The LFS Storage Manager

Mendel Rosenblum
John K. Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720
415-642-9669
mendel@sprite.berkeley.edu
ouster@sprite.berkeley.edu

Abstract

Advances in computer system technology in the areas of CPUs, disk subsystems, and volatile RAM memory are combining to create performance problems existing file systems are ill-equipped to solve. This paper identifies the problems of using the existing UNIX file systems on 1990's technology and presents an alternative file system design that can use disks an order-of-magnitude more efficiently for typical UNIX workloads. The design, named LFS for log-structured file system, treats the disk as a segmented append-only log. This allows LFS to write many small changes to disk in a single large I/O while still maintaining the fast file reads of existing file systems. In addition, the log-structured approach allows near instantaneous file system crash recovery without coupling CPU and disk performance with synchronous disk writes. This paper describes and justifies the major data structures and algorithms of the LFS design. We compare an implementation of LFS in the Sprite distributed operating system to SunOS's file system running on the same hardware. For tests that create, destroy, or modify files at a high rate, LFS can achieve an order-of-magnitude speedup over SunOS. In spite of its obvious write-optimization, LFS's read performance matches or exceeds the SunOS file system under most common UNIX workloads.

This paper was presented at the Summer '90 USENIX Technical Conference, Anaheim, California, June 1990.

1. Introduction

When the current UNIX file systems were designed, a "fast" computer had a one-MIPS CPU and at most a few megabytes of memory. The machines of the 1990's will have many hundreds of MIPS of CPU and many hundreds of megabytes of memory. Unfortunately, the disks attached to these machines will likely have access times within a factor of five of the 1970's disks. Unless current file systems are modified to match this change of balance between technologies, the machines of the 1990's will operate in an I/O-bound mode where order-of-magnitude increases in CPU speed may produce no visible speed-up in applications.

As faster CPUs shrink the CPU time of programs, the overall execution time will be dominated by the time to perform disk accesses. Large volatile main memories will permit large file caches. Disk delays due to file reads will be alleviated by these caches but file writes must be pushed to disk for reliability. Disk traffic will be dominated by these writes. The challenge for computer systems of the 1990's will be to support these workloads without requiring the redesign of programs in order to change their file access patterns. For the UNIX[1] environment this means efficiently supporting changes to many small files as well as large files.

In the existing UNIX file system, creation and deletion of small files cause disk accesses that are small, non-sequential, and synchronous (the application cannot continue until the disk I/O completes.) The synchronous accesses limit application performance to disk performance. The small non-sequential accesses limit the disk bandwidth utilization to a fraction of the disk's maximum bandwidth. File systems of the 1990's must decouple application and disk performance and use the disks much more efficiently to reduce the I/O bottleneck.

This paper describes a disk storage manager designed to use disks as efficiently as possible to support write-dominated workloads. The storage manager, called **LFS**, uses the concepts of **log-structured file systems**[1] to increase the performance of the UNIX file system. In a log-structured file system, all modifications to the file system including data, directories, and metadata blocks are written to disk in large, sequential transfers that proceed at maximum disk bandwidth. Small file creation and deletion in LFS cause disk accesses that are large, sequential, and asynchronous.

LFS's large sequential writes cause a disk layout that is very different from existing UNIX file systems. Although the layout differs, LFS maintains many of the same metadata structures such as inodes and indirect blocks. This allows LFS to efficiently support the full UNIX file system semantics including fast random and sequential read access to files. The major difference between LFS and UNIX disk layout is that inodes, the disk resident data structure containing file attributes, are not at fixed locations on disk in LFS. LFS must maintain a data structure that tracks the current location of the inode of every file. Once LFS has indexed through this data structure, file reads are identical to UNIX.

The log-structure of LFS requires that large disk regions be available for writing. LFS manages these regions by partitioning the disk into large sequential pieces called

[1] UNIX is a trademark of AT&T.

segments. Blocks are added to the file system a segment at a time and disjoint segments are linked together to form a logically consecutive segmented log. In order to keep segments available for writing, LFS performs an operation called **segment cleaning**. The operation reads fragmented segments into memory, compacts the live data, and writes it back to segments on disk.

The log-structure of LFS provides several other benefits in addition to high performance. The log allows LFS to recover from system crashes much faster than existing UNIX file systems. LFS never needs to scan the entire file system to recover from a crash. The append-only nature of LFS means that files and directories are not updated in place. This allows LFS to provide better crash guarantees and higher performance.

The rest of this paper is organized into 5 sections. It begins with an examination of current technology trends and their effects on storage manager design. The following section, Section 3, describes failures of existing file systems. Section 4 describes the LFS design, and Section 5 presents some performance numbers taken from an implementation in the Sprite distributed operating system[2]. The paper concludes with a status report and discussion of future directions for LFS.

