An Implementation of a Log-Structured File System for UNIX

Margo Seltzer -- Harvard University
Keith Bostic -- University of California, Berkeley
Marshall Kirk McKusick -- University of California, Berkeley
Carl Staelin -- Hewlett-Packard Laboratories

ABSTRACT

Research results [ROSE91] suggest that a log-structured file system (LFS) offers the potential for dramatically improved write performance, faster recovery time, and faster file creation and deletion than traditional UNIX file systems. This paper presents a redesign and implementation of the Sprite [ROSE91] log-structured file system that is more robust and integrated into the vnode interface [KLEI86]. Measurements show its performance to be superior to the 4BSD Fast File System (FFS) in a variety of benchmarks and not significantly less than FFS in any test. Unfortunately, an enhanced version of FFS (with read and write clustering) [MCVO91] provides comparable and sometimes superior performance to our LFS. However, LFS can be extended to provide additional functionality such as embedded transactions and versioning, not easily implemented in traditional file systems.

1. Introduction

Early UNIX file systems used a small, fixed block size and made no attempt to optimize block placement [THOM78]. They assigned disk addresses to new blocks as they were created (preallocation) and wrote modified blocks back to their original disk addresses (overwrite). In these file systems, the disk became fragmented over time so that new files tended to be allocated randomly across the disk, requiring a disk seek per file system read or write even when the file was being read sequentially.

The Fast File System (FFS) [MCKU84] dramatically increased file system performance. It increased the block size, improving bandwidth. It reduced the number and length of seeks by placing related information close together on the disk. For example, blocks within files were allocated on the same or a nearby cylinder. Finally, it incorporated rotational disk positioning to reduce delays between accessing sequential blocks.

The factors limiting FFS performance are synchronous file creation and deletion and seek times between I/O requests for different files. The synchronous I/O for file creation and deletion provides file system disk data structure recoverability after failures. However, there exist alternative solutions such as NVRAM hardware [MORA90] and logging software [KAZA90]. In a UNIX environment, where the vast majority of files are small [OUST85] [BAKE91], the seek times between I/O requests for different files can dominate. No solutions to this problem currently exist in the context of FFS.

The log-structured file system, as proposed in [OUST89], attempts to address both of these problems. The fundamental idea of LFS is to improve file system performance by storing all file system data in a single, continuous log. Such a file system is optimized for writing, because no seek is required between writes. It is also optimized for reading files written in their entirety over a brief period of time (as is the norm in UNIX systems), because the files are placed contiguously on disk. Finally, it provides temporal locality, in that it is optimized for accessing files that were created or modified at approximately the same time.

The write-optimization of LFS has the potential for dramatically improving system throughput, as large main-memory file caches effectively cache reads, but do little to improve write performance [OUST89]. The goal of the Sprite log-structured file system (Sprite-LFS) [ROSE91] was to design and implement an LFS that would provide acceptable read performance as well as improved write performance. Our goal is to build on the Sprite-LFS work, implementing a new version of LFS that provides the same recoverability guarantees as FFS, provides performance comparable to or better than FFS, and is well-integrated into a production quality UNIX system.

This paper describes the design of log-structured file systems in general and our implementation in particular, concentrating on those parts that differ from the Sprite-LFS implementation. We compare the performance of our implementation of LFS (BSD-LFS) with FFS using a variety of benchmarks.

2. Log-Structured File Systems

There are two fundamental differences between an LFS and a traditional UNIX file system, as represented by FFS; the on-disk layout of the data structures and the recovery model. In this section we describe the key structural elements of an LFS, contrasting the data structures and recovery to FFS. The complete design and implementation of Sprite-LFS can be found in [ROSE92]. Table 1 compares key differences between FFS and LFS. The reasons for these differences will be described in detail in the following sections.

2.1. Disk Layout

In both FFS and LFS, a file's physical disk layout is described by an index structure (*inode*) that contains the disk addresses of some *direct*, *indirect*, *doubly indirect*, and *triply indirect blocks*. Direct blocks contain data, while indirect blocks contain disk addresses of direct blocks, doubly indirect blocks contain disk addresses of indirect blocks, and triply indirect blocks contain disk addresses of doubly indirect blocks. The inodes and single, double and triple indirect blocks are referred to as ''meta-data'' in this paper.

The FFS is described by a *superblock* that contains file system parameters (block size, fragment size, and file system size) and disk parameters (rotational delay, number of sectors per track, and number of cylinders). The superblock is replicated throughout the file system to allow recovery from crashes that corrupt the primary copy of the superblock. The disk is statically partitioned into cylinder groups, typically between 16 and 32 cylinders to a group. Each group contains a fixed number of inodes (usually one inode for every two kilobytes in the group) and bit maps to record inodes and data blocks available for allocation. The inodes in a cylinder group reside at fixed disk addresses, so that disk addresses may be computed from inode numbers. New blocks are allocated to optimize for sequential file access. Ideally, logically sequential blocks of a file are allocated so that no seek is required between two consecutive accesses. Because data blocks for a file are typically accessed together, the FFS policy routines try to place data blocks for a file in the same cylinder group, preferably at rotationally optimal positions in the same cylinder. Figure 1 depicts the physical layout of FFS.

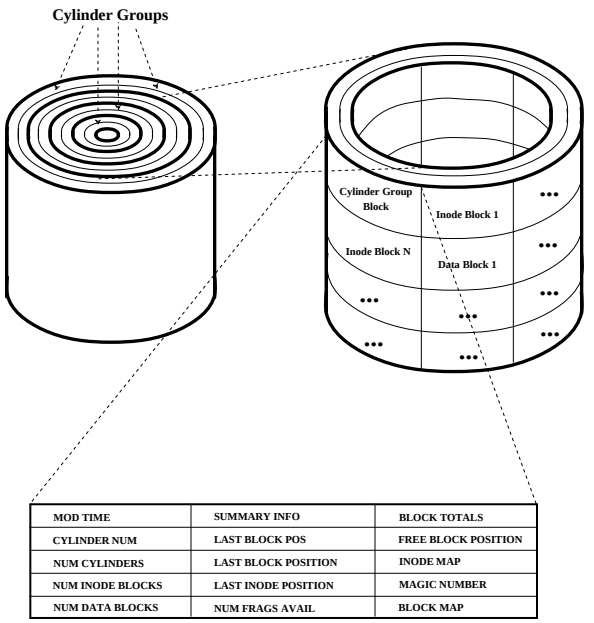

Figure 1: Physical Disk Layout of the Fast File System. The disk is statically partitioned into cylinder groups, each of which is described by a cylinder group block, analogous to a file system superblock. Each cylinder group contains a copy of the superblock and allocation information for the inodes and blocks within that group.

Task	FFS	LFS
Assign disk addresses	block creation	segment write
Allocate inodes	fixed locations	appended to log
Maximum number of inodes	statically determined	grows dynamically
Map inode numbers to disk addresses	static address	lookup in inode map
Maintain free space	bit maps	cleaner segment usage table
Make file system state consistent	fsck	roll-forward
Verify directory structure	fsck	background checker

Table 1: Comparison of File System Characteristics of FFS and LFS.

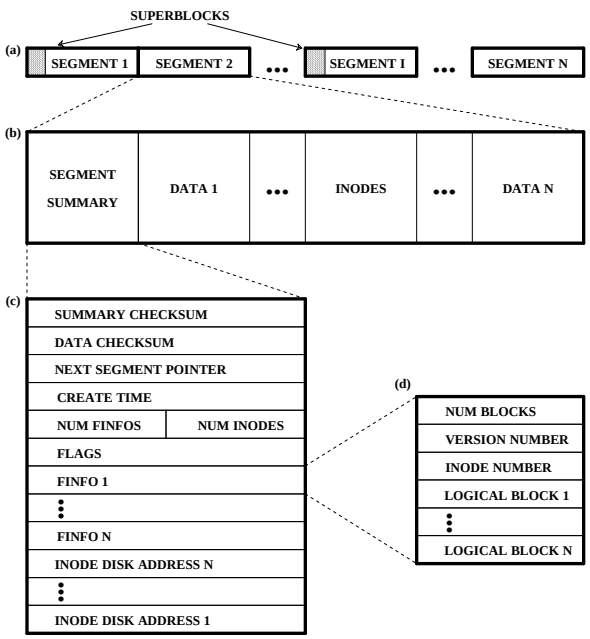

Figure 2: A Log-Structured File System. A file system is composed of **segments** as shown in Figure (a). Each segment consists of a summary block followed by data blocks and inode blocks (b). The **segment summary** contains checksums to validate both the segment summary and the data blocks, a timestamp, a pointer to the next segment, and information that describes each file and inode that appears in the segment (c). Files are described by **FINFO** structures that identify the inode number and version of the file (as well as each block of that file) located in the segment (d).

LFS is a hybrid between a sequential database log and FFS. It performs all writes sequentially, like a database log, but incorporates the FFS index structures into this log to support efficient random retrieval. In an LFS, the disk is statically partitioned into fixed size segments, typically one-half megabyte. The logical ordering of these segments creates a single, continuous log.

An LFS is described by a superblock similar to the one used by FFS. When writing, LFS gathers many dirty pages and prepares to write them to disk sequentially in the next available segment. At this time, LFS sorts the blocks by logical block number, assigns them disk addresses, and updates the meta-data to reflect their addresses. The updated meta-data blocks are gathered with the data blocks, and all are written to a segment. As a result, the inodes are no longer in fixed locations, so, LFS requires an additional data structure, called the *inode map* [ROSE90], that maps inode numbers to disk addresses.

Since LFS writes dirty data blocks into the next available segment, modified blocks are written to the disk in different locations than the original blocks. This space reallocation is called a ‘‘no-overwrite’’ policy, and it necessitates a mechanism to reclaim space resulting from deleted or overwritten blocks. The *cleaner* is a garbage collection process that reclaims space from the file system by reading a segment, discarding ‘‘dead’’ blocks (blocks that belong to deleted files or that have been superseded by newer blocks), and appending any ‘‘live’’ blocks. For the cleaner to determine which blocks in a segment are ‘‘live,’’ it must be able to identify each block in a segment. This determination is done by including a summary block in each segment that identifies the inode and logical block number of every block in the segment. In addition, the kernel maintains a *segment usage table* that shows the number of ‘‘live’’ bytes and the last modified time of each segment. The cleaner uses this table to determine which segments to clean [ROSE90]. Figure 2 shows the physical layout of the LFS.

