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 [5] has received much attention for its potential order-of-magnitude improvement in file system performance. Early research results [6] 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 bandwidth. Later work in [8] showed that the need for 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%. The same work showed that the addition of clustered reads and writes in the Berkeley Fast File System (FFS) makes it competitive with LFS in large-file handling and software development environments as exemplified by the Andrew benchmark [3].

These seemingly inconsistent results have caused confusion in the file system research community. This paper presents a detailed performance comparison of the 4.4 BSD Log-structured File System and the 4.4 BSD Fast File System. We investigate the read and write performance of the two file systems across a wide range of file sizes (one kilobyte to four megabyte) and examine the impact of disk utilization on the performance of a transaction processing workload.

Our results show that the potential order-of-magnitude speedup in writing from LFS is not realized. Instead, for small files (under one-half megabyte), LFS offers a factor of 1.5 - 2.5 improvement over FFS. For large files (one megabyte and larger), the performance of the two file systems is comparable. Over time, FFS suffers mildly increased fragmentation, but over the typical lifetime of a file system, this degrades performance by only 10-15%. The read performance of the two file systems is comparable, with both file systems dependent upon the correlation between the order in which files are written to disk and the order in which they are read. In the transaction processing area, even though LFS can outperform FFS by 50% without the cleaner, the introduction of the cleaner brings the performance below that of FFS on file systems that are 50% or more utilized.

1.0 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 a 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 challenges 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 [6] showed how conventional file system structures could be implemented in an LFS and how the combination of a segmented log and 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 measurements of the file system and cleaner and presented micro-benchmarks that demonstrated the strengths of LFS.

The work of Seltzer et al. [8] discussed design modifications necessary to incorporate LFS into a BSD framework. The performance analysis presented there focused on those areas not covered in [6] with an emphasis on environments that stressed the cleaning capabilities of LFS. This work 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 degrades 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. We focus on three main issues:

- the interaction of file size and performance on sequential access,
- the impact of disk utilization on cleaning overhead in a transaction processing workload, and
- the impact of file layout on FFS clustering.

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 so fragmented that it is impossible to find large clusters of blocks. The two 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 a one large, contiguous write. However, it must maintain large contiguous regions of disk called *segments*. LFS uses a copying garbage collector, 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. If possible, 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 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.

This paper analyzes the performance of LFS and FFS, focusing on the areas that pose the greatest challenges to each system. In section 2, we examine performance as a function of file size. In section 3, 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 utilization. In section 4, we discuss the interaction of FFS clustering and disk allocation policy. Section 5 summarizes the study.

2.0 Sequential Performance as a Function of File Size

Our first benchmark examines the sequential read and write performance of the two file systems across a wide 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 can dominate all other processing overhead, the files are divided into subdirectories each containing no more than 100 files. Before the benchmark is run, all the test files are created. The benchmark consists of reading or writing the files in their creation order. Note that since there are more files in the small-file tests, the benchmark performs more open and close operations, increasing overhead for those tests.

Our benchmark configuration consists of a SPARC-2, running BSD4.4-Lite, with 40 megabytes of memory and several DSP 3105 3.5-inch SCSI drives. The details of the system are summarized in Table 1.

SparcStation 2	
CPU	CY7C601
Mhz	40
External Cache	0 KB
I/D Cache	64 KB
SPEC int92	21.8
File System Parameters	
Block size	8 KB
maxcontig	8
rotdelay	4 ms (2 8KB blocks)
DSP 3105 SCSI Disk	
RPM	5400
Sector Size	512 bytes
Sectors per Track	57
Cylinders	2568
Tracks per Cylinder	14
Average Seek	9.5 ms

Table 1. Benchmark Configuration.

2.1 Read Performance

Figure 1 shows the read performance for both LFS and FFS as a function of the log of the file size.

In the region from 1 KB to 8 KB both file systems show low bandwidth, but linear growth. LFS is handicapped in this test in two ways. First, the current implementation does not support fragments. When a file smaller than 8 KB is read or written, LFS reads/writes the entire block, thus wasting some fraction of the disk bandwidth. We confirmed this by rerunning the small file tests on a file system with a 1-kilobyte block size. As expected, performance is approximately 1 MB/sec. Second, there is no mechanism in the file system to guarantee that the files are written to disk in the same order in which they were created. If the files were written in order of the time of last modification, then rereading the files in create-order would produce sequential reading. Unfortunately, the files are actually laid out in reverse order, producing pessimal ordering for this benchmark. While it is possible to sort the files by last modification time, in practice, it is unclear that this would impact a real workload. The FFS performance is poor in this region because it is issuing individual I/Os for each of the inodes and the files being read. Since the inodes and data blocks are placed in separate regions of the cylinder group, FFS must pay seeks and rotational delays between the two I/Os.

