The Design and Implementation of a Log-Structured File System

Mendel Rosenblum
John K. Ousterhout

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

March 4, 1991

Abstract

This paper presents a new technique for disk storage management called a *log-structured file system*. A log-structured file system writes all modifications to disk sequentially in a log-like structure, thereby speeding up both file writing and crash recovery. The log is the only structure on disk; it contains indexing information so that files can be read back from the log efficiently. In order to maintain large free areas on disk for fast writing, we divide the log into *segments* and use a *segment cleaner* to compress the live information from heavily fragmented segments. We present a series of simulations that demonstrate the efficiency of a simple cleaning policy based on cost and benefit. We have implemented a prototype log-structured file system called Sprite LFS; it outperforms current Unix file systems by an order of magnitude for small-file writes while matching or exceeding Unix performance for reads and large writes. Even when the overhead for cleaning is included, Sprite LFS can use 70% of the disk bandwidth for writing, whereas Unix file systems typically can use only 5-10%.

The work described here was supported in part by the National Science Foundation under grant CCR-8900029, and in part by the National Aeronautics and Space Administration and the Defense Advanced Research Projects Agency under contract NAG2-591.

The Design and Implementation of a Log-Structured File System

1. Introduction

Over the last decade CPU speeds have increased dramatically while disk access times have only improved slowly. This trend is likely to continue in the future and it will cause more and more applications to become disk-bound. To lessen the impact of this problem, we have devised a new disk storage management technique called a *log-structured file system*, which uses disks an order of magnitude more efficiently than current file systems.

Log-structured file systems are based on the assumption that files are cached in main memory and that increasing memory sizes will make the caches more and more effective at satisfying read requests[1]. As a result, disk traffic will become dominated by writes. A log-structured file system writes all new information to disk in a sequential structure called the *log*. This approach increases write performance dramatically by eliminating almost all seeks. The sequential nature of the log also permits much faster crash recovery: current Unix file systems typically must scan the entire disk to restore consistency after a crash, but a log-structured file system need only examine the most recent portion of the log.

The notion of logging is not new, and a number of recent file systems have incorporated a log as an auxiliary structure to speed up writes and crash recovery[2-4]. However, these other systems use the log only for temporary storage; the permanent home for information is in a traditional random-access storage structure on disk. In contrast, a log-structured file system stores data permanently in the log: there is no other structure on disk. The log contains indexing information so that files can be read back with efficiency comparable to current file systems.

For a log-structured file system to operate efficiently, it must ensure that there are always large extents of free space available for writing new data. This is the most difficult challenge in the design of a log-structured file system. In this paper we present a solution based on large extents called *segments*, where a *segment cleaner* process continually regenerates empty segments by compressing the live data from heavily fragmented segments. We used a simulator to explore different cleaning policies and discovered a simple but effective algorithm based on cost and benefit: it segregates older, more slowly

changing data from young rapidly-changing data and treats them differently during cleaning.

We have constructed a prototype log-structured file system called Sprite LFS, which is now in production use as part of the Sprite network operating system[5]. Benchmark programs demonstrate that the raw writing speed of Sprite LFS is more than an order of magnitude greater than that of Unix for small files. Even for other workloads, such as those including reads and large-file accesses, Sprite LFS is at least as fast as Unix in all cases but one (files read sequentially after being written randomly). We also measured the long-term overhead for cleaning in the production system. Overall, Sprite LFS permits about 65-75% of a disk's raw bandwidth to be used for writing new data (the rest is used for cleaning). For comparison, Unix systems can only utilize 5-10% of a disk's raw bandwidth for writing new data; the rest of the time is spent seeking.

The remainder of this paper is organized into six sections. Section 2 reviews the issues in designing file systems for computers of the 1990's. Section 3 discusses the design alternatives for a log-structured file system and derives the structure of Sprite LFS, with particular focus on the cleaning mechanism. Section 4 describes the crash recovery system for Sprite LFS. Section 5 evaluates Sprite LFS using benchmark programs and long-term measurements of cleaning overhead. Section 6 compares Sprite LFS to other file systems, and Section 7 concludes.