2. Technology and storage managers

This section examines the effect that the current technology trends in CPU speeds, disk performance, and main memory sizes are having on storage managers. Many of the design decisions of LFS can be attributed to the desire to allow graceful scaling of systems in the face of unbalanced technology changes.

2.1. Disks and CPUs speed

The most obvious technology mismatch affecting disk storage managers is the exponential increase in CPU speeds compared with the small increase in disk access speeds. The later part of the 1980's has seen CPUs speeds doubling every year. Disk access times, constrained by mechanics, have only improved by a factor of two in the last decade. Although the the bandwidth and throughput of disk subsystems can be substantially increased by the use of arrays of disks such as RAIDs[3], the access time for small disk accesses is not substantially improved and can even be hurt by this technique. The widening gap between CPU and disk access time suggests that the performance of future systems may be limited by the disk subsystem.

2.2. Memory sizes and disk read/write ratios

Disk and CPU speeds are not the only technology that affects storage manager design. Main memory sizes of machines are increasing with CPU speeds. The fast CPUs of the 1990s will have multi-billions of bytes of memory, from which effective file caches will be built. Large file caches will alter the read-to-write ratio presented to the disk subsystems. Most reads will be satisfied by the cache while most writes must be written to disk for reliability. In contrast to the file systems of the 1970's and 1980's, in which disk traffic was dominated by reads, the file systems of the 1990's will see disk traffic dominated by writes.

2.3. Storage manager design

The radical difference in CPU and disk speeds make it imperative that storage managers decouple an application's performance from the disk access speed. This section describes disk access modes that storage managers can exploit in order to use disks more efficiently.

Asynchronous I/O

Storage managers can lessen the effect of the disk/CPU speed unbalance by avoiding operations in which the CPU is forced to wait for disk operations to complete. Blocking operations, called synchronous disk operations, couple the CPU and disk by requiring the requesting processes to wait while the disk is accessed.

Sequential I/O

Disk access efficiency depends on how the disk is accessed. Disk subsystems can be accessed in sequential mode an order-of-magnitude more efficiently than they can be accessed randomly. Storage managers should exploit this property by performing large sequential accesses. Disk head motion (seeks) should be avoided whenever possible.

3. Problems with existing UNIX file systems

Before presenting the design of LFS it is useful to outline the failures of existing file systems. Although this section uses the BSD file system[4] as an example, the problems are present in most commercial file systems in use today. The major reason that existing file systems will not scale with technology is that they perform too many random and synchronous disk operations.

A disk storage manager design is strongly influenced by the expected workload it must support. LFS is designed to support the workload of the office/engineering environment. This environment has been characterized (see[5]) by a large number of relatively small files (less than 8 kilobytes) whose contents are accessed sequentially and in their entirety. The average file life time is short, less than a day before it is overwritten or deleted. Efficient handling of file systems containing both small, rapidly changing files and large files is needed by this and many other UNIX environments.

3.1. File creation example

The performance problems of the BSD file system in the office/engineering environment can be seen by examining the disk access patterns caused by small file creation. Figure 1 shows the disk accesses required to create two single-block files in different directories. The UNIX system calls used to create the files were:

```
fd = creat("dir1/file1", 0);
write(fd,buffer,blockSize);
close(fd);
fd = creat("dir2/file2", 0);
write(fd,buffer,blockSize);
close(fd);
```

Figure 1 illustrates the disturbing access patterns of many small random and synchronous writes. Each *creat* system call causes a random synchronous write of both the allocated inode block and the directory block. Synchronous writes of directory and inode

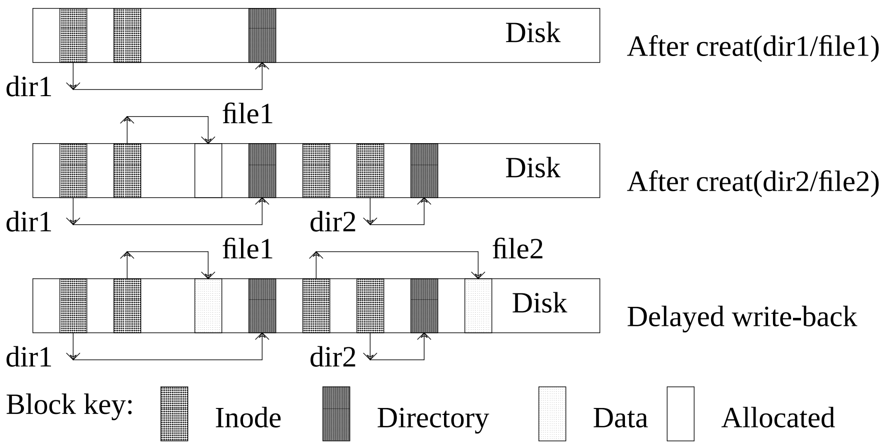

Figure 1 — BSD file system file creation.
Disk image of the BSD file system at different points in the creation of two files. The state is shown after each *creat* system call and after the delayed write-back has occurred. Each *creat* call forces the modified directory block and newly allocated inode to disk. For example during the first *creat*, file1's inode and dir1's inode and directory data block are written to disk. The write system calls allocate blocks on disk but the blocks are not written to disk until the delayed write-back occurs. After the second *creat*, the data block of file1 is allocated but is not written to disk until the delayed write-back.