In the region between 8 KB and 56 KB, performance on both file systems improves as each approaches the 56 KB transfer size to the disk. Files larger than 56 KB occupy more than one cluster (the cluster size is determined by the maximum transfer size supported by the SCSI controller and is 56 KB on this system). In FFS, the cluster size is controlled by the *maxcontig* parameter. Files that consist of more than one cluster

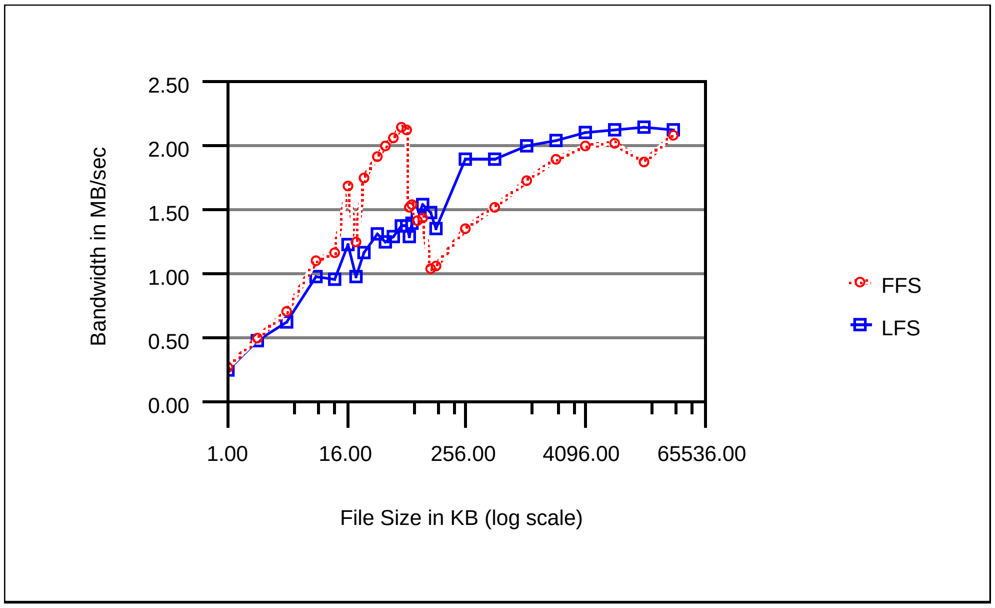

Figure 1. Read performance as a function of file size. In the region between 16 KB and 96 KB, FFS provides superior performance due to the ordering of the files on disk. The LFS allocator has placed the files in the reverse order of their creation, thereby incurring additional seeks and rotations between successive file accesses. In the region between 96 KB and 4 MB, LFS provides superior performance due to the tight clustering achieved in its layout. FFS allocates indirect blocks in cylinder groups separate from the file data.