2. Design Issues for File Systems of the 1990's

File system design, like many other aspects of computer science, is governed by two general forces: technology, which provides a set of basic building blocks, and workload, which determines a set of operations that must be carried out efficiently. This section summarizes the technology changes that are underway and describes their impact on file system design. It also describes the workloads that influenced the design of Sprite LFS and shows how current file systems are ill-equipped to deal with the workloads and technology changes.

2.1. Technology

Three components of technology are particularly significant for file system design: processors, disks, and main memory. Processors are significant because their speed is increasing at a nearly exponential rate, and the improvements seem likely to continue through much of the 1990's. This puts pressure on all the other elements of the computer system to speed up as well, so that the system doesn't become unbalanced.

Disk technology is also improving rapidly, but the improvements have been primarily in the areas of cost and capacity rather than performance. There are two components of disk performance: transfer bandwidth and access time. Although both of these factors are improving, the rate of improvement is much slower than for CPU speed. Disk transfer bandwidth can be improved substantially with the use of disk arrays and parallel-head disks[6] but no major improvements seem likely for access time (it is determinated by mechanical motions that are hard to improve). If an application causes a sequence of small disk transfers separated by seeks, then the application is not likely to experience much speedup over the next ten years, even with faster processors.

The third component of technology is main memory, which is increasing in size at an exponential rate. Modern file systems cache recently-used file data in main memory, and larger main memories make larger file caches possible. This has two effects on file system behavior. First, larger file caches alter the workload presented to the disk by absorbing a greater fraction of the read requests[1]. Most write requests must eventually be reflected on disk for safety, so disk traffic (and disk performance) will become more and more dominated by writes.

The second impact of large file caches is that they can serve as write buffers where large numbers of modified blocks can be collected before writing any of them to disk. Buffering may make it possible to write the blocks more efficiently, for example by writing them all in a single sequential transfer with one seek. Of course, write-buffering has the disadvantage of increasing the amount of data lost during a crash. For this paper we will assume that crashes are infrequent and that it is acceptable to lose a few

seconds or minutes of work in each crash; for applications that require better crash recovery, non-volatile RAM may be used for the write buffer.

2.2. Workloads

Several different file system workloads are common in computer applications, but the most challenging workload for file system design is that found in office and engineering environments. Office and engineering applications tend to be dominated by accesses to small files; several studies have measured mean file sizes of only a few kilobytes[1, 7, 8]. Small files usually result in small random disk I/Os, and the creation and deletion times for such files are often dominated by updates to file system ‘‘metadata’’ (the data structures used to locate the attributes and blocks of the file).

Workloads dominated by sequential accesses to large files, such as those found in supercomputing environments, also pose interesting problems, but not for file system software. A number of techniques exist for ensuring that such files are laid out sequentially on disk, so I/O performance tends to be limited by the bandwidth of the I/O and memory subsystems rather than the file allocation policies. In designing a log-structured file system we decided to focus on the efficiency of small-file accesses, and leave it to hardware designers to improve the efficiency of large-file accesses. Fortunately, the techniques used in Sprite LFS work even better for large files than they do for small ones.

2.3. Problems with Existing File Systems

Current file systems suffer from two general problems that make it hard for them to cope with the technologies and workloads of the 1990’s. First, they spread information around the disk in a way that causes too many small accesses. For example, the Berkeley fast file system[9] (Berkeley FFS) is quite effective at laying out each file sequentially on disk, but different files are placed at different locations. Furthermore, the attributes (‘‘inode’’) for a file are in a different location than the file, as is the directory entry containing the file’s name. It takes at least five separate disk I/O’s, each preceded by a seek, to create a new file in Berkeley FFS: two different accesses to the file’s attributes plus one access each for

the file's data, the directory's data, and the directory's attributes. When writing small files in such a system, less than 5% of the disk's potential bandwidth is used for new data; the rest of the time is spent seeking.

The second problem with current file systems is that they tend to write synchronously: the application must wait for the write to complete, rather than continuing while the write is handled in the background. Synchronous writes couple the application's performance to that of the disk and make it hard for the application to benefit from faster CPUs. They also defeat the potential use of the file cache as a write buffer. Unfortunately, network file systems like NFS[10] have introduced additional synchronous behavior where it didn't used to exist. This has simplified crash recovery, but it has reduced write performance.