modifications are intended to limit the damage caused by system crashes. The file data blocks and inode blocks for the directories are allocated in a manner that allows fast reads. Unfortunately, when files are small, allocations cause small random writes. The file data blocks and inode blocks for the directories are written asynchronously but still randomly. The total disk I/O in this example includes 8 random writes of which half are synchronous. Because of these random accesses, only a small fraction of the maximum disk write bandwidth can be used by the file system. The synchronous updates effectively limit the speed of file creation and deletion to the disk's speeds. For example, a .9-MIPS DEC MicroVaxII using the BSD file system can create and delete an empty file in 100 milliseconds. A 14-MIPS DEC DecStation 3100 using the same file system can create and delete an empty file in 80 milliseconds. Because of the synchronous disk I/O, an order-of-magnitude increase in CPU speeds causes only a 20 percent increase in program speed!

4. LFS storage manager design

Efficiently supporting office/engineering workloads means eliminating small synchronous writes. This section presents the major data structures and algorithms used by the LFS storage manager to accomplish this goal. The algorithms include those for file writing, file reading, and disk free space management.

4.1. File Writing

The key idea of LFS is to make writes fast by accessing the disk sequentially and asynchronously. LFS collects the changes to the file system in the file cache, packs them together, and writes them to disk in large sequential disk transfers. Modified file data blocks, directory blocks, and file system metadata such as inode blocks, are packed together and written sequentially to disk. LFS provides the abstraction that data is added to the file system in an append-only log format. The log format implies that LFS never updates disk blocks in place. This feature is the key to LFS's fast recovery and its ability to eliminate synchronous writes.

Because all writes are asynchronous, LFS uses the file cache as a write buffer that accumulates changes to the file system and performs speed matching between the CPU and disk subsystem. Bursts of small writes to the file system are combined together in the file cache and converted into large transfers. This conversion means that, unlike the UNIX file system, file creation and deletion speeds in LFS are tied to the disk's maximum bandwidth and not its latency. Figure 2 shows the single disk access generated by the LFS storage manager executing the file creation example from Section 3.1. In LFS, the *creat* system call simply allocates the file's number and modifies the directory and inode in memory. No synchronous disk writes are performed. The modified file and directory blocks are packed together and written in a single transfer. This form of I/O can be an order-of-magnitude faster than the many random write transfers done by the UNIX file system. For workloads that are limited by disk write performance, LFS permits files of any size to be written to disk at near maximum bandwidth. The elimination of synchronous writes allows LFS to scale with CPU speeds.

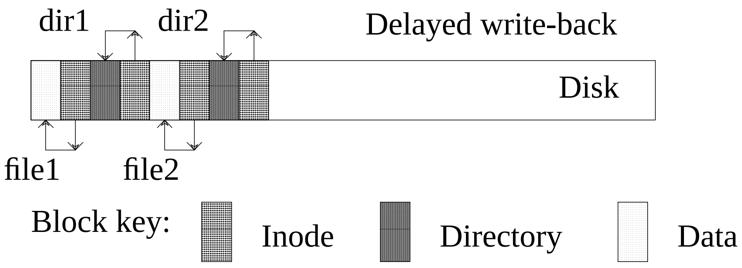

Figure 2 — LFS file system file creation.
Disk image of the LFS file system after two files are created. When a disk write is started, all modified file system blocks are packed together and written to disk in a single transfer. Note that the formats of directories and inodes are the same as in the BSD example (Figure 1). Rather than placing the blocks to optimize reading, LFS writes all changes sequentially to disk. LFS performs the 8 writes in one large transfer. Unlike the BSD example, all writes are sequential and none are synchronous.

4.2. File reading

This section describes the data structures and algorithms used to read files from an LFS file system. Given the very large file caches that future systems will have, it is likely that read performance will have only a small impact on overall system performance. Nevertheless, LFS's read performance will match or exceed the performance of current read-optimized file systems in many cases. Although LFS always writes blocks sequentially, it still maintains the same data structures that permit fast read access in the UNIX file system.