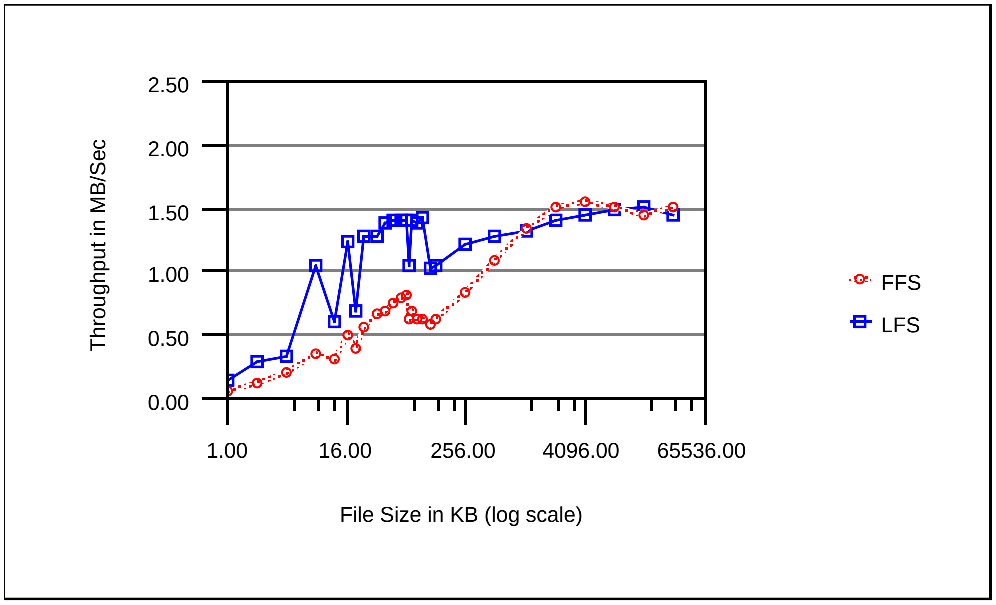

Figure 2. Write Performance as a function of file size. As expected, LFS write performance for small files is superior to FFS with dips introduced by the lack of fragments in our implementation. For large files, the two file systems are comparable, both attaining approximately 2/3 of the maximum bandwidth.

parameter. When a new file is allocated, its blocks are placed in these gaps. Therefore, at file sizes greater than 56 KB, FFS files become increasingly fragmented. LFS does not suffer from this fragmentation since all of the blocks of a file are allocated contiguously. However, it suffers an even greater penalty because there is no gap between 56 KB chunks and the file system loses an entire disk rotation between reads of successive clusters.

Another performance drop occurs at 96 KB when the file systems must use an indirect block requiring an additional I/O. This drop is more pronounced for FFS since FFS allocates its indirect blocks in a different cylinder group (the next group that has more than the average amount of free space).

For large files, both file systems approach a maximum of 2.2 MB/sec which is maximum bandwidth achievable over our SCSI bus.

2.2 Write Performance

In this test, the 32 MB of data are written to the already-created files. The results are shown in Figure 2.

The LFS performance drops at file sizes of 4 KB, 12 KB, 20 KB, 64 KB, and 128 KB. The first three drops are due to the lack of fragments in LFS. Since it does not support fragments, whenever a partial block is needed, it wastes a fraction of the bandwidth writing a full 8-kilobyte block. As was shown in the read tests, the absence of fragments adversely affects performance. The addition of fragments is a top priority in our list of LFS improvements.

The performance drops at 64 KB and 128 KB are due to the limitations of SCSI transfers. The transfer size of the device is specified by a 16-bit quantity, and is therefore restricted to being less than 64 kilobytes. Since LFS must issue two contiguous I/O operations for a 64 kilobyte file, it loses a disk rotation before the final block. While a track buffer can hide this effect during reading, it cannot diminish the effect for writing. One possible work-around is to trade space for time and leave empty blocks in segments similar to FFS’s rotdelay. This would reduce the penalty for multiple I/Os.

The FFS shows the expected gradual increase in performance as the file size grows. There is a small dip at 96 KB where FFS requires indirect blocks which are allocated in different cylinder groups. In the range between 56 KB and 512 KB, FFS performance suffers due to its allocation policy. The gaps that are left between successive clusters are filled with the blocks of the next file. Since the gaps between clusters are only large enough to hold two blocks, files are fragmented and performance suffers.

2.3 Synchronous Write Performance

FFS and LFS offer slightly different semantics with respect to the point at which new file data is written to disk. In LFS, the data is cached until over one-half megabyte of data is ready to be written. In FFS, blocks are written asynchronously as soon as they are filled.