While FFS flushes individual blocks and files on demand, the LFS must gather data into segments. Usually, there will not be enough dirty blocks to fill a complete segment [BAKE92], in which case LFS writes *partial segments*. A physical segment contains one or more partial segments. For the remainder of this paper, *segment* will be used to refer to the physical partitioning of the disk, and *partial segment* will be used to refer to a unit of writing. Small partial segments most commonly result from NFS operations or *fsync*(2) requests, while writes resulting from the *sync*(2) system call or system memory shortages typically form larger partials, ideally taking up an entire segment. During a *sync*, the inode map and segment usage table are also written to disk, creating a *checkpoint* that provides a stable point from which the file system can be recovered in case of system failure. Figure 3 shows the details of allocating three files in an LFS.

2.2. File System Recovery

There are two aspects to file system recovery: bringing the file system to a physically consistent state and verifying the logical structure of the file system. When FFS or LFS add a block to a file, there are several different pieces of information that may be modified: the block itself, the inode, the free block map, possibly indirect blocks, and the location of the last allocation. If the system crashes during the addition, the file system is likely be left in a physically inconsistent state. Furthermore, there is currently no way for FFS to localize inconsistencies. As a result, FFS must rebuild the entire file system state, including cylinder group bit maps and meta-data. At the same

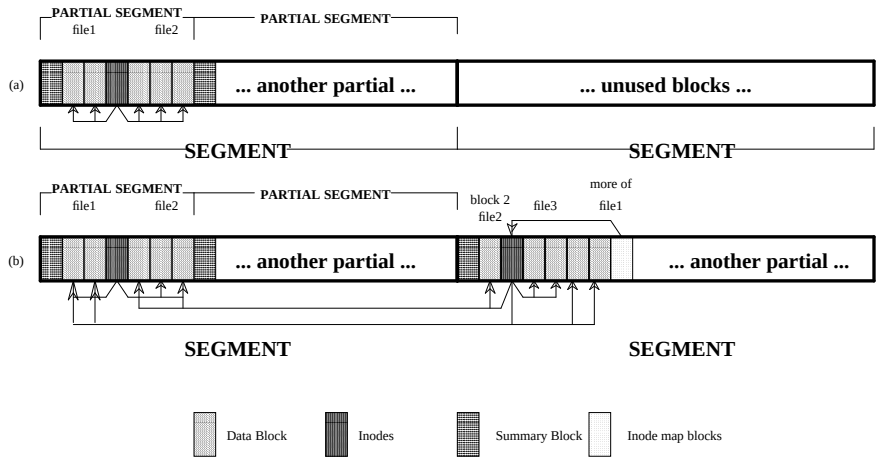

Figure 3: File Allocation in a Log-Structured File System. In figure (a), two files have been written, file1 and file2. Each has an index structure in the meta-data block that is allocated after it on disk. In figure (b), the middle block of file2 has been modified. A new version of it is added to the log, as well as a new version of its meta-data. Then file3 is created, causing its blocks and meta-data to be appended to the log. Next, file1 has two more blocks appended to it, causing the blocks and a new version of file1's meta-data to be appended to the log. On checkpoint, the *inode map* containing pointers to the meta-data blocks, is written.

time, FFS verifies the directory structure and all block pointers within the file system. Traditionally, *fsck*(8) is the agent that performs both of these functions.

In contrast to FFS, LFS writes only to the end of the log and is able to locate potential inconsistencies and recover to a consistent physical state quickly. This part of recovery in LFS is more similar to standard database recovery [HAER83] than to *fsck*. It consists of two parts: initializing all the file system structures from the most recent checkpoint and then ''rolling forward'' to incorporate any modifications that occurred subsequently. The roll forward phase consists of reading each segment after the checkpoint in time order and updating the file system state to reflect the contents of the segment. The next segment pointers in the segment summary facilitate reading from the last checkpoint to the end of the log, the checksums are used to identify valid segments, and the timestamps are used to distinguish the partial segments written after the checkpoint and those written before which have been reclaimed. The file and block numbers in the FINFO structures are used to update

Phase I	Traverse inodes Validate all block pointers. Record inode state (allocated or unallocated) and file type for each inode. Record inode numbers and block addresses of all directories.
Phase II	Sort directories by disk address order. Traverse directories in disk address order. Validate ''.''. Record ''..''. Validate directories' contents, type, and link counts. Recursively verify ''..''.
Phase III	Attach any unresolved ''..'' trees to lost+found. Mark all inodes in those trees as ''found''.
Phase IV	Put any inodes that are not ''found'' in lost+found. Verify link counts for every file.
Phase V	Update bit maps in cylinder groups.

Table 2: Five Phases of *fsck*.

the inode map, segment usage table, and inodes making the blocks in the partial segment extant. As is the case for database recovery, the recovery time is proportional to the interval between file system checkpoints.

While standard LFS recovery quickly brings the file system to a physically consistent state, it does not provide the same guarantees made by *fsck*. When *fsck* completes, not only is the file system in a consistent state, but the directory structure has been verified as well. The five passes of *fsck* are summarized in Table 2. For LFS to provide the same level of robustness as FFS, LFS must make many of the same checks. While LFS has no bit maps to rebuild, the verification of block pointers and directory structure and contents is crucial for the system to recover from media failure. This recovery will be discussed in more detail in Section 3.4.

3. Engineering LFS

While the Sprite-LFS implementation was an excellent proof of concept, it had several deficiencies that made it unsuitable for a production environment. Our goal was to engineer a version of LFS that could be used as a replacement for FFS. Some of our concerns were as follows:

1. Sprite-LFS consumes excessive amounts of memory.
2. Write requests are successful even if there is insufficient disk space.
3. Recovery does nothing to verify the consistency of the file system directory structure.
4. Segment validation is hardware dependent.
5. All file systems use a single cleaner and a single cleaning policy.
6. There are no performance numbers that measure the cleaner overhead.

The earlier description of LFS focused on the overall strategy of log-structured file systems. The rest of Section 3 discusses how BSD-LFS addresses the first five problems listed above. Section 4 addresses the implementation issues specific to integration in a BSD framework, and Section 5 presents the performance analysis. In most ways, the logical framework of Sprite-LFS is unchanged. We have kept the segmented log structure and the major support structures associated with the log, namely the inode map, segment usage table, and cleaner. However, to address the problems described above and to integrate LFS into a BSD system, we have altered nearly all of the details of implementation, including a few fundamental design decisions. Most notably, we have moved the cleaner into user space, eliminated the directory operation log, and altered the segment layout on disk.

3.1. Memory Consumption

Sprite-LFS assumes that the system has a large physical memory and ties down substantial portions of it. The following storage is reserved:

Two 64K or 128K staging buffers
Since not all devices support scatter/gather I/O, data is written in buffers large enough to allow the maximum transfer size supported by the disk controller, typically 64K or 128K. These buffers are allocated per file system from kernel memory.

One cleaning segment
One segment's worth of buffer cache blocks per file system are reserved for cleaning.

Two read-only segments
Two segments' worth of buffer cache blocks per file system are marked read-only so that they may be reclaimed by Sprite-LFS without requiring an I/O.

Buffers reserved for the cleaner
Each file system also reserves some buffers for the cleaner. The number of buffers is specified in the superblock and is set during file system creation. It specifies the minimum number of clean buffers that must be present in the cache at any point in time. On the Sprite cluster, the amount of buffer space reserved for 10 commonly used file systems was 37 megabytes.

One segment
This segment (typically one-half megabyte) is allocated from kernel memory for use by the cleaner. Since this one segment is allocated per system, only one file system per system may be cleaned at a time.

The reserved memory described above makes Sprite-LFS a very ‘‘bad neighbor’’ as kernel subsystems compete for memory. While memory continues to become cheaper, a typical laptop system has only three to eight megabytes of memory, and might very reasonably expect to have three or more file systems.

BSD-LFS greatly reduces the memory consumption of LFS. First, BSD-LFS does not use separate buffers for writing large transfers to disk, instead it uses the regular buffer cache blocks. For disk controllers that do not coalesce contiguous reads, we use 64K staging buffers (briefly allocated from the regular kernel memory pool) to do transfers. The size of the staging buffer was set to the minimum of the maximum transfer sizes for currently supported disks. However, simulation results in [CAR92] show that for

current disks, the write size minimizing the read response time is typically about two tracks; two tracks is close to 64 kilobytes for the disks on our systems.

Secondly, rather than reserving read-only buffers, we initiate segment writes when the number of dirty buffers crosses a threshold. That threshold is currently measured in available buffer headers, not in physical memory, although systems with an integrated buffer cache and virtual memory will have simpler, more straight-forward mechanisms.

Finally, the cleaner is implemented as a user space process. This approach means that it requires no dedicated memory, competing for virtual memory space with the other processes.

3.2. Block Accounting

Sprite-LFS maintains a count of the number of disk blocks available for writing, i.e. the real number of disk blocks that do not contain useful data. This count is decremented when blocks are actually written to disk. This approach implies that blocks can be successfully written to the cache but fail to be written to disk if the disk becomes full before the blocks are actually written. Even if the disk is not full, all available blocks may reside in uncleaned segments and new data cannot be written. To prevent the system from deadlocking or losing data in these cases, BSD-LFS uses two forms of accounting.

The first form of block accounting is similar to that maintained by Sprite-LFS. BSD-LFS maintains a count of the number of disk blocks that do not contain useful data. It is decremented whenever a new block is created in the cache. Since many files die in the cache [BAKE91], this number is incremented whenever blocks are deleted, even if they were never written to disk.