4.2.1. Inode map

The major difference between file location in LFS and UNIX is that in LFS, inodes are no longer at fixed locations on disks. The append-only log abstraction provided by LFS requires that inodes be written to a new location on disk every time they are modified. LFS quickly locates inodes using a data structure called the **inode map**. The data structure maintains a mapping between an inode number and the current disk address of the inode. The inode map also keeps the inode status (allocated or free), the file's access time[2], and a version number that is updated every time the file is truncated to length zero. The version number usage is discussed in Section 4.3.

For file systems having large numbers of files the inode map can get large. To reduce memory usage, LFS partitions the map into blocks that are cached like regular files. It is expected that the blocks mapping active files will stay memory resident and impose little overhead to the file inode fetch. Modified inode map blocks are written to the log like any other file block. The timing of inode map block writes and the recovery of the inode map after crashes are discussed in Section 4.4.

Except for the address lookup using the inode map, the file reading algorithm of LFS is identical to that of existing UNIX file systems. The format of inodes and indirect blocks is unchanged. For files that are written in their entirety, the log layout algorithm places the data blocks sequentially on disk. The read performance of such a file is excellent because the inode and all of the file's data blocks are located close together on disk.

4.3. Free space management

An LFS file system must manage the disk free space to keep open large regions of consecutive disk sectors. LFS simplifies this task by dividing the disk storage into large fixed-size pieces called **segments**. The idea is that the sequential log abstraction of LFS need not be totally sequential on disk. What really matters is that the log is written in large enough pieces to support I/O at near-maximum disk bandwidth. This can be achieved by sizing segments so that the disk seek at the start of a segment write is amortized across a long data transfer time. The test presented in Section 5 used a segment size of one megabyte.

---

[2]The file's access time is kept in the inode map because it is the only file attribute that is updated when the file is read. Keeping the access time in the inode map rather than the inode allows faithful implementation of the UNIX file access time semantics without inodes constantly moving every time a file is read.

4.3.1. Segment summary blocks

Each LFS segment contains a few sectors of summary information that identify the contents of the segment and allow segments to be formed into a linked list. The information is kept in a region of the segment called the **summary block**. The append-only log abstraction can be visualized by following the links between segments.

For each block in the segment, the summary blocks indicates the file number of the block's file and the position of the block within the file. All the information needed for the summary blocks of a segment is available when a segment is formed in memory. The cost of the summary blocks is small in terms of CPU, disk space, and transfer time overheads.

4.3.2. Segment cleaning

The segmented-log structure of LFS reduces the free space management problem to that of finding a free segment to write. During normal operations segments will become fragmented: blocks in a segment will be overwritten or deleted, leaving the segment partially utilized. LFS generates free segments, called **clean segments**, from fragmented segments using an operation called **segment cleaning**. During segment cleaning two or more fragmented segments are read into memory, combined, and appended to the log on disk. Segment cleaning in LFS is simply a form of incremental **garbage collection**[6] where the fragmented segments are compressed together to create space to write new segments. LFS implements cleaning by reading the live blocks of a segment into the file cache and then using the cache write-back code to combine and copy the blocks into a new segment. The cleaning algorithm proceeds in two phases. During the first phase the live blocks of fragmented segments are identified and read into the cache. The second phase combines the blocks into a new segment and writes them to disk. Segment cleaning in LFS can be overlapped with normal file system operations. Files can be read and written while segments are being cleaned.

4.3.3. Block allocation determination

In order to perform segment cleaning efficiently, the LFS storage manager must be able to identify the "owner" (file number and block offset) of each block in a segment. It also needs to determine if a block is **live**, belonging to an active file, or if a block is **dead**, having been overwritten or truncated. LFS identifies blocks using the segment summary blocks, the inode map, and inodes. The block identification algorithm works for files and directories as follows:

1) The segment's summary block is used to determine the file number and block offset of the block being cleaned. Included in the summary entry is the file's version number from the inode map when the block was written. If the version number does not match the current version number of the file, the block is known to have been deleted or overwritten.

2) If the previous step fails to determine the allocation status of the block, the inode and any indirect blocks that map the block of the file are examined to see if the block is still part of the file. The block is classified as being live if it is still a member of the file.

Using these steps the cleaning algorithm can tell which blocks need to be copied into the new segment. Since total overwrite or deletion are the most common write access modes to files in the workstation environment, Step 1 is able to determine the live blocks quickly. Even if Step 2 must fetch the inode and indirect blocks to check the block's status, these blocks are exactly the blocks that must be fetched and modified anyway when the block of interest is copied to a new segment.

4.3.4. Choosing segments to clean

Although cleaning full segments will not harm the system, it is desirable to choose the segments with the most free space. To keep this information available, LFS keeps a data structure called the **segment usage array** that keeps an estimate of the number of live blocks in each segment in the system. The array is updated when files are truncated or overwritten and when segments are written or cleaned. Since segments are large and the usage array only takes a few bytes per segment, the array is small enough to stay memory-resident. Since the usage level of nonclean segments is used only as a hint during cleaning, costly exact crash recovery of this data structure is not needed.

