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] demonstrate 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 [OUST88], 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 [OUST88]. 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.

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 ''metadata'' 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 bitmaps 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.

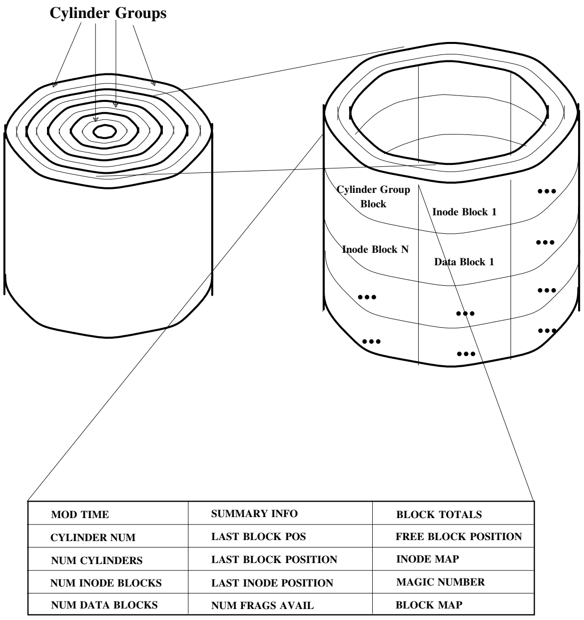

Figure 1: Physical Disk Layout of the Fast File System. The disk is statically partitioned into cylinder groups, each of which is described by a cylinder group block, analogous to a file system superblock. Each cylinder group contains a copy of the superblock and allocation information for the inodes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

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.

LFS is a hybrid between a sequential database log and FFS. It does 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.

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	bitmaps	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

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.

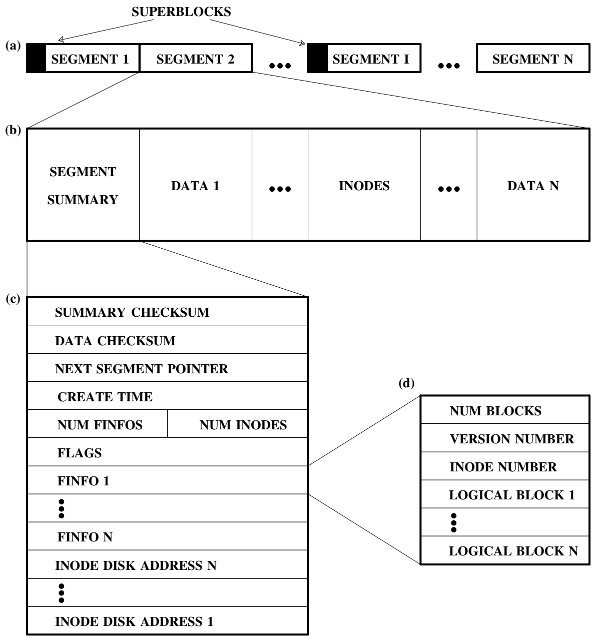

Figure 2: A Log-Structured File System. A file system is composed of **segments** as shown in Figure (a). Each segment consists of a summary block followed by data blocks and inode blocks (b). The **segment summary** contains checksums to validate both the segment summary and the data blocks, a timestamp, 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).

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.

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 bitmaps and meta-data. At the same 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 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 bitmaps 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.