The second form of accounting keeps track of how much space is currently available for writing. This space is allocated as soon as a dirty block enters the cache, but is not reclaimed until segments are cleaned. This count is used to initiate cleaning. If an application attempts to write data, but there is no space currently available for writing, the write will sleep until space is available. These two forms of accounting guarantee that if the operating system accepts a write request from the user, barring a crash, it will perform the write.

Accounting for the actual disk space required is difficult because inodes are not written into dirty buffers and segment summaries are not created until segments are written. Every time an inode is modified in the inode cache, a count of inodes to be written is incremented. When blocks are dirtied, the number of available disk blocks is decremented. To decide if there is enough disk space to allow another write into the cache, the number of segment summaries necessary to write what is in the cache is computed, added to the number of inode blocks necessary to write the dirty inodes and compared to the amount of space available on the disk. To create more available disk space, either the cleaner must run or dirty blocks in the cache must be deleted.

3.3. Segment Structure and Validation

Sprite-LFS places segment summary blocks at the end of segments trusting that if the write containing the segment summary is issued after all other writes in a partial segment, the presence of the segment summary validates the partial segment. This approach requires two assumptions: the disk controller will not reorder the write requests and the disk writes the contents of a buffer in the order presented. Since controllers often reorder writes and reduce rotational latency by beginning track writes anywhere on the track, we felt that BSD-LFS could not make these assumptions. We build segments from front to back, placing the segment summary at the beginning of each segment as shown in Figure 4. We compute a checksum across four bytes of each block in the partial segment, store it in the segment summary, and use this to verify that a partial segment is valid. This approach avoids write-ordering constraints and allows us to write multiple partial segments without an intervening seek or rotation. We do not yet have reason to believe that our checksum is insufficient, however, if it is, *patch tables* can be used to guarantee that any missing

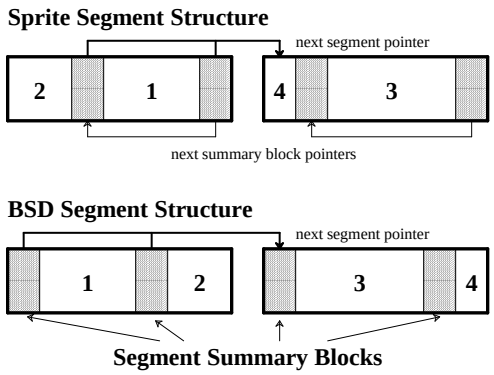

Figure 4: Partial Segment Structure Comparison Between Sprite-LFS and BSD-LFS. The numbers in each partial show the order in which the partial segments are created. Sprite-LFS builds segments back to front, chaining segment summaries. BSD-LFS builds segments front to back. After reading a segment summary block, the location of the next segment summary block can be easily computed.

sector can be detected during roll-forward, at the expense of a bit per disk sector stored in the segment usage table and segment summary blocks.

3.4. File System Verification

Fast recovery from system failure is desirable, but reliable recovery from media failure is necessary. Consequently, the BSD-LFS system provides two recovery strategies. The first quickly rolls forward from the last checkpoint, examining data written between the last checkpoint and the failure. The second does a complete consistency check of the file system to recover lost or corrupted data, due to the corruption of bits on the disk or errant software writing bad data to the disk. This check is similar to the functionality of *fsck*, the file system checker and recovery agent for FFS, and like *fsck*, it takes a long time to run.

As UNIX systems spend a large fraction of their time, while rebooting, in file system checks, the speed at which LFS is able to recover its file systems is considered one of its major advantages. However, FFS is an extremely robust file system. In the standard 4BSD implementation, it is possible to clear the root inode and recover the file system automatically with *fsck*(8). This level of robustness is necessary before users will accept LFS as a file system in traditional UNIX environments.

In terms of recovery, the advantage of LFS is that writes are localized, so the file system may be recovered to a physically consistent state very quickly. The BSD-LFS implementation permits LFS to recover quickly, and applications can start running as soon as the roll-forward has been completed, while basic sanity checking of the file system is done in the background. There is the obvious problem of what to do if the sanity check fails. It is expected that the file system will be forcibly made read-only, fixed, and then once again write enabled. These events should have a limited effect on users as it is unlikely to ever occur and is even more unlikely to discover an error in a file currently being written by a user, since the opening of the file would most likely have already caused a process or system failure. Of course, the root file system must always be completely checked after every reboot, in case a system failure corrupted it.

3.5. The Cleaner

In Sprite-LFS the cleaner is part of the kernel and implements a single cleaning policy. There are three problems with this, in addition to the memory issues discussed in Section 3.1. First, there is no reason to believe that a single cleaning algorithm will work well on all workloads. In fact, measurements in [SELT93b] show that coalescing randomly updated files would improve sequential read performance dramatically. Second, placing the cleaner in kernel-space makes it difficult to experiment with alternate cleaning policies. Third, implementing the cleaner in the kernel forces the kernel to make policy decisions (the cleaning algorithm) rather than simply providing a mechanism. To handle these problems, the BSD-LFS cleaner is implemented as a user process.

The BSD-LFS cleaner communicates with the kernel via system calls and the read-only *ifile*. Those functions that are already handled in the kernel (e.g. translating logical block numbers to disk addresses via *bmap*) are made accessible to the cleaner via system calls. If necessary functionality did not already exist in the kernel (e.g. reading and parsing segment summary blocks), it was relegated to user space.

There may be multiple cleaners, each implementing a different cleaning policy, running in parallel on a single file system. Regardless of the particular policy, the basic cleaning algorithm works as follows:

1. Read one or more target segments.
2. Decide which blocks are still alive.
3. Write live blocks back to the file system.
4. Mark the segment clean.

The *ifile* and four new system calls, summarized in Table 3, provide the cleaner with enough information to implement this algorithm. The cleaner reads the *ifile* to find out the status of segments in the file system and selects segments to clean based on this information. Once a segment is selected, the cleaner reads the segment from the raw partition and uses the first segment summary to find out what blocks reside in that partial segment. It constructs an array of BLOCK_INFO structures (shown in Figure 5) and continues scanning partial segments, adding their blocks to the array. When the entire segment has been read, and all the BLOCK_INFOs constructed, the cleaner calls *lfs_bmapv* which returns the current physical disk address for each BLOCK_INFO. If the disk address is the same as the location of the block in the segment being examined by the cleaner, the block is ‘‘live’’. Live blocks must to be written back into the file system without changing their access or modify times, so the cleaner issues an *lfs_markv* call, which is a special write causing these blocks to be appended into the log without updating the inode times.

Before rewriting the blocks, the kernel verifies that none of the blocks have ‘‘died’’ since the cleaner called *lfs_bmapv*. Once *lfs_markv* begins, only cleaned blocks are written into the log, until *lfs_markv* completes. Therefore, if cleaned blocks die after *lfs_markv* verifies that they are alive, partial segments written after the *lfs_markv* partial segments will reflect that the blocks have died. When *lfs_markv* returns, the cleaner calls *lfs_segclean* to mark the segment

lfs_bmapv	Take an array of inode number/logical block number pairs and return the disk address for each block. Used to determine if blocks in a segment are ‘‘live’’.
lfs_markv	Take an array of inode number/logical block number pairs and append them into the log. This operation is a special purpose write call that rewrites the blocks and inodes without updating the inode’s access or modification times.
lfs_segwait	Causes the cleaner to sleep until a given timeout has elapsed or until another segment is written. This operation is used to let the cleaner pause until there may be more segments available for cleaning.
lfs_segclean	Mark a segment clean. After the cleaner has rewritten all the ‘‘live’’ blocks from a segment, the segment is marked clean for reuse.

Table 3: The System Call Interface for the Cleaner.

clean. Finally, when the cleaner has cleaned enough segments, it calls *lfs_segwait*, sleeping until the specified timeout elapses or a new segment is written into an LFS.

Since the cleaner is responsible for producing free space, the blocks it writes must get preference over other dirty blocks to be written to avoid running out of free space. There are degenerative cases where cleaning a segment can actually consume more space than it frees [SELT93a]. To ensure that the cleaner can always run and eventually generate more free space, normal writing is suspended when the number of clean segments drops to two.

The cleaning simulation results in [ROSE91] show that selection of segments to clean is an important design parameter in minimizing cleaning overhead, and that the cost-benefit policy defined there does extremely well for the simulated workloads. Briefly, each segment is assigned a cleaning *cost* and *benefit*. The *cost* to clean a segment is equal to:

$$1 + utilization$$

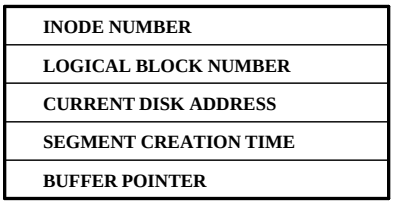

Figure 5: BLOCK_INFO Structure used by the Cleaner. The cleaner calculates the current disk address for each block from the disk address of the segment. The kernel specifies which have been superceded by more recent versions.

where utilization is the fraction of ‘‘live’’ data in the segment. The *benefit* of cleaning a segment is:

$$free\ bytes\ generated * age\ of\ segment$$

where *free bytes generated* is the fraction of ‘‘dead’’ blocks in the segment ($1 - utilization$) and *age of segment* is the time since the most recent modification to any block in that segment. When the file system needs to reclaim space, the cleaner selects the segment with the largest benefit to cost ratio. We retained this policy as the default cleaning algorithm.

Currently the cost-benefit cleaner is the only cleaner we have implemented, but two additional policies are under consideration. The first would run during idle periods and select segments to clean based on coalescing and clustering files. The second would flush blocks in the cache to disk during normal processing even if they were not dirty, if it would improve the locality for a given file. These policies will be analyzed in future work.

4. Implementing LFS in a BSD System

While the last section focused on those design issues that addressed problems in the design of Sprite-LFS, this section presents additional design issues either inherent to LFS or resulting from the integration of an LFS into 4BSD.