3. Log-structured file systems

The fundamental idea of a log-structured file system is to improve write performance by buffering a sequence of file system changes in the file cache and then writing all the changes to disk sequentially in a single disk write operation. The information written to disk in the write operation includes file data blocks, attributes, index blocks, directories, and almost all the other information used to manage the file system. For workloads that contain many small files, a log-structured file system converts the many small synchronous random writes of traditional file systems into large asynchronous sequential transfers that can utilize nearly 100% of the raw disk bandwidth.

Although the basic idea of a log-structured file system is simple, there are two key issues that must be resolved to achieve the potential benefits of the logging approach. The first issue is how to retrieve information from the log; this is the subject of Section 3.1 below. The second issue is how to manage the free space on disk so that large extents of free space are always available for writing new data. This is a much more difficult issue; it is the topic of Sections 3.2-3.6. Table 1 contains a summary of the on-disk data structures used by Sprite LFS to solve the above problems; the data structures are discussed in detail in later sections of the paper.

3.1. File location and reading

Although the term ‘‘log-structured’’ might suggest that sequential scans are required to retrieve information from the log, this is not the case in Sprite LFS. Our goal was to match or exceed the read performance of the Unix file system. To accomplish this goal, Sprite LFS outputs index structures in the log to permit random-access retrievals. The basic structures used by Sprite LFS are identical to those used in the Unix file system[9]: for each file there exists a data structure called an *inode*, which contains the file's attributes (type, owner, permissions, etc.) plus the disk addresses of the first ten blocks of the file; for files larger than ten blocks, the inode also contains the disk addresses of one or more *indirect blocks*, each of which contains the addresses of more data or indirect blocks. Once a file's inode has been found, the number of disk I/Os required to read the file is identical in Sprite LFS and Unix.

Data structure	Purpose	Location	Section
Inode	Locates blocks of file, holds protection bits, modify time, etc.	Log	3.1
Inode map	Locates position of inode in log, holds time of last access plus version number.	Log	3.1
Indirect block	Locates blocks of large files.	Log	3.1
Segment summary	Identifies contents of segment (file number and offset for each block).	Log	3.2
Segment usage table	Counts live bytes still left in segments, stores last write time for data in segments.	Log	3.6
Superblock	Holds static configuration information such as number of segments and segment size.	Fixed	None
Checkpoint region	Locates blocks of inode map and segment usage table, identifies last checkpoint in log.	Fixed	4.1
Directory change log	Records directory operations to maintain consistency of reference counts in inodes.	Log	4.2

Table 1 — Summary of the major data structures stored on disk by Sprite LFS.
For each data structure the table indicates the purpose served by the data structure in Sprite LFS. The table also indicates whether the data structure is stored in the log or at a fixed position on disk and where in the paper the data structure is discussed in detail. Inodes, indirect blocks, and superblocks are similar to the Unix file system data structures with the same names. Note that Sprite LFS contains neither a bitmap nor a free list.

In Unix file systems each inode is at a fixed location on disk; given the identifying number for a file, a simple calculation yields the disk address of the file's inode. In contrast, Sprite LFS doesn't place inodes at fixed positions; they are written to the log just like everything else. Sprite LFS uses a data structure called the *inode map* to maintain the current location of each inode. Given the identifying number for a file, the inode map must be indexed to determine the disk address of the inode. The inode map is divided into blocks that are written to the log; a fixed checkpoint region on each disk identifies the locations of all the inode map blocks. Fortunately, inode maps are compact enough to keep the active portions cached in main memory: inode map lookups rarely require disk accesses.

Figure 1 shows the disk layouts that would occur in Sprite LFS and a typical Unix file system after creating two new files in different directories. Although the two layouts have the same logical structure, the log-structured file system produces a much more compact arrangement. As a result, the write performance of Sprite LFS is much better than Unix, while its read performance is just as good.