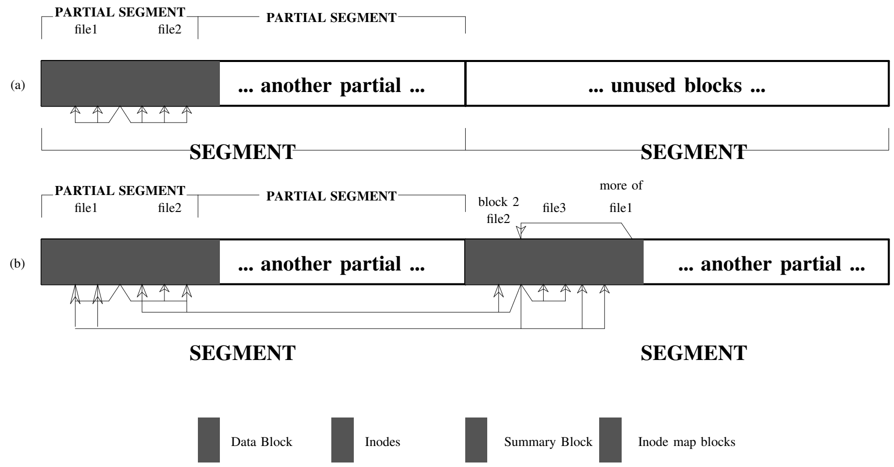

Figure 3: A Log-Structured File System. In figure (a), two files have been written, file1 and file2. Each has an index structure in the meta-data block that is allocated after it on disk. In figure (b), the middle block of file2 has been modified. A new version of it is added to the log, as well as a 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.

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 bitmaps in cylinder groups.

Table 2: Five Phases of *fsck*

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.

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.

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.

Segment Structure and Validation

Sprite-LFS places segment summary blocks at the end of the segment trusting that if the write containing the segment summary is issued after all other writes in that partial segment, the presence of the segment summary validates the partial segment.

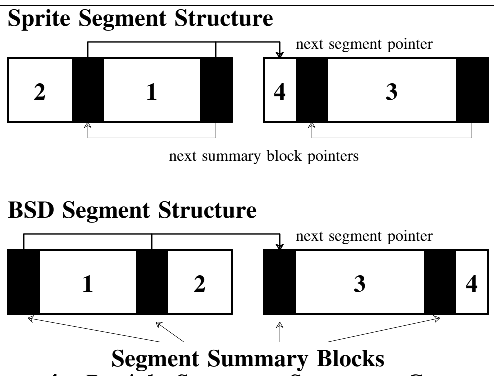

Figure 4: Partial Segment Structure Comparison Between Sprite-LFS and BSD-LFS. The numbers in each partial show the order in which the partial segments are created. Sprite-LFS builds segments back to front, chaining segment summaries. BSD-LFS builds segments front to back. After reading a segment summary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

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. Therefore, 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, methods exist for guaranteeing that any missing 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.[1]

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.

[1]The database community uses a technique called patch tables to verify multi-sector block writes. Although many people have heard of this technique, none knew of any published reference for it.

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 [SELT93] 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 theses 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. Choose one or more target segments and read them.
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.

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

BLOCK_INFO STRUCTURE

INODE NUMBER
LOGICAL BLOCK NUMBER
CURRENT DISK ADDRESS
SEGMENT CREATION TIME
BUFFER POINTER

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.

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 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. To ensure that the cleaner can always run, 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 the 1 + *utilization* (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 of the most recent modification to a 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.

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 reasonable 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 (i.e., 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.).

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 bitmaps 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 bitmaps while BSD-LFS inserts the inode onto a free list.
truncate	Truncate a file from the given offset. FFS marks bitmaps 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.

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, layout, 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.

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 multiblock policy simulated in [SELT91].

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 never attempts to 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.

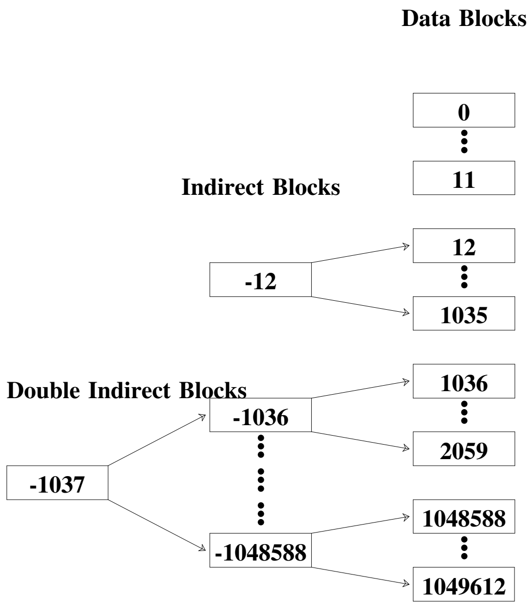

Figure 6: Block-numbering in BSD-LFS. In BSD-LFS, data blocks are assigned positive block numbers beginning with 0. Indirect blocks are numbered with the negative of the first data block to which they point. Double and triple indirect blocks are numbered with one less than the first indirect or double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