4.1. Integration with FFS

The on-disk data structures used by BSD-LFS are nearly identical to the ones used by FFS. This decision was made for two reasons. The first one was that many applications have been written over the years to interpret and analyze raw FFS structures. It is desirable that these tools could continue to function as before, with minor modifications to read the structures from a new location. The second and more important reason was that it was easy and increased the maintainability of the system. A basic LFS implementation,

without cleaner or reconstruction tools, but with *dumpfs*(1) and *newfs*(1) tools, was reading and writing from/to the buffer cache in under two weeks, and reading and writing from/to the disk in under a month. This implementation was done by copying the FFS source code and replacing about 40% of it with new code. The FFS and LFS implementations have since been merged to share common code.

In BSD and similar systems (e.g. SunOS, OSF/1), a file system is defined by two sets of interface functions, *vfs* operations and *vnode* operations [KLEI86]. *Vfs* operations affect entire file systems (e.g. mount, unmount, etc.) while *vnode* operations affect files (open, close, read, write, etc.).

File systems could share code at the level of a vfs or vnode subroutine call, but they could not share the UNIX naming while implementing their own disk storage algorithms. To allow sharing of the UNIX naming, the code common to both the FFS and BSD-LFS was extracted from the FFS code and put in a new, generic file system module (UFS). This code contains all the directory traversal operations, almost all vnode operations, the inode hash table manipulation, quotas, and locking. The common code is used not only by the FFS and BSD-LFS, but by the memory file system [MCKU90] as well. The FFS and BSD-LFS implementations remain responsible for disk allocation and actual I/O.

In moving code from the FFS implementation into the generic UFS area, it was necessary to add seven new *vnode* and *vfs* operations. Table 4 lists the operations that were added to facilitate this integration and explains why they are different for the two file systems.

Vnode Operations	
blkatoff	Read the block at the given offset, from a file. The two file systems calculate block sizes and block offsets differently, because BSD-LFS does not implement fragments.
valloc	Allocate a new inode. FFS must consult and update bit maps to allocate inodes while BSD-LFS removes the inode from the head of the free inode list in the *ifile*.
vfree	Free an inode. FFS must update bit maps while BSD-LFS inserts the inode onto a free list.
truncate	Truncate a file from the given offset. FFS marks bit maps to show that blocks are no longer in use, while BSD-LFS updates the segment usage table.
update	Update the inode for the given file. FFS pushes individual inodes synchronously, while BSD-LFS writes them in a partial segment.
bwrite	Write a block into the buffer cache. FFS does synchronous writes while BSD-LFS puts blocks on a queue for writing in the next segment.
Vfs Operations	
vget	Get a vnode. FFS computes the disk address of the inode while BSD-LFS looks it up in the *ifile*.

Table 4: New Vnode and Vfs Operations. These routines allowed us to share 60% of the original FFS code with BSD-LFS.

4.1.1. Block Sizes

One FFS feature that is not implemented in BSD-LFS is fragments. The original reason FFS had fragments was that, given a large block size (necessary to obtain contiguous reads and writes and to lower the data to meta-data ratio), fragments were required to minimize internal fragmentation (allocated space that does not contain useful data). LFS does not require large blocks to obtain contiguous reads and writes as it sorts blocks in a file by logical block number, writing them sequentially. Still, large blocks are desirable to keep the meta-data to data ratio low. Unfortunately, large blocks can lead to wasted space if many small files are present. Since managing fragments complicates the file system, we decided to allocate progressively larger blocks instead of using a block/fragment combination. This improvement has not yet been implemented but is similar to the restricted buddy simulated in [SELT91].

4.1.2. The Buffer Cache

Prior to the integration of BSD-LFS into 4BSD, the buffer cache had been considered file system independent code. However, the buffer cache contains assumptions about how and when blocks are written to disk. First, it assumes that a single block can be flushed to disk, at any time, to reclaim its memory. There are two problems with this: flushing blocks a single block at a time would destroy any possible performance advantage of LFS, and, because of the modified meta-data and partial segment summary blocks, LFS may require additional memory to write.

Therefore, BSD-LFS needs to guarantee that it can obtain any additional buffers it needs when it writes a segment. To prevent the buffer cache from trying to flush a single BSD-LFS page, BSD-LFS puts its dirty buffers on the kernel LOCKED queue, so that the buffer cache cannot reclaim them. The number of buffers on the locked queue is compared against two variables, the *start write threshold* and *stop access threshold*, to prevent BSD-LFS from using up all the available buffers. This problem can be much more reasonably handled by systems with better integration of the buffer cache and virtual memory.

Second, BSD maintains a logical block cache, hashed by vnode and logical block number. In FFS, since indirect blocks do not have logical block numbers, they are hashed by the vnode of the device (the file that represents the disk partition) and the disk block number. Since LFS does not assign disk addresses until blocks are written to disk, indirect blocks have no valid addresses on which to hash. To solve this problem, the block name space had to incorporate meta-data block numbering. This naming is done by making block addresses be signed integers with negative numbers referencing indirect blocks, while zero and positive numbers reference data blocks. Figure 6 shows how the blocks are numbered. Singly indirect blocks take on the negative of the first data block to which they point. Doubly and triply indirect blocks take the next lower negative number of the singly or doubly indirect block to which they point. This approach makes it simple to traverse the indirect block chains in either direction, facilitating reading a block or creating indirect blocks. Sprite-LFS partitions the ‘‘block name space’’ in a similar fashion. Although it is not possible for BSD-LFS to use FFS meta-data numbering, the reverse is not true. In 4.4BSD, FFS uses the BSD-LFS numbering and the *bmap* code has been moved into the UFS area.

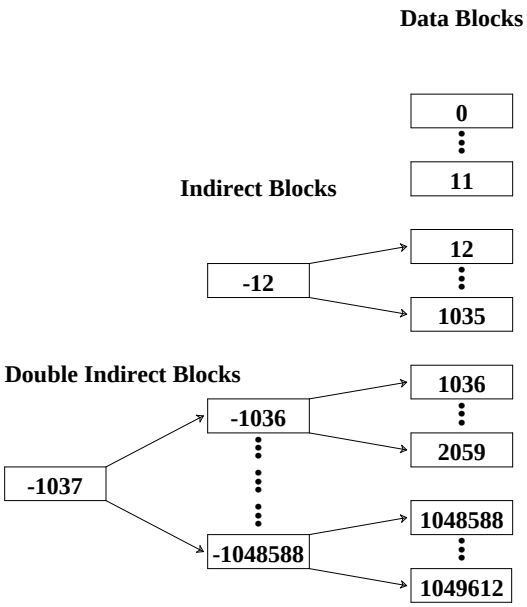

Figure 6: Block-numbering in BSD-LFS. In BSD-LFS, data blocks are assigned positive block numbers beginning with 0. Indirect blocks are numbered with the negative of the first data block that they address. Double and triple indirect blocks are numbered with one less than the first indirect or double indirect block that they address.

4.2. The IFILE

Sprite-LFS maintained the inode map and segment usage table as kernel data structures which are written to disk at file system checkpoints. BSD-LFS places both of these data structures in a read-only regular file, visible in the file system, called the *ifile*. There are three advantages to this approach. First, while Sprite-LFS and FFS limit the number of inodes in a file system, BSD-LFS has no such limitation, growing the *ifile* via the standard file mechanisms. Second, it can be treated identically to other files, in most cases, minimizing the special case code in the operating system. Finally, as is discussed in Section 3.5, we intended to move the cleaner into user space, and the *ifile* is a convenient mechanism for communication between the operating system and the cleaner. A detailed view of the *ifile* is shown in Figure 7.

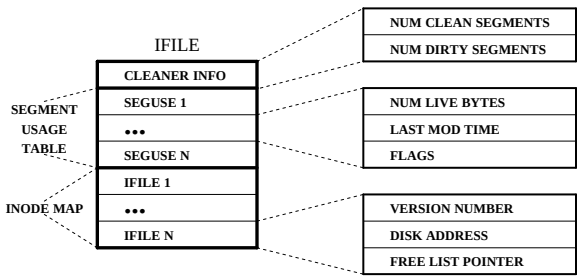

Figure 7: Detail Description of the IFILE. The *ifile* is maintained as a regular file with read-only permission. It facilitates communication between the file system and the cleaner.

Both Sprite-LFS and BSD-LFS maintain disk addresses and inode version numbers in the inode map. The version numbers allow the cleaner to easily identify groups of blocks belonging to files that have been truncated or deleted. Sprite-LFS also keeps the last access time in the inode map to minimize the number of blocks that need to be written when a file system is being used only for reading. Since the access time is eight bytes in 4.4BSD and maintaining

it in the inode map would cause the *ifile* to grow by 67%, BSD-LFS keeps the access time in the inode.

Sprite-LFS clusters inodes in the inode map, and allocates new inodes by picking a starting point and scanning forward sequentially until it finds a free inode. To create a new file, the inode map is searched from the inode entry of the containing directory. If a directory is being created, a random location is chosen. When a directory contains many files this scan is costly. On six Sprite file systems, the average number of entries searched per directory or file creation ranged from 26 to 192, with an average across all the file systems of 94 entries per allocation. BSD-LFS avoids this scan by maintaining a free list of inodes in the inode map.

The segment usage table contains the number of live bytes in and the last modified time of the segment, and is largely unchanged from Sprite-LFS. In order to support multiple and user mode cleaning processes, we have added a set of flags indicating whether the segment is clean, contains a superblock, is currently being written to, or is eligible for cleaning.

4.3. Directory Operations

Directory operations[1] pose a special problem for LFS. Since the basic premise of LFS is that operations can be postponed and coalesced to provide large I/Os, it is counterproductive to retain the synchronous behavior of directory operations. At the same time, if a file is created, filled with data and *fsync*ed, then both the file's data and the directory entry for the file must be on disk. Additionally, the UNIX semantics of directory operations are defined to preserve ordering (i.e. if the creation of file *a* precedes the creation of file *b*, then any post-recovery state of a file system that includes file *b* must include file *a*). We believe this semantic is used in UNIX systems to provide mutual exclusion and other locking protocols[2].