Cleaning is activated either when the number of clean segments drops below a threshold value or when a user-level process initiates it. The user-level process interface allows cleaning to be initialized at night or other times of slack usage. Segments are cleaned until all segments are either clean or contain at least a file-system-settable fraction of live blocks.

4.3.5. Segment write timing

Because LFS does not perform synchronous writes, algorithms are needed to decide when segments should be written to disk. Segment writes in LFS are initiated by conditions similar to those that cause UNIX file data blocks to be written. A segment write is generated triggered by one of three conditions:

Cache full The file cache may request a segment write when it detects a shortage of clean blocks in the cache. This condition will be triggered when many changes are made to the file system and the file cache is filled with dirty blocks.

Cache write-back The file cache may request a segment write to start if it detects modified blocks older than a certain age threshold. The age threshold is much like the delayed write-back policy of UNIX. The current LFS implementation uses a threshold of 30 seconds.

Sync request A program executing a **sync** or **fsync** system call can cause data to be pushed to disk. The system call will block until the dirty blocks are written to disk.

Note that either of the last two conditions can cause a partial segment to be written because the file cache may not contain enough data to fill an entire segment. This may appear wasteful but since there were not enough dirty blocks for the entire segment this is a situation where the system was running much below capacity. Partially-written segments are handled by the segment cleaning algorithm like any other fragmented segment.

4.4. Crash recovery

The last part of the LFS design presented in this paper is the fast crash recovery system. System crashes can cause file systems to become inconsistent when disk operations are terminated by the crash or delayed writes are not completed. The log structure of LFS allows techniques commonly used in data base systems[7] to be used by LFS for fast recovery from system crashes. Unlike the UNIX file system, which must scan the entire disk after a crash to repair damage, LFS need only examine the tail of the log to find crash damage. LFS speeds recovery even more by periodically marking places in the log where the file system is known to be consistent. From these consistent points, called **checkpoints**, LFS can attach the file system without fear of inconsistent metadata.

The current implementation of LFS does not fully implement the high performance, reliable crash recovery mechanism designed for LFS. A much simpler algorithm with zero recovery time is used in the current implementation and described in this section. The current implementation has more overhead during normal operations and is more vulnerable to data loss at crashes than the recovery system LFS will ultimately use.

4.4.1. Checkpoints

LFS provides for the saving of dynamic file system state using an operation called a **checkpoint**. During a checkpoint, all of the memory-resident data structures that describe the current state of the file system are written to a known disk location called the **checkpoint region**. The checkpoint marks a consistent state of the file system. To allow for recovery from crashes during checkpoints, two checkpoint regions are used and checkpoint writes alternate between them. The checkpoint includes a timestamp that can be used to determine which region is the most recent checkpoint.

During a checkpoint, all modified blocks in the cache including all file system metadata blocks such as the inode map and segment usage array, are packed together and written to segments. Once all modifications are safely on disk, a checkpoint region is written that contains a pointer to the last segment written and the locations of the inode map and segment usage map. Crash recovery consists of nothing more than the normal file system "mount" code that uses the last checkpoint area to recover the file system state. Unfortunately, if the system crashes without writing the cache to disk, any changes made to the file system since the last checkpoint will be lost. The window of vulnerability can be controlled be setting the checkpointing interval. For example, our current checkpointing interval of 30 seconds means that in the worst case, changes made in the thirty seconds before a crash may be lost.

Ultimately LFS will be able to recover the information written between the last checkpoint and the crash. Using information in the segment summary blocks, LFS can "roll forward" from the last checkpoint, updating metadata structures such as the inode map and indirect blocks. This system will not require the entire file cache to be written on every checkpoint. Recovery will be slightly slower because the portion of the log between the last checkpoint and and the crash must be processed.

4.5. Related Work

LFS borrows ideas from many other systems. This section lists some of these systems. Organizing a file system as an append-only log is not a new idea. Several other file systems, motivated by the advent of write-once media such as optical disks, have used similar mechanisms. Write-once storage managers with random access to files include SWALLOW[8], the Optical File Cabinet[9], and others[10]. These file systems were intended principally for archival storage and not as high-performance file servers. Another way of viewing the LFS design is to see its roots in file systems like Cedar[11] that use logging to improve write performance and recovery time. The difference between LFS and other logging systems is that a read-optimized copy of the data is not kept and hence a copy does not need to be updated. The read-optimized format is not needed because reads are infrequent and the log's format is already well optimized for most file reads. The writing of several files to disk in one operation is much like the concept of **group commit**[12] found in database literature. Using group commit, database systems such as IBM's FASTPATH[13] delay writing so that several transactions can be committed in a single I/O.