Second, the buffer cache assumes that meta-data (both indirect blocks and inodes) are assigned disk addresses when they are created and can be assigned block numbers corresponding to those disk addresses. In BSD-LFS, the disk address is assigned when blocks are written to disk instead of when they are written in to the cache. This lazy assignment of disk addresses violates the assumption that all blocks have disk addresses. FFS accesses indirect blocks by hashing on the raw device vnode and the disk block number. Since BSD-LFS has no disk address for these indirect blocks, 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.

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 3.6, 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.

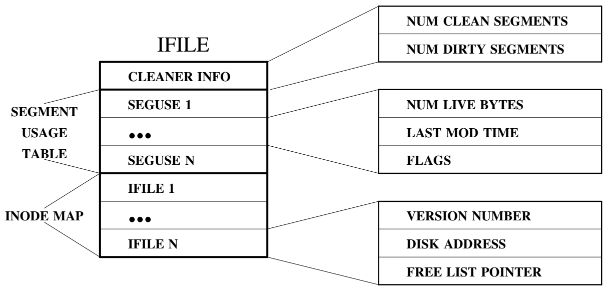

Figure 7: Detail Description of the IFILE. The *ifile* is maintained as a regular file with read-only permission. It facilitates communication between the file system and the cleaner.

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 significantly larger, BSD-LFS keeps the access time in the inode. Sprite-LFS allocates inodes by scanning the inode map sequentially until it finds a free inode. This scan is costly if the file system has many inodes. 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.

Directory Operations

Directory operations[2] 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[3].

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,

[2]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.

[3]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!

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.

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.

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.

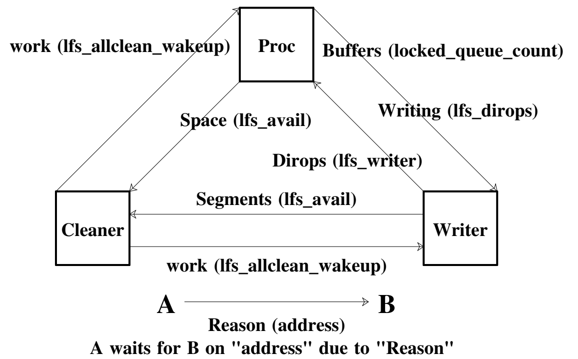

Figure 8: Synchronization Relationships in BSD-LFS. The cleaner has precedence over all components in the system. It waits on the *lfs_allclean_wakeup* condition and wakes the segment writer or user processes using the *lfs_avail* condition. The segment writer and user processes maintain directory operation 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.

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.

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.

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, read-optimized file systems on a variety of benchmarks based on real workloads. Read-optimized policies that favor large blocks or contiguous layout perform similarly [SELT91], so we analyze FFS and a variant of FFS that does extent-like allocation. The new log-structured file system was written in November of 1991 and was left largely untouched until late spring 1992, and is therefore 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.

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.

The Evaluation Platform

Our benchmarking configuration consists of a Hewlett-Packard series 9000/300 computer with a 25 Mhz MC68040 processor. It has 16 megabytes of main memory, and an HP RD335 with an HPIB[4] interface. The hardware configuration is summarized in Table 5. The system is 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 uses a 4K block size with FFS and EFS having 1K fragments.

Disk (HP RD335)	
Average seek	16.6 ms
Single rotation	15.0 ms
Transfer time	1 MB / sec
Track size	56.5 KB / track
CPU	25 Mhz
MIPS	10-12

Table 5: Hardware Specifications

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 read clustering.

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 attempt to model specific workloads. A time-sharing environment is modeled by a software development benchmark, and a database environment is modeled by the industry-standard TPCB benchmark [TPCB90].

[4] HPIB is an unbelievably slow interface as is shown in Table 6.