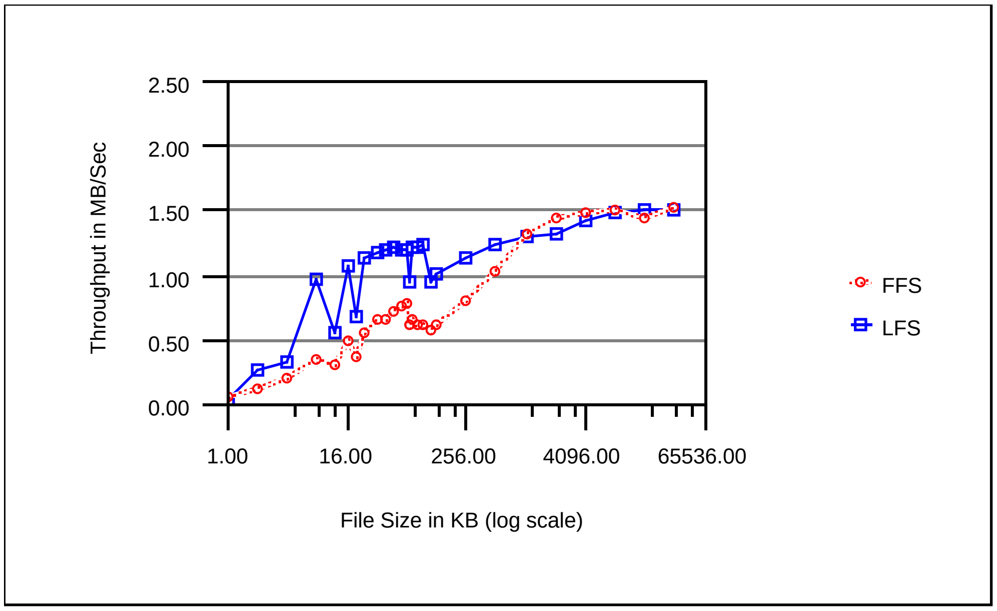

Figure 3. Synchronous write performance as a function of file size. These results do not differ dramatically from those in Figure 2 due to the 56 KB transfer size imposed by the SCSI controller.

In order to measure the impact of this difference, we revised the benchmark of the previous section to issue an *fsync* call after each file was written. The fsync has little impact on FFS as its normal behavior is to write the data out immediately, but for LFS, the fsync forces LFS to write the data sooner than it would otherwise have done, reducing LFS's ability to perform large, sequential writes.

The results, shown in Figure 3, are strikingly similar to those in Figure 2. We might expect LFS to show a decrease in performance since it is no longer able to amass a large amount of data before writing to disk, but the results show only a 15% degradation for midsize files and no degradation for large files. The large file results are not surprising since the fsync takes place after LFS's flushing mechanism has already begun writing data to disk. The surprising midsize file performance is a result of the rotation that LFS normally loses between 56 KB transfers. Since performance is already degraded, the impact of the smaller writes is hardly noticeable.

3.0 Transaction Processing Performance

Although LFS was designed for a UNIX time-sharing load, there has been speculation that the ability to convert small, random I/Os into large sequential ones would make it ideal for transaction processing [9]. Seltzer et al. measured a modified TPC-B implementation and found that the cleaning overhead severely limited its applicability. The disk utilization in the benchmark configuration was 80%. In this section, we examine the performance of the same benchmark across a range of disk utilizations since the disk utilization will affect cleaning cost. The benchmark configuration is identical to that described in Section 2.0, except that the file systems are configured with a four-kilobyte block size to match the block size in the database indexing structures.

The TPC-B benchmark simulates a check-cashing application. There are four main files in the benchmark, described in Table 2. For each transaction, an account

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	CHECK-THIS	Append-only; 50 bytes per transaction

Table 2. File specifications for a 10 TPS TPC-B database.

record is read randomly, its balance is updated, the balances in the associated teller and branch records are updated, and a history record is written. Although we run a single-user version of the test, the implementation supports full concurrency control and logging 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 not, although the internal pages in the Btree index structure are. 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 set of results are shown in Figure 4. The topmost line indicates the performance of LFS in the absence of the cleaner. The performance was measured on the lowest utilized system (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 disk 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. 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 and remain there until LFS triggers a segment 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. In order to perform the read, we must copy the page being read from the kernel into the user cache and must also evict a page form 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, 1 inode block, and 1 segment summary for a total of approximately 700 KB. Using the bandwidth numbers from Section 2.2, we can write the 700 KB at a rate of 1.23 MB/sec for a total time of.6 seconds. Therefore, processing 115 transactions requires 115 * 18.4 + 600 ms yielding 42.6 transactions per second, within 4% of our measurement.

The calculation for EFS 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 disk utilization of 48%, LFS performance is comparable to FFS, yielding a performance degradation of 34% strictly from the impact of the cleaner. As the disk becomes more full, the impact increases to the approximately 41% degradation observed in [8]. The good news is that this performance can be comparable to FFS, but the distressing result is that most of the performance advantage is already lost at disk utilization of only 48%.