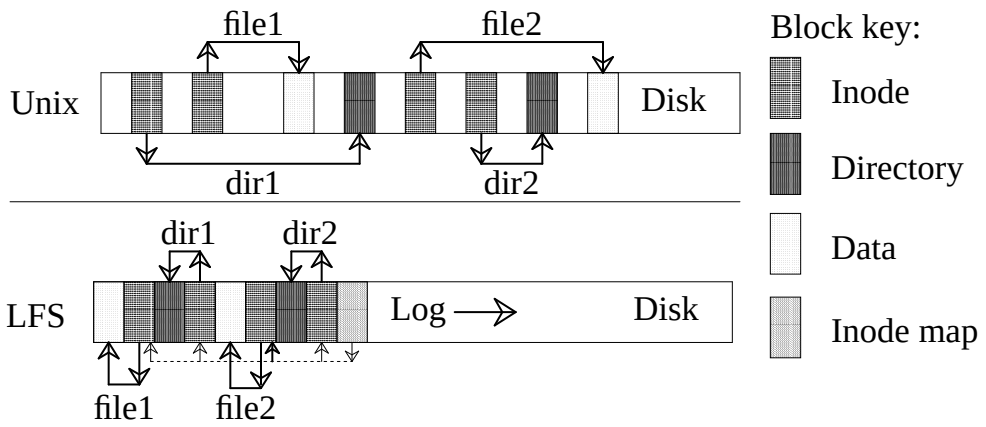

Figure 1 — A comparison between Sprite LFS and the Unix file system.
This example shows the modified disk blocks written by Sprite LFS and a typical Unix file system when creating two single-block files named `dir1/file1` and `dir2/file2`. Each system must write new data blocks and inodes for `file1` and `file2`, plus new data blocks and inodes for the containing directories. Unix requires ten non-sequential writes for the new information (the inodes for the new files are each written twice), while Sprite LFS performs the operations in a single large write. The same number of disk accesses will be required to read the files in the two systems. Sprite LFS also writes out new inode map blocks to record the new inode locations.

3.2. Free space management: segments

The most important and interesting issue in the design of a log-structured file system is how to manage free space. The goal is to maintain large free extents for writing new data. Initially all the free space is in a single extent on disk, but by the time the log reaches the end of the disk the free space will have been fragmented into many small extents corresponding to the files that were deleted.

From this point on, the file system has two choices: threading and copying. These are illustrated in Figure 2. The first alternative is to leave the live data in place and thread the log through the free extents. Unfortunately, threading will cause the free space to become severely fragmented, so that large contiguous writes won't be possible and a log-structured file system will be no faster than traditional file systems. The second alternative is to copy live data out of the log in order to leave large free extents for writing. For this paper we will assume that the live data is written back in a compacted form at the head of the log; it could also be moved to another log-structured file system to form a hierarchy of logs, or it could be moved to some totally different file system or archive. The disadvantage of copying is its cost, particu-

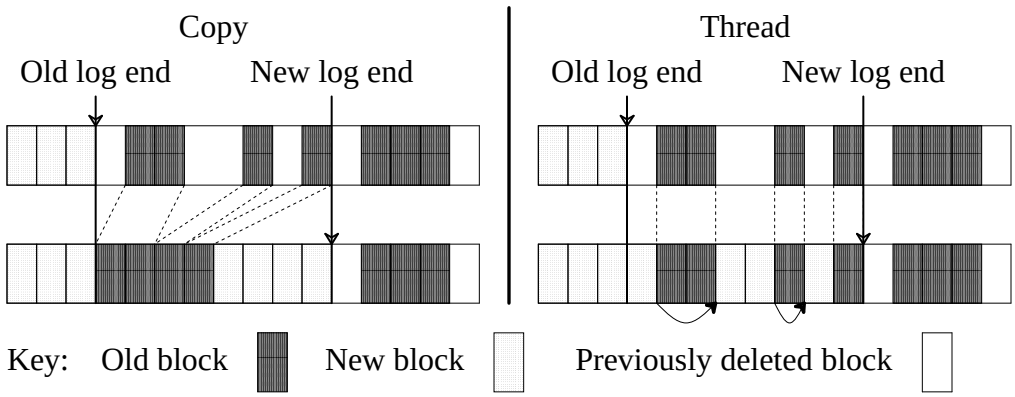

Figure 2 — Possible free space management solutions for log-structured file systems.
In a log-structured file system, free space for the log can be generated either by copying the old blocks or by threading the log around the old blocks. This figure shows the old blocks copied back into the log or threaded around in place.

larly for long-lived files; in the simplest case where the log works circularly across the disk and live data is copied back into the log, all of the long-lived files will have to be copied in every pass of the log across the disk.