An interesting related area of research is main-memory database systems. The metrics used to evaluate these systems are checkpoint overhead (flushing the dirty blocks to disk) and crash recovery speed. These metrics are similar to the design goals of write-optimized file systems. Main-memory database designs use logging and large asynchronous writes[12, 14].

5. Performance of LFS

This section presents the performance of the LFS implementation running under the Sprite operating system. For comparison purposes, the same tests were also run under SunOS 4.0.3 using Sun's version of the BSD fast file system. The benchmarks were designed to demonstrate the important features of LFS; they were not meant to provide a thorough comparison between LFS and the SunOS file system.

The machine used for the test was a Sun-4/260 (16.6Mhz SPARC CPU) equipped with 32 megabytes of memory, a Sun SCSI3 HBA, and an WREN IV disk (1.3 MBytes/sec maximum transfer bandwidth, 17.5 milliseconds average seek time). For both LFS and SunOS, the disk was formatted with a file system having around 300 megabytes of usable storage. An eight-kilobyte block size was used by SunOS while LFS used a four-kilobyte block size and a one-megabyte segment size. The system was running multiuser but was otherwise quiescent during the test.

5.1. Small-file I/O

The first test measures the creation, reading, and deletion speeds of small files. The test consisted of creating 10 megabytes of small files, followed by flushing the file cache and reading all the files from disk. After reading all the files, they were deleted. Figure 3 presents results of test runs with files of size one kilobyte and ten kilobytes. The results are normalized to files per second created, read, and deleted.

The asynchronous file creation and deletion of LFS permits substantial gains in performance over the synchronous SunOS file system. SunOS performs small synchronous disk I/Os for each creation and deletion while LFS packs many changes into segments and pushs them to disk in megabyte transfers. This accounts for much of LFS's order-of-magnitude faster small file creation and deletion. While the SunOS performance is related to the disk access time, the LFS is CPU-bound on these tests. LFS performance will scale with CPU speeds until it is ultimately limited by the disk sequential-write

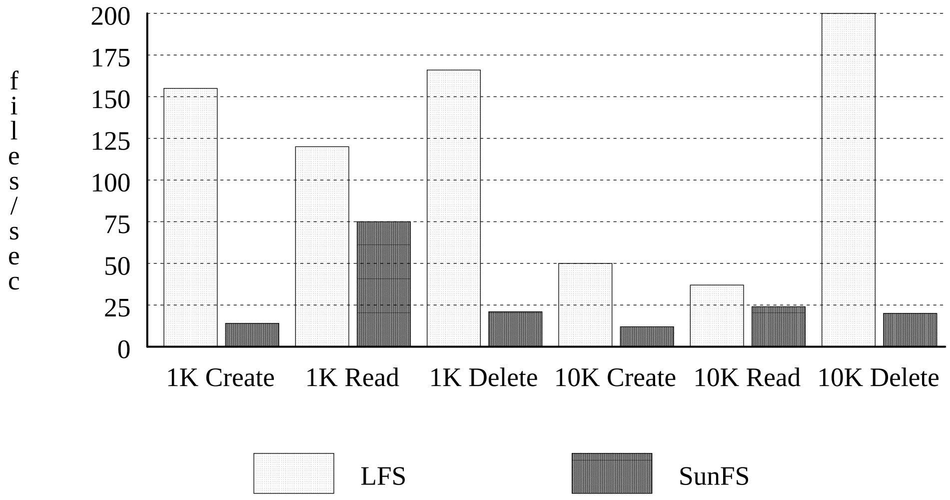

Figure 3 — Small file I/O test.
Measurements of creating, reading, and deleting many 1K and 10K files using LFS and the SunOS file system. The creation phase of the test measured the speed at which 10000 one-kilobyte and 1000 ten-kilobyte files could be created. Following the creation, the file cache was flushed and all the files were read (in the same order as they were created). Finally, we measured the speed at which the files could be deleted. All performance measurements are presented in number of files created, read, or deleted per second.

bandwidth. Because the files are packed tightly together in segments, the read performance of LFS is excellent. Update-in-place file systems such as SunOS must spread files out to support file growth, thus hurting the read performance during these tests.

5.2. Large-file I/O

The second test measures the performance of writing and reading a 100-megabyte file from a newly created file system. The test consisted of five stages: writing a 100-megabyte file sequentially, reading the file sequentially, writing 100 megabytes randomly to the file, reading 100 megabytes randomly from the file, and rereading the file sequentially again. The test program used an eight-kilobyte request size. Figure 4 presents the transfer rates in kilobytes/second for each test. As expected, LFS's write performance comes close to the maximum write bandwidth of the disk. Unlike the SunOS file system, LFS's write bandwidth is independent of how the file is written. LFS especially excels on the random write test because the random file writes become sequential writes when packed into segments. The SunOS file system must do random I/O to update the file in place. The random write rate for LFS is greater than the sequential write rate because the random I/Os were not unique, thus allowing data to be overwritten in the file cache. Both SunOS and Sprite contain sophisticated file cache implementations that grow to use the memory not being used by programs. During the tests, around 15 megabytes of memory were being used as a file cache.