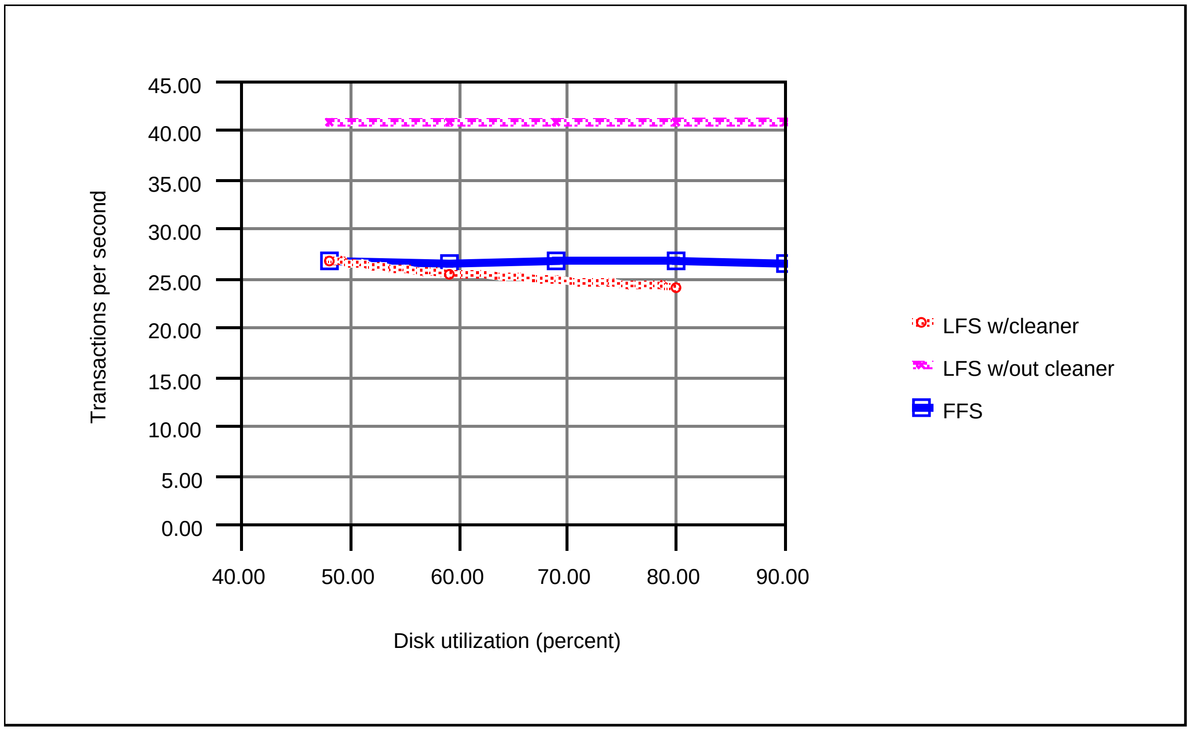

Figure 4. Transaction Processing Performance. While LFS can potentially provide dramatically improved performance, when the cleaner runs, it is only comparable to FFS. The performance is fairly independent of the disk utilization, depending only upon the presence of the cleaner.

4.0 Interaction of Clustering and Disk Allocation in FFS

The results from Section 2 were all measured on newly-created file systems. While the LFS cleaner insures that large, contiguous regions of the disk are always available, FFS makes no such guarantees. In fact, the block allocation policy of FFS remained unchanged when clustering was added [4]. As a result, there is the potential that an FFS will become fragmented and be unable to allocate blocks contiguously. In order to assess this risk, we studied the fragmentation of a collection of SunOS file systems over a period of eight months [10]. (We are continuing to collect data. By the time camera ready copy is due, we will have nearly a full year's worth of this data.)

File fragmentation is important because of the impact it has on file system performance. When reading a file whose blocks are scattered across the file system (i.e., a file which is fragmented), most of the time is spent seeking from one block to the next. In contrast, when a file is well clustered (i.e., not very fragmented), it can be read faster, because fewer seeks are required and more bytes are read with each disk request.

Our results show that file system performance is closely related to the degree of fragmentation in the files that are being read/written. Well-clustered files can be read and written in less time than similar-sized files that are highly fragmented. An analysis of the fragmentation patterns on the file systems in the study found that two-block files are most severely fragmented, with only 40% of their blocks being allocated contiguously. Larger files (nine or more blocks) are more efficiently allocated, with at least 80% of their blocks allocated optimally. Over the lifetime of the file systems in this study, there is a slow increase in the overall fragmentation (the number of blocks optimally allocated decreases by 1.6% per month in the worst case and 0.7% on average). This increase is related to file system utilization as well as to file system age. The important results of this study are summarized in the following sections.

4.1 Data Collection and Analysis

Snapshots were taken of fifty-one different file systems each night for an eight month period. 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 layout of each file on the file system, and a map of the locations of free blocks on the file system. To simplify the presentation, twelve representative file systems were chosen for careful study.

