. The latter were created in the same manner described above, but their free space bitmaps were not modified. The create and read phases of the sequential I/O benchmark were used to analyze the performance of the file systems. One minor modification was made to the benchmark program for these tests. Instead of creating one hundred files per directory, the benchmark used here creates only twenty-five files per directory. This configuration stresses the file system more and the CPU less since it reduces the processing overhead in finding directory entries and causes FFS to create more directories, which are distributed among the cylinder groups.

These results presented here show the performance for three different file sizes (eight kilobytes, sixty-four kilobytes, and one megabyte) and four different dates spread over the nine month measurement period.

5.4 Test Results

For each file size tested, the throughput for reading and creating the copied file systems was compared to the throughput for the corresponding empty file system. The graphs in Figure 8 show the performance of each of the test file systems as a percentage of the throughput of the corresponding empty file system. The disk utilization (percentage of total blocks that are allocated) for each file system is also displayed.

Ultimately, the performance of these benchmarks depends on the layout of the test files. Files that are laid out contiguously on the disk can be read and

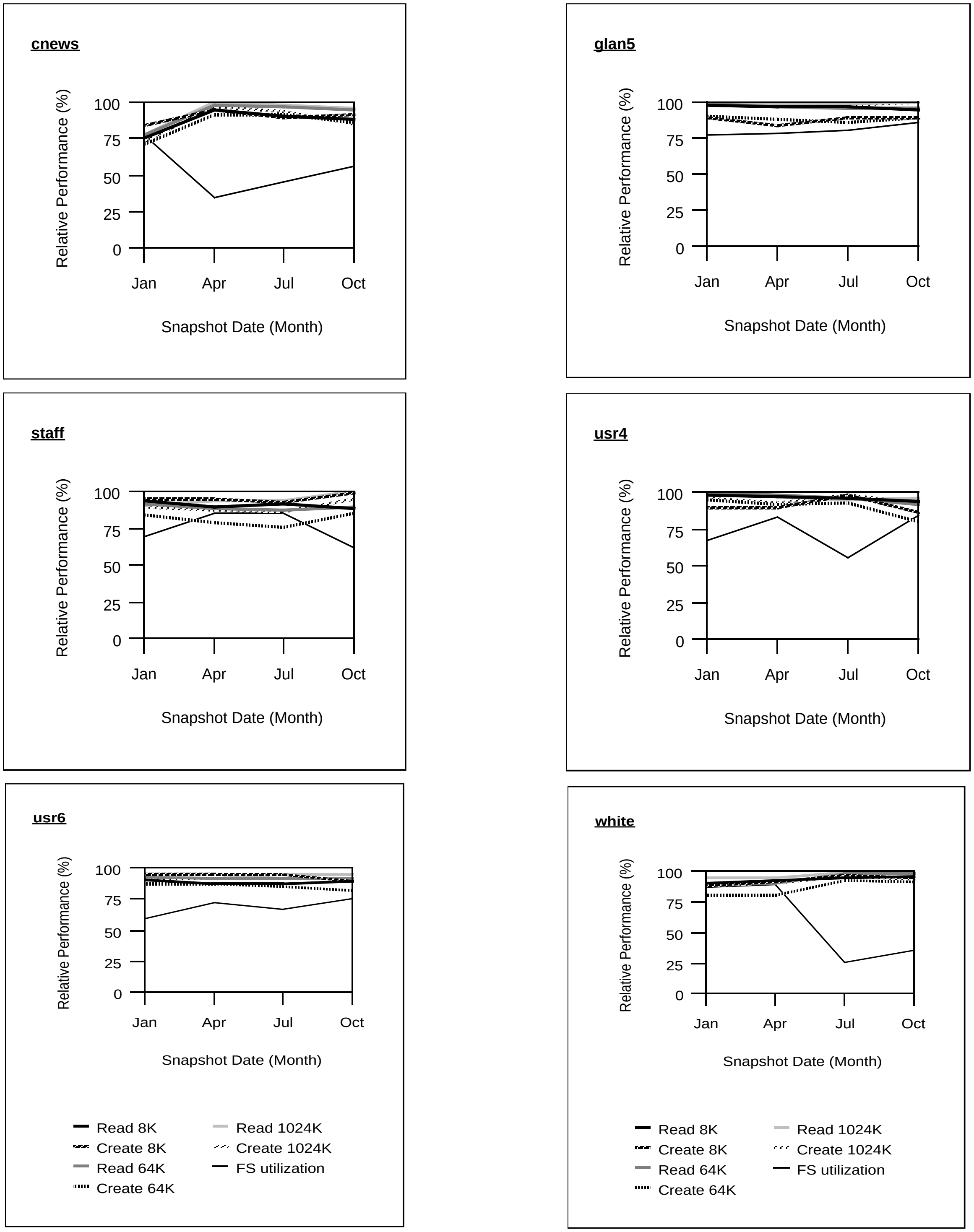

Figure 8. Effect of fragmentation on FFS performance. The graphs display the performance of the six test file systems as a percentage of the performance of a comparable empty FFS on the same disk. The utilization of each file system for each test date is also shown. The benchmarks were run using snapshots of the file systems taken on the 26th day of the designated months (except for the October snapshots for *cnews* and *staff*, which were taken on the 20th). Most file systems exhibit little deterioration over the time period, performing at 85-90% of the empty file system performance. Where changes occur, they often correlate to utilization (e.g. *cnews*, *staff*, *white*, *usr4*). We do not yet see a correlation between file system age and performance degradation.

written quickly. In contrast, files whose blocks are scattered across the disk do not perform as well. Achieving optimal file layout requires contiguous free space on the file system. Thus, anything that causes free space to become fragmented will degrade the performance of a file system. There are three factors that may contribute to the fragmentation of free space in an FFS: high file turnover, high utilization, and file system age.