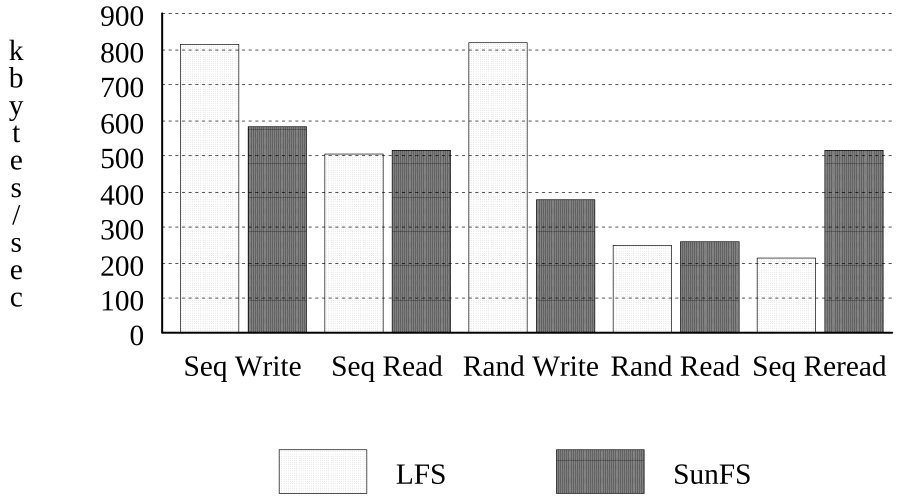

Figure 4 — Large file I/O test.
Transfer rates for reading and writing an 100 megabytes file. The figure shows the rate in kilobytes per second to create and write 100 megabytes sequentially, read 100 megabytes sequentially, write 100 megabytes randomly to the file, and read 100 megabytes randomly from the file. The final test was to read 100 megabytes sequentially after randomly writing the file.

For the sequential write test, both file systems allocated the files sequentially on disk and hence had similar read rates. The file layout differed between the file systems after the random writes were performed. SunOS updates the file in place while LFS writes the file to disk in the order the blocks are written. When the file is randomly read, both file systems must do random disk I/O to process the requests and thus they have equivalent read performance. The read performance of the file systems differed when the file was read sequentially after the random updates. For SunOS, the blocks were sequential on disk while LFS had to do random disk reads to fetch the blocks. This example demonstrates file access patterns in which conventional update-in-place file systems will have better disk read performance than LFS. LFS's sequential read performance can be reduced if the read access pattern is different than the write access pattern. An example for this would be sequentially reading a randomly written file. Note that the performance reduction only occurs in reads that miss in the file cache and would be satisfied sequentially by a conventional file system and not by LFS. Randomly reading a large sequentially written file will cause random I/O in both file systems.

5.3. Cleaning cost

During the tests described above, no segment cleaning was done in LFS. File I/O has traditionally been very bursty so we hope that much of the cleaning can be done using the idle cycles of the disk subsystem. This section examines the cost of segment cleaning and its effect on sustained performance.

The cost of segment cleaning is directly related to the utilization, or number of live blocks, in the segments being cleaned. Segments with no live blocks have no cost while full segments are expensive to clean and yield almost no free space. To evaluate the impact of cleaning on LFS performance, the rate at which bytes can be cleaned was measured for segments with varying utilization. The results of these tests are displayed in Figure 5. The varying utilization was generated by creating many one-kilobyte files, deleting some fixed percentage of them, and measuring the rate at which LFS could clean the fragmented segments. As expected, the more utilized the segment was, the longer it took to clean and the lower the rate at which clean segments could be generated. It is clear that cleaning highly utilized segments will significantly reduce the write-throughput of LFS.

Note that the x-axis of Figure 5 is the average utilization of the segments at cleaning time and not the overall disk-space utilization. For example, using 80% of the disk does not imply that the system will have to clean segments with 80% utilization. For non-synthetic workloads, segment utilization will form a distribution having a mean equal to the overall disk utilization. The shape and variance of the distribution are controlled by many factors including; file system access patterns such as amount of locality in file updates, LFS segment layout algorithms, and segment cleaning algorithms. It is currently not known what the segment distribution looks like for nonsynthetic workloads. The test presented in this section measured a worst case scenario for LFS. Under non-synthetic workloads it would be highly unlikely that all of the segments would have the same fragmentation. The write performance of LFS will be controlled by how well LFS can hide cleaning cost during idle cycles. If cleaning can not be hidden, the question will be how full LFS can allow the disk to become and still keep the cleaning cost down.