Sprite LFS uses a combination of the threading and copying approaches. The disk is partitioned into large fixed-size extents called *segments*. Any given segment is always written sequentially from its beginning to its end, and all live data must be copied out of a segment before the segment can be written. However, the log is threaded on a segment-by-segment basis; if the system can collect long-lived data together into segments, those segments can be skipped over so that the data doesn't have to be copied repeatedly. The segment size is chosen large enough that the transfer time to read or write a whole segment is much greater than the cost of a seek to the beginning of the segment. This allows whole-segment operations to run at nearly the full bandwidth of the disk, regardless of the order in which segments are accessed. Sprite LFS currently uses segment sizes of either 512 Kbytes or one megabyte.

3.3. Segment cleaning mechanism

The process of copying live data out of a segment is called *segment cleaning*. In Sprite LFS it is a simple three-step process: read a number of segments into memory, identify the live data, and write the live data back to a smaller number of clean segments. After this operation is complete, the segments that were read are marked as clean, and they can be used for new data or for additional cleaning.

As part of segment cleaning it must be possible to identify which blocks of each segment are live, so that they can be written out again. It must also be possible to identify the file to which each block belongs and the position of the block within the file; this information is needed in order to update the file's inode to point to the new location of the block. Sprite LFS solves both of these problems by including a *segment summary block* to each log write to a segment. The summary block identifies each piece of information that is written in the segment; for example, for each file data block the summary block contains the file number and block number for the block. Segments can contain multiple segment summary blocks when more than one log write is needed to fill the segment. Non-segment size log writes occur

when the number of dirty blocks buffered in the file cache is insufficient to fill a segment. Segment summary blocks impose little overhead during writing, and they are useful during crash recovery (see Section 4) as well as during cleaning.

Sprite LFS also uses the segment summary information to distinguish live blocks from those that have been overwritten or deleted. Once a block's identity is known, its liveness can be determined by checking the file's inode or indirect block to see if the appropriate block pointer still refers to this block. If it does, then the block is live; if it doesn't, then the block is dead. Sprite LFS optimizes this check slightly by keeping a version number in the inode map entry for each file; the version number is incremented whenever the file is deleted or truncated to length zero. The segment summary block records the file version number for each block in the segment; if the version number is no longer current when the segment is cleaned, the block can be discarded immediately without examining the file's inode.

This approach to cleaning means that there is no free-block list or bitmap in Sprite. In addition to saving memory and disk space, the elimination of these data structures also simplifies crash recovery by eliminating the code to recover them.

The segment cleaning implementation in Sprite LFS uses the normal file cache code during cleaning. Once live blocks have been read in and identified, they are linked into the file cache. This use of the file cache synchronizes cleaning with other accesses to files and allows the file cache write-back code to be used to write the cleaned blocks to disk. As a result, segment cleaning can be performed at any time and in parallel with other file accesses.

3.4. Segment cleaning policies

Given the basic mechanism described above, four policy issues must be addressed:

(1) When should the segment cleaner execute? Some possible choices are for it to run continuously in background at low priority, or only at night, or only when disk space is nearly exhausted.

(2) How many segments should it clean at a time? Segment cleaning offers an opportunity to reorganize data on disk; the more segments cleaned at once, the more opportunities to rearrange.

(3) Which segments should be cleaned? An obvious choice is the ones that are most fragmented, but this turns out not to be the best choice.

(4) How should the live blocks be grouped when they are written out? One possibility is to try to enhance the locality of future reads, for example by grouping files in the same directory together into a single output segment. Another possibility is to sort the blocks by the time they were last modified and group blocks of similar age together into new segments; we call this approach *age sort*.