The news partition, *cnews*, is an example of a file system with high turnover. As a repository for netnews articles, most of the files on this file system are extremely small, and there is a high rate of file turnover. This should cause its free space to rapidly become fragmented. This hypothesis is supported by the fact that approximately two weeks after the January copy of this file system was made, *cnews* reported that it was out of disk space, but an examination of the file system showed that although there were no free blocks, there was 92 megabytes of free space (16% of the file system)—all of it in fragments.

Not surprisingly, the January tests on *cnews* exhibited the greatest performance differences of any of the file systems. For five out of the six test cases, performance was less than 80% of that of the empty *cnews* file system. The single greatest performance difference in all of the tests was in the 64 KB write test on cnews. This test achieved only 71% of the empty file system performance.

In contrast, *glan5* is a file system with very little file turnover. It is used to store the sources and binaries for various software packages. Thus, files on this file system are created or deleted only when new software is installed or when existing software is upgraded. Not surprisingly, *glan5* performed better than the other file systems. Fifteen of the twenty-four tests on copies of *glan5* achieved 95-100% of the performance of the empty *glan5* file system.

The second factor that contributes to fragmentation is high disk utilization. A highly utilized file system has few free blocks, which are unlikely to be well-clustered. The effect of disk utilization on file system performance is demonstrated by several of the file systems in Figure 8. Nearly all of the large changes in utilization are accompanied by inverse changes in performance (see *cnews* and *staff* for particularly noticeable correlation). Although performance drops as the file systems become full, the performance can be regained by removing a fraction of the files.

The third parameter affecting FFS fragmentation is the age of the file system. Even if a file system has a light workload with little file turnover, after several years the cumulative effect of these small changes should be comparable to a high file turnover on a younger file system. The data provides some evidence for this phenomenon, but it is inconclusive. The two oldest file systems (*cnews* and *staff*) are the only file systems where performance on the copied file systems was less than 80% of the performance of the empty file system. As discussed above, however, part of the *cnews* performance should be attributed to its usage pattern rather than its age. Although several test cases performed poorly on the *staff* file system, other tests on same date performed as well on *staff* as on the younger file systems. In our department, file systems “turnover” approximately every three years. That is, disks are replaced, so the data are dumped and restored. An informal poll indicated that this three-year turnaround is fairly typical, and coincides with IRS regulations concerning equipment depreciation.