These snapshots were used to analyze trends in file fragmentation. Examining a single snapshot allows us to analyze the fragmentation of the individual files on the file system at the time of the snapshot. By looking at the sequence of snapshots of a single file system, we can search for trends in the changes in file fragmentation.

The fragmentation of a file is measured by a "layout score." This score is computed as the percentage of blocks in the file that are optimally allocated (i.e. that follow their predecessors contiguously). A layout score of 1.00 indicates that a file is completely contiguous. Similarly, a layout score of 0.00 indicates that no two successive blocks of the file are contiguous on the disk. When evaluating the fragmentation of a group of files or of an entire file system, the layout scores for the individual files are averaged, weighted by the number of blocks in each file.

4.2 Layout by File Size

Figure 5 shows the relationship between file size and layout score on one representative file system. The figure shows that layout score improves as file size increases. Two-block files have an average layout score of 0.37, indicating that only 37% of such files are allocated to contiguous disk blocks. Layout score improves quickly for larger files. The average score from eight-block files (0.74) is twice as high as for two-block files. For still larger files, layout score continues to improve, but not as quickly. This correlation between file size and layout score is consistent with the FFS block allocator. Each time a new file is created, the allocator attempts to place blocks at the beginning of cylinder group. Since previously allocated files have left gaps between clusters, more recently allocated files use up these clusters before finding large, unfragmented portions of the disk. When files are small, the majority of their blocks are allocated from the gaps left by older files.

While most files are small, the majority of file I/O is performed to/from large files. In seven out of eight file system traces collected by [1], less than 30% of the bytes transferred to or from the file system were from files of four blocks or fewer (assuming the eight KB block size used by the file systems in [10]). These are the file sizes that

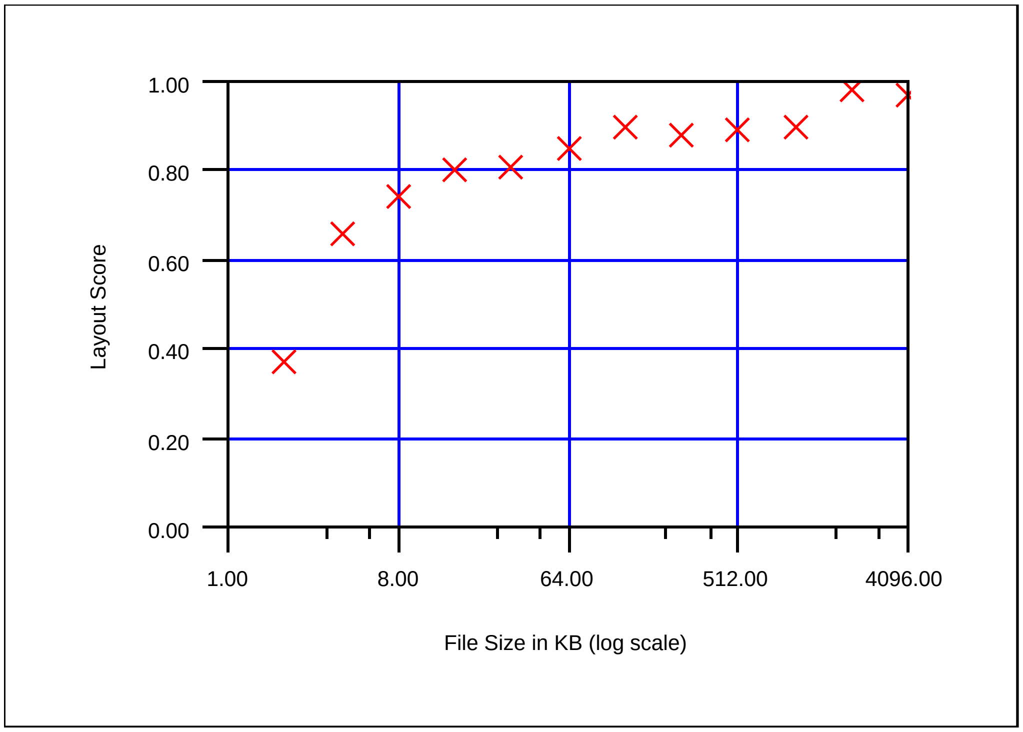

Figure 5. Correlation between Layout Score and File Size. Smaller files have dramatically lower layout scores than the larger files due to the interaction of the FFS rotdelay and the block allocator. The allocator begins allocating blocks of a file at the beginning of the cylinder group, typically using single blocks that have been left in rotdelay gaps.

have the worst layout scores. In other words, most (more than 70%) of the bytes transferred are to/from files with layout scores better than 0.66. Therefore, most I/O activity is to those files that demonstrate the best performance.