Sprite-LFS preserves the ordering of directory operations by maintaining a directory operation log inside the file system log. Before any directory updates are written to disk, a log entry that describes the directory operation is written. The log information always appears in an earlier segment, or the same segment, as the actual directory updates. At recovery time, this log is read and any directory operations that were not fully completed are rolled forward. Since this approach requires an additional, on-disk data structure, and since LFS is itself a log, we chose a different solution, namely *segment batching*.

Since directory operations affect multiple inodes, we need to guarantee that either both of the inodes and associated changes get written to disk or neither does. BSD-LFS has a unit of atomicity, the partial segment, but it does not have a mechanism that guarantees that all inodes involved in the same directory operation will fit into a single partial segment. Therefore, we introduced a mechanism that allows operations to span partial segments. At recovery, we never roll forward a partial segment if it has an unfinished directory operation and the partial segment that completes the directory operation did not make it to disk.

The requirements for segment batching are defined as follows:

1. If any directory operation has occurred since the last segment was written, the next segment write will append all dirty blocks from the *ifile* (that is, it will be a checkpoint, except that the superblock need not be updated).
2. During recovery, any writes that were part of a directory operation write will be ignored unless the entire write completed. A completed write can be identified if all dirty blocks of the *ifile* and its inode were successfully written to disk.

This definition is essentially a transaction where the writing of the *ifile* inode to disk is the commit operation. In this way, there is a coherent snapshot of the file system at some point after each directory operation. The penalty is that checkpoints are written more frequently in contrast to Sprite-LFS's approach that wrote additional logging information to disk.

The BSD-LFS implementation requires synchronizing directory operations and segment writing. Each time a directory operation is performed, the affected vnodes are marked. When the segment writer builds a segment, it collects vnodes in two passes. In the first pass, all unmarked vnodes (those not participating in directory operations) are collected, and during the second pass those vnodes that are marked are collected. If any vnodes are found during the second pass, this means that there are directory operations present in the current segment, and the segment is marked, identifying it as containing a directory operation. To prevent directory operations from being partially reflected in a segment, no new directory operations are begun while the segment writer is in pass two, and the segment writer cannot begin pass two while any directory operation is in progress.

[1] Directory operations include those system calls that affect more than one inode (typically a directory and a file) and include: create, link, mkdir, mknod, remove, rename, rmdir, and symlink.

[2] We have been unable to find a real example of the ordering of directory operations being used for this purpose and are considering removing it as unnecessary complexity. If you have an example where ordering must be preserved across system failure, please send us email at margo@das.harvard.edu!

When recovery is run, the file system can be in one of three possible states with regard to directory operations:

1. The system shut down cleanly so that the file system may be mounted as is.
2. There are valid segments following the last checkpoint and the last one was a completed directory-operation write. Therefore, all that is required before mounting is to rewrite the superblock to reflect the address of the *ifile* inode and the current end of the log.
3. There are valid segments following the last checkpoint or directory operation write. As in the previous case, the system recovers to the last completed directory operation write and then rolls forward the segments from there to either the end of the log or the first segment beginning a directory operation that is never finished. Then the recovery process writes a checkpoint and updates the superblock.

While rolling forward, two flags are used in the segment summaries: SS_DIROP and SS_CONT. SS_DIROP specifies that a directory operation appears in the partial segment. SS_CONT specifies that the directory operation spans multiple partial segments. If the recovery agent finds a segment with both SS_DIROP and SS_CONT set, it ignores all such partial segments until it finds a later partial segment with SS_DIROP set and SS_CONT unset (i.e. the end of the directory operation write). If no such partial segment is ever found, then all the segments from the initial directory operation on are discarded. Since partial segments are small [BAKE92] this should rarely, if ever, happen.

4.4. Synchronization

To maintain the delicate balance between buffer management, free space accounting and the cleaner, synchronization between the components of the system must be carefully managed. Figure 8 shows each of the synchronization relationships. The cleaner is given precedence over all other processing in the system to guarantee that clean segments are available if the file system has space. It has its own event variable on which it waits for new work (*lfs_allclean_wakeup*). The segment writer and user processes will defer to the cleaner if the disk system does not have enough clean space. A user process detects this condition when it attempts to write a block but the block accounting indicates that there is no space available. The segment writer detects this condition when it attempts to begin writing to a new segment and the number of clean segments has reached two.

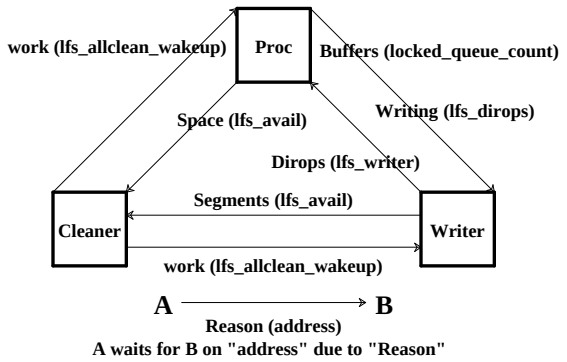

Figure 8: Synchronization Relationships in BSD-LFS. The cleaner has precedence over all components in the system. It waits on the *lfs_allclean_wakeup* condition and wakes the segment writer or user processes using the *lfs_avail* condition. The segment writer and user processes maintain directory operation synchronization through the *lfs_dirop* and *lfs_writer* conditions. User processes doing writes wait on the *locked_queue_count* when the number of dirty buffers held by BSD-LFS exceeds a system limit.

In addition to cleaner synchronization, the segment writer and user processes synchronize on the the availability of buffer headers. When the number of buffer headers drops below the *start write threshold* a segment write is initiated. If a write request would push the number of available buffer headers below the *stop access threshold*, the writing process waits until a segment write completes, making more buffer headers available. Finally, there is the directory operation synchronization. User processes wait on the *lfs_dirop* condition and the segment writer waits on *lfs_writer* condition.

4.5. Minor Modifications

There are a few additional changes to Sprite-LFS. To provide more robust recovery we replicate the superblock throughout the file system, as in FFS. Since the file system meta-data is stored in the *ifile*, we have no need for separate checkpoint regions, and simply store the disk address of the *ifile* inode in the superblock. Note that it is not necessary to keep a duplicate *ifile* since it can be reconstructed from segment summary information, if necessary.

5. Performance Measurements

This chapter compares the performance of the redesigned log-structured file system to more traditional file systems on a variety of benchmarks based on real workloads. The new log-structured file system was written in November of 1991 and was left largely untouched until late spring 1992 and is a completely

untuned implementation. While design decisions took into account the expected performance impact, at this point there is little empirical evidence to support those decisions.

The file systems against which LFS is compared are the regular fast file system (FFS), and an enhanced version of FFS similar to that described in [MCVO91], referred to as EFS for the rest of this paper.

5.1. Extent-like Performance from FFS

EFS provides extent-based file system behavior without changing the underlying structures of FFS, by allocating blocks sequentially on disk and clustering multiple block requests. FFS is parameterized by a variable called *maxcontig* that specifies how many logically sequential disk blocks should be allocated contiguously. When maxcontig is large (equal to a track), FFS does what is essentially track allocation. In EFS, sequential dirty buffers are accumulated in the cache, and when an extent's worth (i.e. *maxcontig* blocks) have been collected, they are bundled together into a cluster, providing extent-based writing.

To provide extent-based reading, the interaction between the buffer cache and the disk was modified. Typically, before a block is read from disk, the *bmap* routine is called to translate logical block addresses to physical disk block addresses. The block is then read from disk and the next block is requested. Since I/O interrupts are not handled instantaneously, the disk is usually unable to respond to two contiguous requests on the same rotation, so sequentially allocated blocks incur the cost of an entire rotation. For both EFS and BSD-LFS, *bmap* was extended to return, not only the physical disk address, but the number of contiguous blocks that follow the requested block. Then, rather than reading one block at a time and requesting the next block asynchronously, the file system reads many contiguous blocks in a single request, providing extent-based reading. Because BSD-LFS potentially allocates many blocks contiguously, it may miss rotations between reading collections of blocks. Since EFS uses the FFS allocator, it leaves a rotational delay between clusters of blocks and does not pay this penalty.

5.2. The Evaluation Platform

Our benchmarking configuration consisted of a Hewlett-Packard series 9000/380 computer with a 25 Mhz MC68040 processor. It had 16 megabytes of main memory, and an HP 97560 SCSI disk. The hardware configuration is summarized in Table 5. The system was running the 4.4BSD-Alpha operating system and all measurements were taken with the system running single-user, unattached to any network. Each of the file systems used a 4-kilobyte block size with FFS and EFS using 1-kilobyte fragments.

The three file systems being evaluated run in the same operating system kernel and share most of their source code. There are approximately 6000 lines of shared C code, 4000 lines of LFS-specific code, and 3500 lines of FFS-specific code. EFS uses the same source code as FFS plus an additional 500 lines of clustering code, of which 300 are also used by BSD-LFS (for reads).

Each of the next sections describes a benchmark and presents performance analysis for each file system. The first benchmark analyzes raw file system performance. The next two benchmarks emulate specific workloads. A time-sharing environment is represented by a software development benchmark, and a database environment is represented by a modified version of the industry-standard TPC-B benchmark [TPCB90]. For a more thorough analysis of LFS and a wider range of benchmarks, see [SELT93a].

5.2.1. Raw File System Performance

The goal of this test is to measure the maximum throughput that can be expected from the given disk and system configuration for each of the file systems. For this test, the three file systems are compared

Disk (SCSI HP 97560)	
Average seek	13.0 ms
Single rotation	15.0 ms
Track size	36 KB
Track buffer	128 KB
Disk bandwidth	2.2 MB/sec
Bus bandwidth	1.6 MB/sec
Controller overhead	1.0 ms
Track skew	8 sectors
Cylinder skew	10 sectors
Cylinder size	19 tracks
Disk size	1962 cylinders
CPU (Motorola 68040)	
Memory Bandwidth	12.0 MB/sec
CPU	25 Mhz
MIPS	10-12

Table 5: Hardware Specifications. Although the disk can transfer at 2.2 megabytes per second, the bus bandwidth is limited to 1.6 megabytes per second. SCSI supports two transfer modes, synchronous and asynchronous [ADAP85]. Synchronous mode is optional under SCSI-I and is not supported by the disk driver. Therefore, all transfers are performed using asynchronous mode and are limited to 1.6 MB/sec.

against the maximum speed at which the operating system can write directly to the disk. The benchmark consists of creating a file of size *S* and then either reading or writing the entire file 50 times. The measurements recorded are averages across the 50 runs. For the read tests, the cache is flushed before each test by unmounting and remounting the file system.