5.5 File Size and Performance

The graphs in Figure 8 indicate that there are performance differences for the different file sizes tested. Many of the largest differences between the copied file systems and empty file systems occur in the 64 KB test. The 8 KB and 1024 KB were less sensitive to the file system age, utilization, and turnover rate.

For each of the six file systems, we ran read and create tests for each of three file sizes on four different dates for a total of 144 tests. Of the forty-eight tests that used a 1024 KB file size, only two performed at less than 85% of the corresponding empty file system’s bandwidth. These two points were for January on the *cnews* partition. Excluding the January tests on *cnews*, which were generally worse than any other test, of the forty-six tests that used a file size of 8 KB, there was only one test case that achieved less than 85% performance (the April write test on *glan5*, 84.2%). Of the forty-eight tests using 64 KB files, nine performed at less than 85% of the empty file system’s throughput. Of the remaining thirty-nine, thirty-five of them were over 90%.

There are a variety of reasons for the performance differences among the different file sizes. All of the file sizes suffer from the increased fragmentation of free space on the copied file systems. This is most noticeable in the 64 KB files because of ameliorating circumstances in the 8 KB and 1024 KB tests.

The large size of the 1024 KB files allows FFS to perform more read-ahead than for the smaller file sizes. Read-ahead helps to offset the cost of fragmentation by initiating a read (and possibly an accompanying seek) before the data is needed. FFS benefits from read-ahead within a 64 KB file, but does not perform predictive reading of different files. However, on the empty file system, the beginning of the next file will be in the disk's track buffer (if the file is in the same cylinder group and begins on the same track) and can be accessed rapidly. On the copied file system, a seek is often required to access the first block of the next file. This seek is not initiated until the benchmark program has issued the corresponding *read* system call. The 1024 KB file benchmark does not demonstrate this phenomenon because FFS changes to a new cylinder group after allocating the first twelve blocks of a file. In both the empty and copied file systems, a seek is required to move from the end of one file to the beginning of the next.

The fragmentation of free space has little impact in the 8 KB test case. The benchmark creates directories containing twenty-five files. Each directory is placed in a different cylinder group. Since the block size is 8 KB for all of the file systems, the 8 KB test case will read or write twenty-five blocks of data to one cylinder group, then seek to a different cylinder group to start the next directory. Thus, the 8 KB test spends a larger portion of its time performing seeks than either of the other test sizes. Increased fragmentation in the copied file systems means that the files in a given directory are typically spread out more than on an empty file system. Because of the large number of seeks between cylinder groups, however, the amount of overhead introduced by this fragmentation has a smaller impact on overall performance than in the 64 KB test case.

5.6 Benchmark Summary

Performance tests using real-world file systems indicate that there is justification for concerns over the performance impact of fragmentation in FFS. Such concerns should be mitigated, however, by many of the results of these tests. Although the greatest performance differences between a real file system and an empty one were almost 30%, most file systems showed far smaller differences, especially for large files. According to Baker et al. [1], over half the bytes transferred to/from disk come from files over 1 MB in size. The measured performance differences for such files is less than 15% and in 70% of our tests, large files performed at 95% or better on the copied file systems.

It is worth noting that many of the greatest performance differences between real and empty file systems occurred on a file system that demonstrates worst case behavior in almost every way. The *cnews* file system holds small, rapidly replaced files. This file system suffered an unusual fragmentation failure right after these large performance differences were noted. It is difficult to imagine a file system that would incur a greater fragmentation penalty.

6 Conclusions and Future Work

Our results show that the comparison of FFS and LFS is not an easy one. FFS performance can be modified substantially by tweaking parameters such as *maxcontig*, *rotdelay*, and *cpc*.

Unquestionably, when meta-data operations are the bottleneck, LFS provides superior performance to FFS. When creating files of one kilobyte or less, or when deleting files of 64 KB or less, LFS provides an order-of-magnitude performance improvement. This improvement comes in part from LFS's disk layout, and in part from the asynchronous implementation of these operations. There are several alternatives for providing asynchronous meta-data operations, including journaling file systems [4] and ordering updates. (Using an ordered update approach, Ganger reports a factor of five to six improvement in FFS meta-data update performance [3].)