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 against the maximum speed at which the operating system can write directly to the disk. The benchmark consists of either reading or writing a large amount of data sequentially to a newly created file system or a raw partition. Two versions of this test are run. In the first, the data is appended, while in the second the same data is repeatedly overwritten. While FFS and EFS merely rewrite the existing data blocks, BSD-LFS is continually allocating new blocks and marking the old blocks as no longer in use, thus requiring more CPU processing than any of the other file systems. The results for both the APPEND and OVERWRITE tests are shown in Table 6.

Given the sequential layout of both BSD-LFS and EFS, the expectation is that both should perform comparably to the speed of the raw disk. For the write test, both are nearly identical and within 10% of the raw disk speed. This 10% difference is an approximation of the file system overhead. In BSD-LFS, the overhead comes from creating segments, while in EFS the overhead comes from doing block allocation. Since FFS does sequential allocation with a rotational delay between each block, the expectation is that it should exhibit performance approximately half that of EFS and BSD-LFS. The performance measurements support this hypothesis.

FFS demonstrates read performance approximately 10% better than write performance for the append tests. The 10% improvement is due partially to read-ahead and partially to avoiding the file system block allocation overhead required during writing. Since EFS and BSD-LFS do sequential allocation, the expectation is that the read performance

	APPENDS					
	Writes			**Reads**		
Transfer Unit	.5 M	1 M	2 M	.5M	1 M	2 M
RAW	0.31 (0.00)	0.31 (0.00)	0.31 (0.00)	0.45 (0.01)	0.45 (0.00)	0.45 (0.00)
FFS	0.11 (0.00)	0.11 (0.00)	0.12 (0.00)	0.14 (0.00)	0.14 (0.00)	0.14 (0.00)
EFS	0.26 (0.02)	0.28 (0.01)	0.28 (0.01)	0.38 (0.02)	0.38 (0.00)	0.36 (0.03)
LFS	0.27 (0.00)	0.28 (0.00)	0.29 (0.00)	0.33 (0.00)	0.36 (0.00)	0.37 (0.00)

	OVERWRITES					
	Writes			**Reads**		
Transfer Unit	.5 M	1 M	2 M	.5M	1 M	2 M
RAW	0.30 (0.00)	0.30 (0.00)	0.30 (0.00)	0.43 (0.00)	0.43 (0.00)	0.43 (0.00)
FFS	0.12 (0.00)	0.12 (0.00)	0.12 (0.00)	0.12 (0.00)	0.14 (0.00)	0.14 (0.00)
EFS	0.29 (0.01)	0.30 (0.00)	0.28 (0.00)	0.35 (0.00)	0.37 (0.00)	0.37 (0.00)
LFS	0.25 (0.00)	0.26 (0.00)	0.28 (0.00)	0.33 (0.00)	0.35 (0.00)	0.36 (0.00)

	BSD-LFS Overwrite Performance with Cleaner Running					
	Writes			**Reads**		
Transfer Unit	.5 M	1 M	2 M	.5M	1 M	2 M
0% Utilization	0.25 (0.00)	0.26 (0.00)	0.28 (0.00)	0.33 (0.03)	0.35 (0.02)	0.36 (0.01)
50% Utilization	0.24 (0.01)	0.26 (0.01)	0.27 (0.03)	0.33 (0.03)	0.35 (0.02)	0.36 (0.01)

Table 6: Raw File System Performance (MB/sec). The first two tables show the raw file system performance in megabytes/second. The numbers shown are averages of ten runs with the standard deviation in parentheses. In the first table, new data is appended to a file each time, while in the second, the data is overwritten. The benchmark issues the writes in .5 megabyte, 1 megabyte, or 2 megabyte requests as specified by the columns. The third table shows the overwrite test for BSD-LFS when the cleaner is running. In the first row, the same file is repeatedly overwritten, leaving very little work for the cleaner while in the second, alternating files are overwritten so that each segment cleaned is approximately half full.

should be comparable to that of the raw disk. However, Table 6 shows that BSD-LFS and EFS are achieving only 70% and 85% of the raw disk performance respectively. The explanation for BSD-LFS lies in the mapping between the file written and the on-disk segment layout. The file system is configured with one megabyte segments. A one megabyte segment does not hold one megabyte worth of data, due to the presence of meta-data (segment summary blocks, inode blocks, and indirect blocks) in the segment. As a result, the requests span segment boundaries, and the resulting seek often incurs the penalty of a full rotation. The extent-based system does better because it takes advantage of FFS' understanding of rotational delay. It pays only a rotational delay (on the order of 0.25 rotations) between clusters of blocks. All the systems suffer a small (approximately 5%) performance penalty for the overwrite test, since a disk seek is required before each read is issued.