Raw Write Performance

The graph in Figure 9 shows the bandwidth attained for writing, as a function of *S*, the size of the I/O. Given the sequential layout of both LFS and EFS, the expectation is that both should perform comparably to the speed of the raw disk and that FFS, with its rotational positioning, should achieve approximately 50% of the disk bandwidth. However, there are several anomalies.

First, as the I/O size increases, EFS actually provides more bandwidth than the raw disk partition. The explanation for this can be found by looking at the number of synchronous I/O's and the begin time for each operation. When accessing the raw partition, all I/Os are synchronous. Therefore, there is no overlap between the time required to copy the data from user-space into the kernel and the time required to perform the I/Os. As a result, there is a gap of approximately five milliseconds between the completion of each I/O and the initiation of the next I/O. In contrast, EFS has an aggressive buffering policy, allowing it to perform asynchronous writes in units of 64 kilobytes. Therefore, the I/Os are queued, and successive I/Os are begun almost immediately.

The next anomaly lies in the fact that LFS performs noticeably worse than either EFS or the RAW partition. This is an artifact of this benchmark as opposed to a fundamental difference in the attainable write bandwidth of the two file systems. The problem is that the benchmark performs the write and then calls *fsync* to ensure that the blocks have been written to disk.

LFS achieves its write performance by buffering a large number of dirty buffers before initiating I/O. As a result, LFS does not begin writing any data to disk until requested to do so by the application *fsync* or until the *start write threshold* has been reached. On this system, the *start write threshold* results in approximately 800 kilobytes of data being buffered before a write is initiated. As a result, for all the tests where the transfer size was smaller than one megabyte, the benchmark had two phases, the first in which data was written into the cache, and the second during which time the data was being written to disk.

To verify this, timings were taken after all the writes had been issued, but before the call to *fsync* and then again after the call to *fsync*. In the tests where the total transfer size was less than 800K, LFS' elapsed time for the *fsync* was nearly identical to the time required for EFS to write all its buffers. Figure 10 depicts this behavior. In the tests where the transfer size was greater than 800K, LFS' elapsed time for the *fsync* was the time reported for the synchronous LFS write that flushed the data remaining in the cache at the time of the *fsync*.

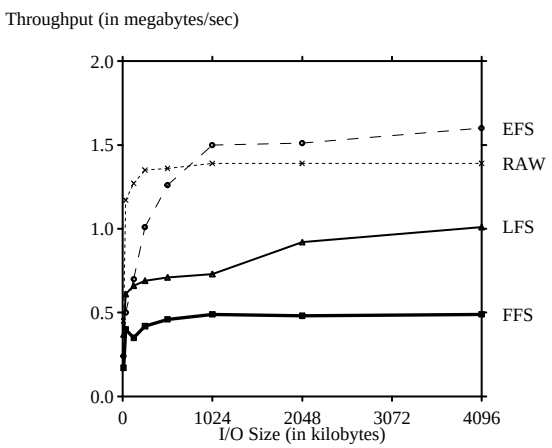

Figure 9: Maximum File System Write Bandwidth. This graph shows the write bandwidth of each file system as a function of the transfer size. EFS attains the best performance, as it performs nearly all its writes asynchronously in maximal-sized buffers. Writes to the RAW partition also occur in maximal-sized units, but are performed synchronously. In LFS, since a large amount of data is gathered in the cache before being written to the disk, there is less overlap between CPU processing and disk activity, leading to the gap shown above. The rotational delay of FFS prohibits it from achieving more than 25% of the available disk bandwidth.

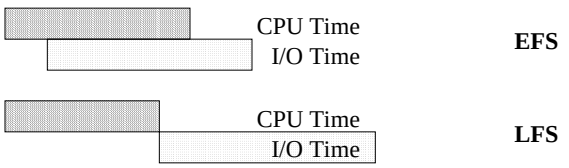

Figure 10: Effects of LFS Write Accumulation. The bars represent elapsed time for each phase of the benchmark on a one-half megabyte write. EFS effectively overlaps I/O and CPU processing while LFS waits until all the data is accumulated before initiating the write. As a result, the bandwidth measured by this test appears much lower for LFS.

These write tests were repeated for LFS with the cleaner running, but the results were indistinguishable from the results without the cleaner. Since the same data is overwritten for each iteration of the test, there are always empty segments available for reclamation by the cleaner. As a result, the cleaner reported that it always cleaned empty segments, and the overhead was unmeasurable.

The last anomaly is that FFS did not achieve the 50% bandwidth expected, but achieved closer to 31% of the transfer bandwidth (0.5 megabytes per second of the possible 1.6 megabytes per second). The explanation of this is in the FFS *rot_delay* parameter. This parameter is used by FFS to express the length of time, from the disk's perspective, that it takes the CPU to acknowledge the completion of an I/O and to issue another one.[3]

For the system under test, the *rot_delay* that provided the best performance was experimentally determined to be 4 milliseconds. This value was determined by building file systems with successively larger *rot_delays* and selecting the value that led to the best performance. However, with a rotational latency of 15 milliseconds, 36 kilobyte tracks, 4-kilobyte blocks, and a 4 millisecond *rot_delay*, only one in four blocks is allocated to the same file, as shown in Figure 11. The maximum transfer bandwidth of the disk is is 2.2 megabytes per second and one-quarter of this is 0.55 megabytes per second, which is close to the observed performance of FFS.

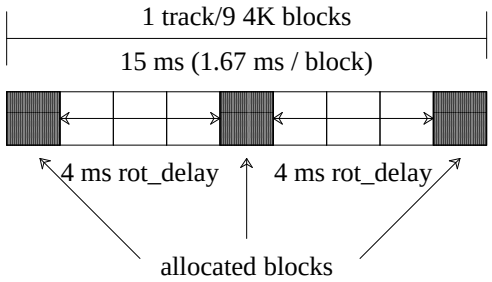

Figure 11: Impact of Rotational Delay on FFS Performance. Since *rot_delay* for this disk is 4 milliseconds, FFS will allocate only one in every four blocks. Therefore, at most 3 blocks (2.25 on average) can be accessed on each disk rotation. Therefore, FFS will attain at most one-quarter of the maximum bandwidth of the disk.

[3] This is based on the assumption that queueing is performed by the host and not the disk.

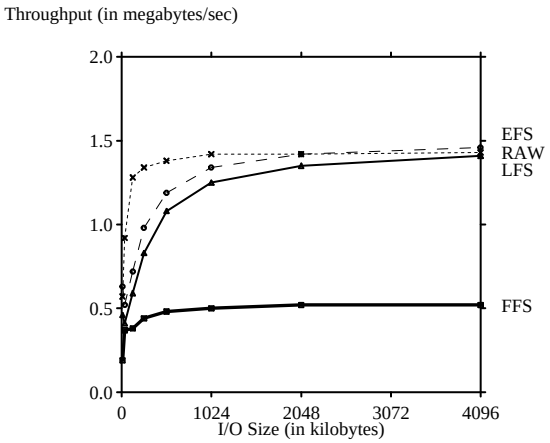

Figure 12: Maximum File System Read Bandwidth. The graph shows the maximum read throughput attained by each file system as a function of the transfer size. As EFS and LFS allocate blocks contiguously and use the exact same read-ahead algorithm, the expectation is that both will perform comparably to the raw partition. Once again, FFS is limited to approximately 25% of the total disk bandwidth due to rotational delays between allocated blocks.

Raw Read Performance

The results of the raw read tests, shown in Figure 12, are much closer to what is expected. FFS demonstrates read performance nearly identical to its write performance since it is limited by the number of blocks transferred during a single rotation. Both LFS and EFS perform comparably to the raw disk with very small (3%) differences in performance.

This benchmark demonstrates that both EFS and LFS can utilize close to 100% of the available I/O bandwidth on large I/Os. When individual write response time is an issue, LFS incurs a performance penalty due to its delayed write policy.

The remaining tests are all designed to stress the file systems. For BSD-LFS, that means the cleaner is running and the disk partitions are 80% utilized, so that the cleaner is forced to reclaim space. For EFS and FFS, as the disk partition fills, it becomes more difficult for them to allocate blocks optimally.

5.2.2. Software Development Workload

The next tests evaluate BSD-LFS in a typical software development environment. The Andrew benchmark [HOWA88] is often used for this type of measurement. It contains five phases:

1. Create a directory hierarchy.

2. Make a copy of the data.
3. Recursively examine the status of every file.
4. Examine every byte of every file.
5. Compile several of the files.

Unfortunately, the test set for the Andrew benchmark is small, and main-memory file caching can make the results uninteresting. In order to exercise the file systems, this benchmark is run both single-user and multi-user (where several invocations of the benchmark are run concurrently).

Single-User Andrew Performance

Table 6 shows the performance of the original Andrew benchmark. The entire five-phase test was run ten times for each of FFS, EFS, and LFS, with the directory hierarchy deleted after each pass. For the LFS test with the cleaner running (LFSC), the test was repeated 100 times to ensure that the file system was completely overwritten at least twice. In order to understand the differences in performance, kernel counters that record disk and LFS statistics were initialized before, and sampled after, each phase.