6. Conclusion

We have presented a file system that attacks the I/O bottleneck problem by accessing the disk asynchronously and sequentially. The file system is structured as append-

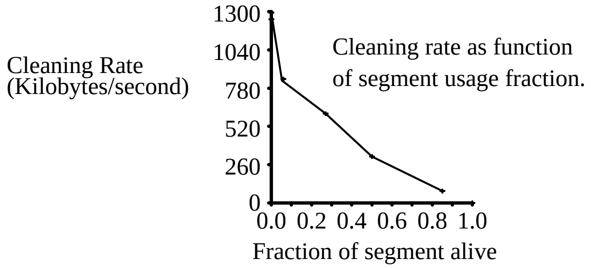

Figure 5 — LFS segment cleaning rate.
Measured rate of segment cleaning as a function of segment utilization. The rate is presented in kilobytes per second that clean segments can be generated.

only segmented log for high write performance and fast crash recovery. File system metadata structures are maintained to allow high read performance. A segment cleaning operation is supported keep segments available for writing and to support high average disk space utilization.

Although LFS has been implemented and is doing very well on micro-benchmarks, the real test of a file system is its performance over months and years of use. As of this writing LFS has not been subjected to a "real" workload for extended periods of time. It is from these workloads that the overheads due to cleaning can be evaluated. The immediate plans for LFS include placing it in continuous use by the Sprite user community and examining its performance.

7. Acknowledgments

The work described here was supported in part by the National Science Foundation under grant CCR-8900029, and in part by the National Aeronautics and Space Administration and the Defense Advanced Research Projects Agency under contract NAG2-591. Diane Greene, Mary Baker, and John Hartman provided helpful comments on drafts of this paper.

References

1. John K. Ousterhout and Fred Douglis, ‘‘Beating the I/O Bottleneck: A Case for Log-structured File Systems,’’ UCB/CSD 88/467, Computer Science Division (EECS), University of California, Berkeley, Berkeley, CA (October 1988).
2. John K. Ousterhout , Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch, ‘‘The Sprite Network Operating System,’’ *IEEE Computer* **21**(2) pp. 23-36 (1988).
3. David A. Patterson , Garth Gibson , and Randy H. Katz, ‘‘A Case for Redundant Arrays of Inexpensive Disks (RAID),’’ *ACM SIGMOD 88*, pp. 109-116 (Jun 1988).
4. Marshall K. McKusick , ‘‘A Fast File System for Unix,’’ *Transactions on Computer Systems* **2**(3) pp. 181-197 (1984).
5. John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson, ‘‘A Trace-Driven Analysis of the Unix 4.2 BSD File System,’’ *Proceedings of the 10th Symposium on Operating System Principles*, pp. 15-24 ACM, (1985).
6. D. Ungar, ‘‘Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm,’’ *Proceedings of the Software Engineering Symposium on Practical Software Development Environments*, pp. 157-167 (Apr 1984).
7. Jim Gray, ‘‘Notes on Data Base Operating Systems,’’ pp. Springer-Verlag in *Operating Systems, An Advanced Course*, (1979).
8. D. Reed and Liba Svobodova, ‘‘SWALLOW: A Distributed Data Storage System for a Local Network,’’ *Local Networks for Computer Communications*, pp. 355-373 North-Holland, (1981).
9. Jason Gait, ‘‘The Optical File Cabinet: A Random Access File System for Write-Once Optical Disks,’’ *IEEE Computer* **21**(6) pp. 11-22 (1988).

10. Ross S. Finlayson and David R. Cheriton, ''Log Files: An Extended File Service Exploiting Write-Once Storage,'' *Proceedings of the Eleventh Symposium on Operating System Principles*, pp. 129-148 (November 1987).

11. Robert B. Hagmann, ''Reimplementing the Cedar File System Using Logging and Group Commit,'' *Proceedings of the 11th Symposium on Operating Systems Principles*, pp. 155-162 (November 1987).

12. David J. DeWitt, Randy H. Katz, Frank Olken, L. D. Shapiro, Mike R. Stonebraker, and David Wood, ''Implementation Techniques for Main Memory Database Systems,'' *Proceedings of SIGMOD 1984*, pp. 1-8 (June 1984).

13. IBM, ''IBM IMS Version 1 Release 1.5 Fast Path Feature Description and Design Guide,'' G320-5775, IBM World Trade Systems Centers (1979).

14. Robert B. Hagmann, ''A Crash Recovery Scheme for a Memory-Resident Database System,'' *IEEE Transaction on Computers* **C-35**(9)(September 1986).