The third table in Table 6 shows impact of the cleaner on BSD-LFS. In the first row (0% utilization), the same file is overwritten repeatedly. As a result, the only valid data is in the current segment, and the cleaner does very little work to reclaim space. In the second row, (50% utilization), every other file is overwritten, leaving each segment half full. In this case, the cleaner copies one-half segment, on average, to reclaim one segment of space.

As expected, the cleaner had virtually no impact during the read test (since there was no data being overwritten). For the 0% utilization test, there is also no impact on performance which is not surprising. Even when each segment is half utilized, the impact is under 10%, However, when the cleaner is running, performance is less predictable as is evidenced by the 10% standard deviations observed during those tests.

The remaining tests are all designed to stress the file systems. For BSD-LFS, that means the cleaner is running and the disk systems are fairly full (above 80%), so that the cleaner is forced to reclaim space. For EFS and FFS, it becomes more difficult for them to allocate blocks optimally when they run on fairly full file systems, so degradation is expected for those systems as well.

Software Development Workload

The next tests evaluate BSD-LFS in a typical software development environment. The Andrew benchmark [OUST90] is often used for this type of measurement. The Andrew benchmark was created by M. Satyanarayanan of the Information Technology Center at Carnegie-Mellon University. It contains five phases.

1. Create a directory hierarchy.
2. Make several copies 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. To exercise the file systems, this benchmark is run both single-user and multi-user, and the system's cache is kept small (1.6 megabytes). Table 7 shows the performance of the standard Andrew benchmark with multi-user results presented in Table 8.

As expected, BSD-LFS does quite well on the directory creation phase of the benchmark (Phase 1), as BSD-LFS avoids synchronous writes and pays no allocation overhead during directory creation. However, the remaining phases of the test exercise read performance, and EFS performs the best with BSD-LFS and FFS performing comparably. Although BSD-LFS can take advantage of contiguous layout, as does EFS, it exhibits poorer performance due to I/Os issued by the cleaner. In this test, the cleaner pays a substantial performance penalty during cleaning, because it reads entire segments before determining which blocks are ''live''. Since the files in this benchmark are created and deleted in large groups, most of the blocks read by the cleaner are discarded and most of the reads accomplished nothing.

These single-user results are significantly different from those of Sprite-LFS discussed in [ROSE92]. There are several reasons for this. As mentioned earlier, a goal of this test is to stress the file systems, so both the cache and the file system are small. The small cache (1.6 megabytes) ensures that both read and write performance to the disk can be measured. The small file system guarantees that

	Phase 1	Phase 2	Phase 3	Phase 4	Phase 5
FFS	3.30	13.20	7.00	16.10	48.80
EFS	3.30	11.80	6.90	13.20	46.00
LFS	1.30	13.70	8.00	16.60	46.20

Table 7: Single-User Andrew Benchmark Results. This table shows the average elapsed time, in seconds, for each phase of the Andrew benchmark. As expected, BSD-LFS does best on phase 1, the directory creation portion of the text. However, in the remaining tests, the extent-based system provides the best performance because it benefits from contiguous layout on both reading and writing, but does not vie for disk servicing with the cleaner.

the cleaner has work to do, so its impact on system performance can be analyzed.