Overall, LFS demonstrates a 9% improvement over EFS and FFS, which perform comparably. The difference is isolated to phases one, two, and five. It is not surprising that LFS would outperform the other systems in phase one, the create phase, as LFS performs all its directory creations asynchronously, performing no writes, while EFS and FFS issue 100 synchronous writes each. As phase two is the write-intensive phase, it is also expected that LFS will perform better, and it does so, demonstrating 37% better performance than the other two systems. Again, EFS and FFS are performing a great deal of I/O (263 requests for about 750 kilobytes), over half of which are synchronous (as a result of closing files). LFS performs no writes during this phase as all the data is written to the cache.

Phase 5, which is moderately CPU-intensive (59% CPU utilization for LFS and 49% for EFS), surprisingly demonstrates a small (3-5%) advantage for LFS. Once again, kernel disk counters reveals that EFS and FFS are synchronously writing the output object files to disk (45 of 48 writes) while LFS is buffering the data and performing nearly one-third the number of writes.

The more striking difference is in the number of reads issued by the two file systems in phase five. LFS issues only a single read while FFS issues 46 of them. The explanation for this also lies in file allocation. When FFS creates a file, it allocates an inode from the appropriate cylinder group and then reads the contents of the inode from disk. (This is an artifact of the file system architecture and could be avoided by modifying the interface to the *vfs* routine *vfs_vget*.) In LFS, new inodes are created in memory, not read from the disk.

These single-user results differ slightly from those presented in [ROSE92]. First, the compilation phase in [ROSE92] is much longer than in this test because different compilers were used. Secondly, the results in [ROSE92] show LFS providing a 40% performance improvement on phase 3 (the phase that examines every inode) and a 29% performance improvement on phase 4 (the phase that examines every byte), while the results here show virtually no difference. Phases 3 and 4 perform no I/O on any of the file systems, so performance is limited strictly by the file system code that reads data from the cache, traverses directories, and reads inodes from the in-memory inode cache. Since the three file systems share the same code for performing these functions, the expectation is that the systems should behave identically. Since the system measured in [ROSE92] is unavailable for instrumentation, it is unclear why results on phases 3 and 4 differ.

Multi-User Andrew Performance

The multi-user version of Andrew shows the file system performance as a function of the degree of

	Phase 1 Create Directories	Phase 2 Copy Files	Phase 3 Stat Touch Inodes	Phase 4 Grep Touch Bytes	Phase 5 Compile	Total
FFS	2.10 (0.30)	7.90 (0.30)	6.30 (0.46)	9.00 (0.00)	44.80 (0.40)	70.1 (0.70)
EFS	2.10 (0.30)	7.90 (0.30)	6.70 (1.19)	9.10 (0.30)	44.40 (0.49)	70.2 (1.60)
LFS	0.33 (0.47)	5.00 (0.00)	6.50 (0.81)	9.07 (0.25)	42.90 (1.40)	63.8 (2.34)
LFSC	0.43 (0.49)	5.09 (0.28)	6.37 (0.48)	9.07 (0.26)	42.61 (0.49)	63.6 (0.62)

Table 6: Single-User Andrew Benchmark Results. This table shows the elapsed time for each phase of the benchmark on each file system. Reported times are the average across ten iterations with the standard deviation in parentheses. LFSC indicates that the benchmark was run on the log-structured file system with the cleaner running, but the similarity in results for most phases indicates that the cleaner had virtually no impact on performance. Overall, LFS demonstrates approximately a 9% difference in performance which can be attributed to asynchronous file creation and write-clustering.

multiprogramming. The test is performed by running N concurrent invocations of the benchmark, with each invocation creating, traversing, and removing its own directory hierarchy. The reported results are the averages of the results of each of ten runs for each invocation. The resulting averages are divided by the multiprogramming level to produce the metric ‘‘elapsed seconds per invocation.’’

The goal of the multi-user test is to examine two aspects of the file systems’ behavior under the software development workload. First, as the multiprogramming level increases, the entire data set no longer fits in the cache, so more I/O is performed. Secondly, with separate directory hierarchies, the different forms of locality used by LFS (temporal locality -- files created at about the same time reside close together) and FFS (logical locality -- files in the same directory are placed close together) can be compared.

The multi-user performance is the result of two competing factors. As concurrent invocations of the benchmark compete for resources, the utilization of both the CPU and the disk increases, as does performance. However, after the multiprogramming level exceeds two, the total working set becomes too large to fit in the cache and the total I/O time increases. Towards the left-hand side of the graph, the predominant factor is the overlap between the CPU and the disk. As LFS is already performing most of its I/O asynchronously, it has less room for improvement than EFS and FFS. So, the CPU utilization for LFS increases from 60% to 80% while the CPU utilization for EFS goes from 50% to 89%, explaining the steeper decline in elapsed time for EFS than for LFS.

As the multiprogramming level exceeds four, the data sets no longer fit in the cache and the read performance becomes the dominant factor for all the file systems. Kernel I/O statistics reveal that, on average, LFS is performing more seeks than EFS, explaining the small difference in performance observed as the multiprogramming level increases.

This benchmark indicates that LFS and EFS perform comparably on this particular software development workload. To generalize, LFS demonstrates superior file creation performance, but logical locality appears better than temporal locality when the working set is too large to fit in the cache. The next benchmark demonstrates this even more dramatically.

5.2.3. Transaction Processing Performance

A modified version of the industry-standard TPC-B is used as the database-oriented benchmark.

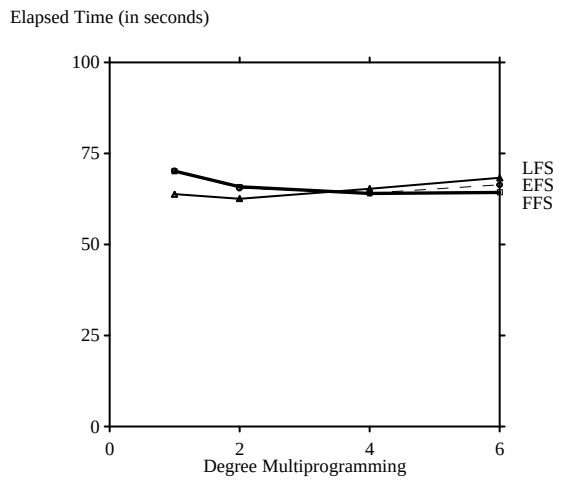

Figure 13: Multi-User Andrew Performance. This graph shows the elapsed time for all five phases of the Andrew benchmark under increasing multiprogramming. Overall, the impact of multiprogramming is less significant than might have been expected, yielding at most a 9% performance improvement.

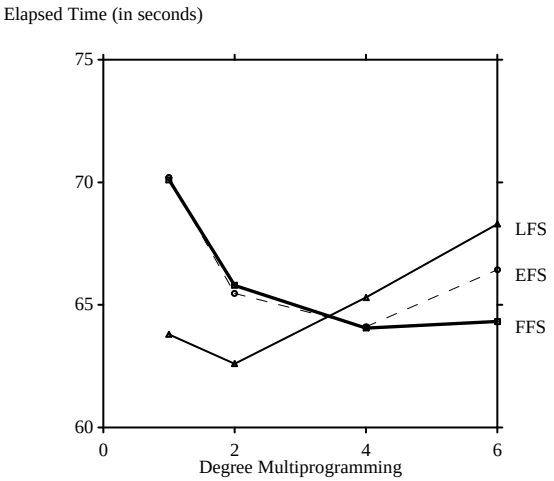

Figure 14: Multi-User Andrew Performance (Blow-Up). This graph emphasizes the small performance differences in the multi-user Andrew benchmark. For EFS and FFS which perform many synchronous operations, multi-programming allows the overlapping of CPU and disk and reduces per-invocation time. LFS also benefits from this overlap, but not as significantly as the other systems. The second effect is that the total data set size begins to exceed the cache capacity and read performance becomes the dominant factor.

The system is configured for a ten transaction-per-second system, but runs single-user without a redundant log and does not model think time. Each measurement in Table 7 represents ten runs of 1000 transactions. The counting of transactions is not begun until the buffer pool has filled, so the measurements do not exhibit any artifacts of an empty cache. Transaction run lengths of greater than 1000 were measured, but there was no noticeable change in performance after the first 1000 transactions.

When the cleaner is not running, LFS provides a 15% performance improvement over EFS. However, the impact of the cleaner is far worse than was anticipated. The benchmark randomly updates blocks in the 237 megabyte account relation, leaving most segments fairly full. During the course of the benchmark, the cleaner cleaned approximately one segment for every 50 transactions executed. On average, the cleaned segments were 71% utilized and cleaner writes accounted for between 60% and 80% of the total blocks written and 31% of all blocks transferred.

In an attempt to reduce cleaner overhead, a second set of tests were run with a smaller segment size (256 kilobytes). The performance before cleaning is the same as for the one megabyte case, but the after-cleaning performance is only slightly better (about 6%). As in the one megabyte case, the majority of the writes performed are on behalf of the cleaner (60-70%). While the smaller segment size reduces the variation in response time as evidenced through the smaller standard deviation, it does not significantly improve performance as most of the write activity is due to the cleaner. Although a user-level cleaner avoids synchronization costs between user processes and the kernel, it cannot avoid contention on the disk arm.

	Transactions per second	Elapsed Time 1000 transactions
FFS	14.2	70.23 (1.0%)
EFS	16.8	59.51 (2.1%)
LFS (no cleaner)	19.3	51.75 (0.6%)
LFS (cleaner, 1M)	11.6	85.86 (5.3%)
LFS (cleaner, 256 K)	12.4	80.72 (1.8%)

Table 7: Modified TPC-B Performance Results. The test database was scaled for a 10 transaction-per-second system (1,000,000 accounts, 100 tellers, and 10 branches). The elapsed time and standard deviation, as a percent of the elapsed time, is reported for runs of 1000 transactions. The LFS results show performance before the cleaner begins to run and after the cleaner begins to run. Since the cleaner decreased performance by 40%, a second test was run with 256 kilobyte segments. Even with the smaller segment size, the cleaner decreased performance by 35%.

6. Conclusions

The implementation of BSD-LFS highlighted some subtleties in the overall LFS strategy as well as some performance deficiencies. While LFS can utilize a large fraction of the disk bandwidth for writing, the cleaner has a severe impact in certain workloads, particularly transaction processing.