When LFS cleaner overhead is ignored, and FFS runs on a new, unfragmented file system, each file system has regions of performance dominance.

- LFS is an order of magnitude faster on small file creates and deletes.
- The systems are comparable on creates of large files (one-half megabyte or more).
- The systems are comparable on reads of files less than 64 kilobytes.
- LFS read performance is superior between 64 kilobytes and four megabytes, after which FFS is comparable.
- LFS write performance is superior for files of 256 kilobytes or less.
- FFS write performance is superior for files larger than 256 kilobytes.

Cleaning overhead can degrade LFS performance by more than 34% in a transaction processing environment. Fragmentation can degrade FFS performance, over a two to three year period, by at most 15% in most environments but by as much as 30% in file systems such as a news partition.

There is more work to be done. The effects of cleaning on LFS in other environments are still not fully understood. Trace analysis indicates that in a network of workstations environment, there may be sufficient idle time that cleaning can be accomplished without I/O penalty [2].

7 Availability

The file system and benchmark source code and trace data are available and freely redistributable. Send electronic mail to *margo@das.harvard.edu*.

8 Acknowledgments

Many people helped make this paper possible. We would like to thank Carl Staelin and Kirk McKusick who provided considerable help in maintaining our systems at Berkeley; Janusz Juda who ran all our data gathering scripts for months on end; John Ousterhout who originally suggested this work and offered much advice and criticism; Ken Lutz who produced disk specifications with amazing rapidity, and Diane Tang, Larry McVoy, John Wilkes, and David Patterson who made many useful suggestions about our presentation.

9 References

[1] Baker, M., Hartman, J., Kupfer, M., Shirriff, K., Ousterhout, J., "Measurements of a Distributed File System," *Proceedings of the 13th Symposium on Operating System Principles*, Pacific Grove, CA, October 1991, 192-212.

[2] Blackwell, T., Harris, J., Seltzer, M., "Heuristic Cleaning Algorithms in LFS," *Proceedings of the 1995 Usenix Technical Conference*, New Orleans, LA, January 1995.

[3] Ganger, G., Patt, Y., "Metadata Update Performance in File Systems," *Proceedings of the First Usenix Symposium on Operating System Design and Implementation*, Monterey, CA, November, 1994, 49-60.

[4] Howard, J., Kazar, Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, N., West, M., "Scale and Performance in a Distributed File System," *ACM Transaction on Computer Systems 6*, 1 (February 1988), 51-81.

[5] Lieberman, H., Hewitt, C., "A real-time garbage collector based on the lifetimes of objects," *Communications of the ACM, 26*, 6, 1983, 419-429.

[6] McKusick, M.Joy, W., Leffler, S., Fabry, R. "A Fast File System for UNIX, "ACM *Transactions on Computer Systems 2*, 3 (August 1984), 181-197.

[7] McVoy, L., Kleiman, S., "Extent-like Performance from a UNIX File System," *Proceedings of the 1990 Summer Usenix*, Anaheim, CA, June 1990, 137-144.

[8] Rosenblum, M., Ousterhout, J., "The LFS Storage Manager," *Proceedings of the 1990 Summer Usenix*, Anaheim, CA, June 1990, 315-324.

[9] Rosenblum, M., Ousterhout, J., "The Design and Implementation of a Log-Structured File System," ACM *Transactions on Computer Systems 10*, 1 (February 1992), 26-52.

[10] Seltzer, M., Bostic, K., McKusick., M., Staelin, C., "The Design and Implementation of the 4.4BSD Log-Structured File System," *Proceedings of the 1993 Winter Usenix*, January 1993, San Diego, CA.

[11] Seltzer, M., "Transaction Support in a Log-Structured File System," *Proceedings of the Ninth International Conference on Data Engineering*, Vienna, Austria, April 1993.

[12] Smith, K. A., Seltzer., M, "File Layout and File System Performance," Harvard Division of Applied Sciences Technical Report, 1994.

[13] Transaction Processing Performance Council, "TPC Benchmark B Standard Specification," Waterside Associates, Fremont, CA., August 1990.