For the multi-user benchmark, four separate trees are created and the benchmark runs concurrently in each tree. The reported results are averaged across ten runs on each of the four trees for a total of forty iterations. The multi-user Andrew results are shown in Table 8.
In a multi-user environment, the strengths and weaknesses of BSD-LFS become more apparent. Phase 1 performance is exceptionally good because BSD-LFS is not doing synchronous file and directory creation. Since phase 2 is write intensive, BSD-LFS performs approximately 35% better than its nearest competitor (EFS), because it pays very few seeks in writing. In phase 3, where every file's inode is examined, EFS demonstrate approximately 10% better performance then either LFS or FFS. In phase 4 every byte of every file is examined. Both LFS and EFS achieve approxmiately a 10% performance improvement over FFS due to the contiguous nature of their reads. LFS performs slightly better because the different trees were likely to be created at approximately the same time, and their data is likely to be physically close on the disk. Phase 5 highlights the weakness in LFS. This phase is characterized by interspersed reads and writes. Once again, the large reads and writes performed by the cleaner compete with synchronous read requests, making LFS the poorest performer by approximately 8%. When the cleaner is forced to run, synchronous I/O performance, typically read performance, suffers. The next benchmark demonstrates this even more dramatically.

Transaction Processing Performance

The industry-standard TPCB is used as a database-oriented test. Our benchmark is a modified version of the TPCB benchmark, configured for a 10 transaction per second system. While the benchmark is designed to be run multi-user, we show only single-user (worst case) results. Additionally, we do not keep redundant logs and we do not model ''think time'' between transactions. Each data point 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, BSD-LFS behaves as predicted in simulation [SELT90]. It shows approximately 20% improvement over the extent-based system. However, the impact of the cleaner is far worse than was expected. With one megabyte segments and the small random I/Os done by TPCB, most segments have only a few dead blocks available for reclamation. As a result, the

	Phase 1	Phase 2	Phase 3	Phase 4	Phase 5
FFS	10.68	36.17	19.18	36.27	167.40
EFS	9.12	34.25	16.90	32.55	168.75
LFS	1.02	22.30	19.23	31.10	183.20

Table 8: Multi-User Andrew Benchmark Results. This table shows the average elapsed time, in seconds, of each phase of the Andrew benchmark with four concurrent invocations. The times are less than four times slower than the single-user times due to the overlapped CPU and I/O time. The multi-user test emphasizes the strengths and weaknesses of BSD-LFS. It performs well in phases 1 and 2 which are predominantly writes and in phase 4 where temporal locality produces good read performance. However, performance suffers in Phase 5 where read and write traffic is interspersed and the impact of the cleaner is most severe.

	Transactions per second	Elapsed Time 1000 transactions
FFS	8.9	112.51 (1.2)
EFS	9.0	111.03 (1.3)
LFS (no cleaner)	10.8	92.76 (0.87)
LFS (cleaner, 1M)	4.3	232.57 (16.84)
LFS (cleaner, 128K)	5.0	200.57 (68.72)

Table 9: TPCB Performance Results for FFS, EFS, and BSD-LFS. The TPCB database was scaled for a 10 TPS system (1,000,000 accounts, 100 tellers, and 10 branches). The elapsed time is reported for runs of 1000 transactions. The BSD-LFS results show performance before the cleaner begins to run and after the cleaner begins to run. The after-cleaner performance is shown for file systems with segment sizes of one megabyte and 128K.

cleaner does large reads as well as large writes to reclaim space. Each of these large I/O operations busies the disk for long periods of time, making the TPCB reads wait.

In an attempt to reduce this wait time, a second set of tests were run with a smaller segment size (128 kilobytes). The performance before cleaning is the same as for the 1 megabyte case, but the after-cleaning performance is slightly better (about 13%). In either case, the impact of the cleaner is so severe that BSD-LFS cannot compete with either FFS or EFS. For the tests presented here, the disk was running at 85% utilization, and the cleaner was continually running. Note that FFS and EFS allocate up to 90% of the disk capacity without exhibiting any performance degradation [MCKU84]. This result shows that log-structured file systems are much more sensitive to the disk utilization. While the user-level cleaner avoids synchronization costs between user processes and the kernel, it cannot avoid the contention on the disk arm.