While allocation in BSD-LFS is simpler than in extent-based file systems or file systems like FFS, the management of memory is much more complicated. The Sprite-LFS implementation addressed this problem by reserving large amounts of memory. Since this is not feasible in most environments, a more complex mechanism to manage buffer and memory requirements is necessary. LFS operates best when it can write out many dirty buffers at once. However, holding dirty data in memory until much data has accumulated requires consuming more memory than might be desirable and may not be allowed (e.g. NFS semantics require synchronous writes). In addition, the act of writing a segment requires allocation of additional memory (for segment summaries and on-disk inodes), so segment writing needs to be initiated before memory becomes a critical resource to avoid memory thrashing or deadlock.

The delayed allocation of BSD-LFS makes accounting of available free space more complex than that in a pre-allocated system like FFS. In Sprite-LFS, the space available to a file system is the sum of the disk space and the buffer pool. As a result, data is written to the buffer pool for which there might not be free space available on disk. Since the applications that wrote the data may have exited before the data is written to disk, there is no way to report the ‘‘out of disk space’’ condition. This failure to report errors is unacceptable in a production environment. To avoid this phenomena, available space accounting must be done as dirty blocks enter the cache instead of when they are written from cache to disk.

7. Future Directions

The novel structures of BSD-LFS makes it an exciting vehicle for adding functionality to the file system. For example, there are two characteristics of BSD-LFS that make it desirable for transaction processing. First, the multiple, random writes of a single transaction get bundled and written at sequential speeds, so we expect to see a dramatic performance improvement in multi-user transaction applications, if sufficient disk space is available. Second, since data is never overwritten, before-images of updated pages

exist in the file system until they are reclaimed by the cleaner. An implementation that exploits these two characteristics is described and analyzed in [SELT93b] on Sprite-LFS, and we plan on doing a prototype implementation of transactions in BSD-LFS.

The ''no-overwrite'' characteristic of BSD-LFS makes it ideal for supporting *unrm* which would undo a file deletion. Saving a single copy of a file is no more difficult than changing the cleaner policy to not reclaim space from the last version of a file, and the only challenge is finding the old inode. More sophisticated versioning should be only marginally more complicated.

Also, the sequential nature of BSD-LFS write patterns makes it nearly ideal for tertiary storage devices [KOHL93]. LFS may be extended to include multiple devices in a single file system. If one or more of these devices is a robotic storage device, such as a tape stacker, then the file system may have tremendous storage capacity. Such a file system would be particularly suitable for on-line archival or backup storage.

An early version of the BSD-LFS implementation was shipped as part of the 4.4BSD-Alpha release. The current version described in this paper will be available as part of 4.4BSD. Additionally, the FFS shipped with 4.4BSD will contain the enhancements to provide clustered reading and writing.

8. Acknowledgements

Mendel Rosenblum and John Ousterhout are responsible for the original design and implementation of Sprite-LFS. Without their work, we never would have tried any of this. We also wish to express our gratitude to John Wilkes and Hewlett-Packard for their support.

9. References

[ADAP85] *SCSI User Guide*, Adaptive Data Systems Inc., Pomona, CA, 1985.

[BAKE91] Baker, M., Hartman., J., Kupfer., M., Shirriff, L., Ousterhout, J., ''Measurements of a Distributed File System,'' *Proceedings of the 13th Symposium on Operating System Principles*, Monterey, CA, October 1991, 198-212. Published as *Operating Systems Review 25*, 5 (October 1991).

[BAKE92] Baker, M., Asami, S., Deprit, E., Ousterhout, S., Seltzer, M., ''Non-Volatile Memory for Fast, Reliable File Systems,'' *Proceedings of the Fifth Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, October 1992.

[CAR92] Carson, S., Setia, S., ''Optimal Write Batch Size in Log-structured File Systems,'' *Proceedings of 1992 Usenix Workshop on File Systems*, Ann Arbor, MI, May 21-22 1992, 79-91.

[KAZA90] Kazar, M., Leverett, B., Anderson, O., Vasilis, A., Bottos, B., Chutani, S., Everhart, C., Mason, A., Tu, S., Zayas, E., ''DECorum File System Architectural Overview,'' *Proceedings of the 1990 Summer Usenix* Anaheim, CA, June 1990, 151-164.

[HAER83] Haerder, T. Reuter, A. ''Principles of Transaction-Oriented Database Recovery,'' *Computing Surveys*, 15(4); 1983, 237-318.

[HOWA88] 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.

[KLEI86] S. R. Kleiman, ''Vnodes: An Architecture for Multiple File System Types in Sun UNIX,'' *Usenix Conference Proceedings*, June 1986, 238-247.

[KOHL93] Kohl, J., Staelin, C., Stonebraker, M., ''Highlight: Using a Log-structured File System for Tertiary Storage Management,'' *Proceedings 1993 Winter Usenix*, San Diego, CA, January 1993.

[MCKU84] Marshall Kirk McKusick, William Joy, Sam Leffler, and R. S. Fabry, ''A Fast File System for UNIX'', *ACM Transactions on Computer Systems*, 2(3), August 1984, 181-197.

[MCKU90] Marshall Kirk McKusick, Michael J. Karels, Keith Bostic, ''A Pageable Memory Based Filesystem,'' *Proceedings of the 1990 Summer Usenix Technical Conference*, Anaheim, CA, June 1990, 137-144.

[MORA90] Moran, J., Sandberg, R., Coleman, D., Kepecs, J., Lyon, B., Breaking Through the NFS Performance Barrier,'' *Proceedings of the 1990 Spring European Unix Users Group*, Munich, Germany, 199-206, April 1990.

[MCVO91] McVoy, L., Kleiman, S., ''Extent-like Performance from a Unix File System,'' *Proceedings Winter Usenix 1991*, Dallas, TX, January 1991, 33-44.

[OUST85] Ousterhout, J., Costa, H., Harrison, D., Kunze, J., Kupfer, M., Thompson, J., ''A Trace-Driven Analysis of the UNIX 4.2BSD File System,'' *Proceedings of the Tenth Symposium on Operating*

System Principles, December 1985, 15-24. Published as *Operating Systems Review 19*, 5 (December 1985).

[OUST89] Ousterhout, J., Douglis, F., ''Beating the I/O Bottleneck: A Case for Log-structured File Systems,'' *Operating Systems Review 23*, 1, January 1989, 11-27. Also available as Technical Report UCB/CSD 88/467.

[ROSE90] Rosenblum, M., Ousterhout, J. K., ''The LFS Storage Manager,'' *Proceedings of the 1990 Summer Usenix*, Anaheim, CA, June 1990, 315-324.

[ROSE91] Rosenblum, M., Ousterhout, J. K., ''The Design and Implementation of a Log-Structured File System,'' *Proceedings of the Symposium on Operating System Principles*, Monterey, CA, October 1991, 1-15. Published as *Operating Systems Review 25*, 5 (October 1991). Also available as *Transactions on Computer Systems 10*, 1 (February 1992), 26-52.

[ROSE92] Rosenblum, M., ''The Design and Implementation of a Log-structured File System,'' PhD Thesis, University of California, Berkeley, June 1992. Also available as Technical Report UCB/CSD 92/696.

[SELT90] Seltzer, M., Chen, P., Ousterhout, J., ''Disk Scheduling Revisited,'' *Proceedings of the 1990 Winter Usenix*, Washington, D.C., January 1990, 313-324.

[SELT91] Seltzer, M., Stonebraker, M., ''Read Optimized File Systems: A Performance Evaluation,'' *Proceedings 7th Annual International Conference on Data Engineering*, Kobe, Japan, April 1991, 602-611.

[SELT93a] Seltzer, M., ''File System Performance and Transaction Support,'' PhD Thesis, University of California, Berkeley, May 1993.

[SELT93b] Seltzer, M., ''Transaction Support in a Log-Structured File System,'' To appear in the *Proceedings of the 1993 International Conference on Data Engineering*, Vienna, Austria, April 1993.

[THOM78] Thompson, K., ''Unix Implementation,'' *Bell Systems Technical Journal*, 57(6), part 2, July-August 1978, 1931-1946.

[TPCB90] Transaction Processing Performance Council, ''TPC Benchmark B,'' Standard Specification, Waterside Associates, Fremont, CA., 1990.

Margo I. Seltzer is an Assistant Professor at Harvard University. Her research interests include file systems, databases, and transaction processing systems. She spent several years working at startup companies designing and implementing file systems and transaction processing software and designing microprocessors. Ms. Seltzer completed her Ph.D. in Computer Science at the University of California, Berkeley in December of 1992 and received her AB in Applied Mathematics from Harvard/Radcliffe College in 1983.

Keith Bostic has been a member of the Berkeley Computer Systems Research Group (CSRG) since 1986. In this capacity, he was the principle architect of the 2.10BSD release of the Berkeley Software Distribution for PDP-11's. He is currently one of the two principal developers in the CSRG, continuing the development of future versions of Berkeley UNIX. He received his undergraduate degree in Statistics and his Masters degree in Electrical Engineering from George Washington University. He is a member of the ACM, the IEEE and several POSIX working groups.

Dr. Marshall Kirk McKusick got his undergraduate degree in Electrical Engineering from Cornell University. His graduate work was done at the University of California, where he received Masters degrees in Computer Science and Business Administration, and a Ph.D. in the area of programming languages. While at Berkeley he implemented the 4.2BSD fast file system and was involved in implementing the Berkeley Pascal system. He currently is the Research Computer Scientist at the Berkeley Computer Systems Research Group, continuing the development of future versions of Berkeley UNIX. He is past-president of the Usenix Association, and a member of ACM and IEEE.

Carl Staelin works for Hewlett-Packard Laboratories in the Berkeley Science Center and the Concurrent Systems Project. His research interests include high performance file system design, and tertiary storage file systems. As part of the Science Center he is currently working with Project Sequoia at the University of California at Berkeley. He received his PhD in Computer Science from Princeton University in 1992 in high performance file system design.