4.3 File Layout Over Time

Since the performance of an FFS file system depends on the amount of fragmentation in the file system, it is important to understand how this fragmentation changes over time. Average layout scores were computed for the twelve file systems examined in [10]. These layout scores were tracked on a daily basis through the time of the study. The lowest layout score observed during this period was 0.19 on the news partition. Fragmentation on this file system was substantially worse than on any of the other file systems in the study. This unusual amount of fragmentation is caused by the unique load on the news partition, which consists of many, frequently changing, small files. Ignoring the news partition, the lowest layout score was 0.61. This file system was also the oldest file system in the study (greater than 2 years old).

Of the twelve file systems examined over a four-month period studied in [10], the greatest decline in average layout score occurred on the news partition. The layout score on this file system declined by 0.17 during this period. Again, this seems to be the result of the unusual load on this file system. Among the other file systems studied, the greatest decline in layout score was 0.064, or 0.016 per month. The average decline over all twelve file systems (including the news partition) was 0.029, or 0.00725 per month.

In order to understand the impact of layout score on performance, we ran the benchmarks from Section 2.0 on two of our SunOS file systems. The hardware configuration for this test is shown in Table 3. These two file systems (glan6 and glan8) are representative of our environment. Figure 5 depicts both the bandwidth reported by the benchmark and the layout scores for the file systems under test. These two file systems are the same age (approximately 2 years old), but glan6 has a layout score that is approximately 0.2 greater than glan8 across all file sizes. Similarly, the measured performance of glan6 is approximately 50% better.

It is clear that file system performance is directly related to the layout score we compute. While there is degradation in layout score over time, the degradation is very gentle and does not seem to contribute substantially to a decline in performance. Instead, the particular workload applied to a file system will determine its overall layout score and performance. Furthermore, results of an informal poll of system administrators indicates that most file systems are refreshed (dumped and restored) after approximately three years. The three-year cycle coincides both with the typical lifetime of a disk as well as the depreciation period for the disk.

5.0 Conclusions and Future Work

Our results show that the performance of the FFS with clustering is comparable to that of LFS across many ranges of file sizes. For both very large (greater than or equal to 2 MB) and very small files (smaller than the block size), the results for both reading and writing

.

SparcStation 2	
CPU	Cypress 601
Mhz	40
External Cache	64 KB
I/D Cache	64 KB
SPEC int92	25.0
File System Parameters	
Block size	8 KB
maxcontig	7
rotdelay	0
HP SCSI Disk	
RPM	5400
Sector Size	512 bytes
Sectors per Track	57
Cylinders	2255
Tracks per Cylinder	19
Average Seek	11.5 ms

Table 3. SunOS Benchmark Configuration.