6. Conclusions

The implementation of BSD-LFS highlighted some subtleties in the overall LFS strategy. 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. Accounting for the actual disk space required is difficult because inodes are not written into dirty buffers and segment summaries are not created until the segment is 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

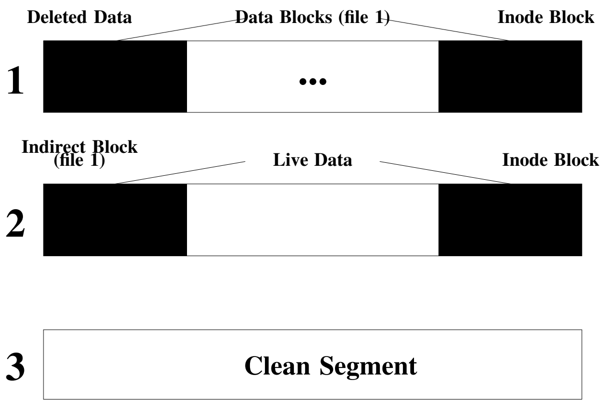

Figure 9: Segment Layout for Bad Cleaner Behavior. Segments 1 and 2 contain data. The cleaner will attempt to free up the one disk block of deleted data from segment 1. However, to rewrite the data in segment 1, it will dirty the meta-data block currently in segment 2. As a result, the cleaner will not generate any additional clean blocks.

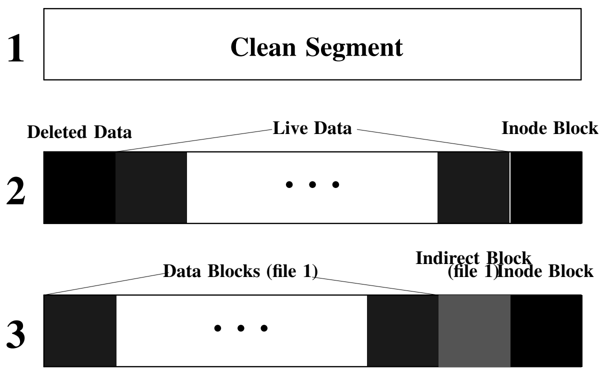

Figure 10: Segment Layout After Cleaning. The cleaner cleaned segment 1. In doing so, it rewrote the indirect block that previously resided in segment 2. Now that block has been deleted and the cleaner will be able to reclaim a disk block by cleaning segment 2.

available on the disk. To create more available disk space, either the cleaner must run or dirty blocks in the cache must be deleted.

A third and more critical situation arises when the cleaner consumes more disk space than it frees during cleaning. Although this cannot happen over the long term, during short term intervals it can. Consider the simple three segment file system shown below in Figure 9. Segment 1 contains one free block (the first block marked ''Deleted Data''). However, cleaning that segment requires rewriting the indirect block for file 1. Therefore, after segment 1 is cleaned, segment 3 will be full, segment 1 will be clean, and one block in segment 2 will be dead (Figure 10). While the total number of live blocks on the system has not increased, it has not decreased either, and the act of cleaning the segment has not created any additional space. It is possible to construct cases where cleaning a segment actually decreases the amount of available space (consider a segment that contains N blocks from N different files, each of which is accessed via an indirect block and the indirect block resides in a different segment). Therefore two segments are reserved for the cleaner. One guarantees that the cleaner can run at all, and the second ensures that small overflows can be accommodated until more space is reclaimed.

7. Future Directions

The novel structures of BSD-LFS makes it an exciting vehicle for adding functionality to the file system. For example, two characteristics of BSD-LFS 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 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 reclaimed by the cleaner. An implementation that exploits these two characteristics is described and analyzed in [SELT93] 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 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 for 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

[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,'' to appear in *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.

[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).

[OUST88] Ousterhout, J., Douglis, F., ''Beating the I/O Bottleneck: A Case for Log Structured File Systems'', Computer Science Division (EECS), University of California, Berkeley, UCB/CSD 88/467, October 1988.

[OUST90] Ousterhout, J. ''Why Aren't Operating Systems Getting Faster As Fast as Hardware?'' Proceedings of the 1990 Summer Usenix, Anaheim, CA, June 1990, 247-256.

[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.

[SELT93] Seltzer, M., ''Transaction Support in a Log-Structured File System,'' To appear in the Proceedings of the 1993 International Conference on Data Engineering, April 1993, Vienna.

[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.

Author Information

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.