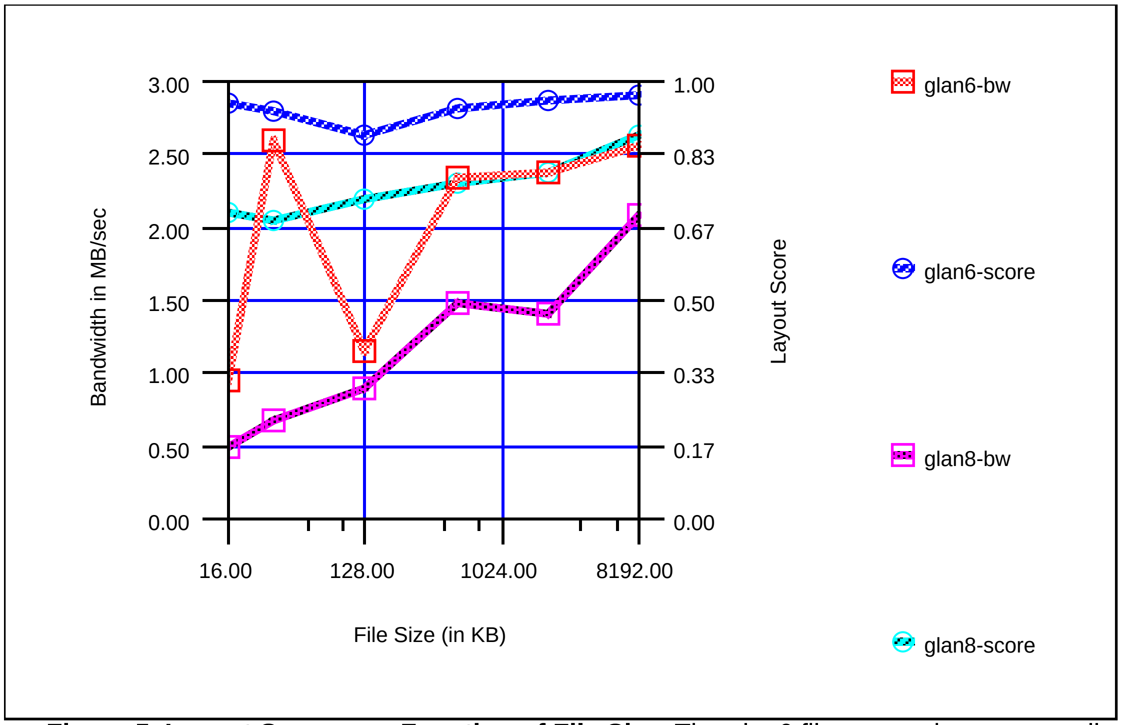

Figure 5. Layout Score as a Function of File Size. The glan6 file system has an overall layout score of 0.84 while glan8 has an overall score of 0.62. The figure shows the layout scores by file size, demonstrating that the layout score of the file system is strongly correlated to the layout score of the large files in the file system. Additionally, the layout scores correlate to the performance of the file system as well. The layout score difference of.22 translates into a performance improvement of between 50% and 75%.

are comparable, although LFS's small file handling will improve with support for fragments in the file system. In the range between 8 KB and 96 KB, the read performance is very dependent upon the layout of the specific files being read/written and the order in which they are read relative to the order in which they are written to disk. After 96 KB, the FFS policy of allocating indirect blocks in potentially different cylinder groups reduces the read performance substantially. For writing, LFS performance is between 1.5 and 2.5 times better than that of FFS for files less than 512 KB.

In the realm of transaction processing, the potential of LFS to offer superior performance is lost as a result of the overhead introduced by cleaning. While the LFS performance with no cleaner is 50% better than that of FFS on a modified TPC-B benchmark, even on a half-empty disk, the cleaning overhead reduces the performance to slightly less than that of FFS. As the disk becomes more full, LFS performance degrades even more.

Concerns over FFS fragmentation over time seem largely unjustified. According to [1], over half the bytes transferred to/from disk come from files over 1 MB in size. Even over time, these large files are efficiently allocated and performance differences between most efficiently and least efficiently laid out files are approximately 15%. The one exception to this result occurs on file systems that hold small, rapidly changing files, such as news partitions. For these applications, LFS is likely to offer substantially better performance than FFS.

This work pointed out several improvements that can be made to LFS and we intend to pursue these modifications. First, LFS needs support for small files. The fragment scheme used by FFS would be sufficient, although a variable-sized block scheme similar to that proposed in [7] might be even more beneficial.

Since the relationship between the order in which files are written to disk and the order in which they are later read can impact performance, it is necessary to understand the relationship between read and create order. Further analysis of the traces discussed in [2] should clarify this relationship. Based on that, we can modify the order in which files are flushed from the cache in LFS.

Finally, while most decisions in the FFS allocator seem correct, we will investigate the impact of allocating indirect blocks in a manner more like the allocation of data blocks (keeping them close to the blocks referenced immediately before them). We anticipate that this will improve FFS performance in the range of files from 96 KB to 2 MB.

6.0 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*, Monterey, CA, October 1991, 192-212.

[2] Blackwell, T., Harris, J., Seltzer, M., "Heuristic Cleaning Algorithms in LFS," submitted to January 1995 Usenix Conference.

[3] 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.

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

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

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

[7] Seltzer, M., Stonebraker, M., “Read Optimized File Systems: A Performance Evaluation,” *Proceedings of the 7th Annual International Conference on Data Engineering*, Kobe, Japan, April 1991.

[8] Seltzer, M., “The Design and Implementation of the 4.4BSD Log-Structured File System,” *Proceedings of the 1993 Winter Usenix*, January 1993, San Diego, CA.

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

[10] Smith, K. A., Seltzer., M, “File Layout and File System Performance,” submitted to Symposium on Operating System Design and Implementation, 1994.