In our work so far we have not methodically addressed the first two of the above policies. Sprite LFS starts cleaning segments when the number of clean segments drops below a threshold value (typically a few tens of segments). It cleans a few tens of segments at a time until the number of clean segments surpasses another threshold value (typically 50-100 clean segments). The overall performance of Sprite LFS does not seem to be very sensitive to the exact choice of the threshold values. In contrast, the third and fourth policy decisions are critically important: in our experience they are the primary factors that determine the performance of a log-structured file system. The remainder of Section 3 discusses our analysis of which segments to clean and how to group the live data.

We use a term called *write cost* to compare cleaning policies. The write cost is the average amount of time the disk is busy per byte of new data written, including all the cleaning overheads. The write cost is expressed as a multiple of the time that would be required if there were no cleaning overhead and the data could be written at its full bandwidth with no seek time or rotational latency. A write cost of 1.0 is perfect: it would mean that new data could be written at the full disk bandwidth and there was no cleaning overhead. A write cost of 10 means that only one-tenth of the disk's maximum bandwidth was actually used for writing new data; the rest of the disk time was spent in seeks, rotational latency, or cleaning.

For a log-structured file system with large segments, seeks and rotational latency are negligible both for writing and for cleaning, so the write cost is the total number of bytes moved to and from the disk divided by the number of those bytes that represent new data. This cost is determined by the utilization (the fraction of data still live) in the segments that are cleaned. In the steady state, the cleaner must generate one clean segment for every segment of new data written. To do this, it reads N segments in their entirety and writes out N**u* segments of live data (where *u* is the utilization of the segments and 0 <= *u* < 1). This creates N*(1–*u*) segments of contiguous free space for new data. Thus

$$\text{write cost} = \frac{\text{total bytes read and written}}{\text{new data written}} = \frac{\text{read segs + write live + write new}}{\text{new data written}}$$
$$= \frac{N + N*u + N*(1-u)}{N*(1-u)} = \frac{2}{1-u} \quad (1)$$

In the above formula we made the conservative assumption that a segment must be read in its entirety to recover the live blocks; in practice it may be faster to read just the live blocks, particularly if the utilization is very low (we haven't tried this in Sprite LFS). If cleaned segments have no live blocks (u = 0) then they need not be read at all and the write cost is 1.0.

Figure 3 graphs the write cost as a function of u. For reference, conventional Unix file systems on small-file workloads typically utilize only 5-10% of the disk bandwidth, for a write cost of 10-20 (see [11] and Figure 8 in Section 5.1 for specific measurements). With logging, delayed writes, and disk request sorting this can probably be improved to about 25% of the bandwidth or a write cost of 4. Figure 3 suggests that the segments cleaned must have a utilization of less than .8 in order for a log-structured

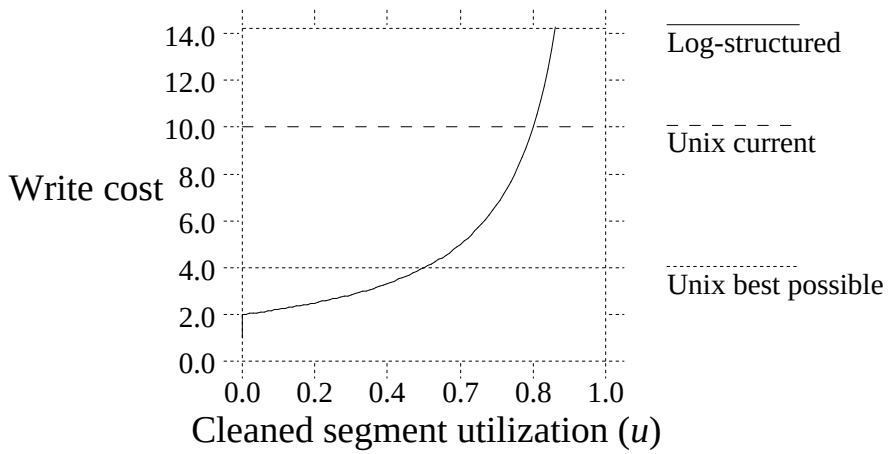

Figure 3 — Write cost as a function of utilization, for small files.
In a log-structured file system, the write cost depends strongly on the utilization of the segments that are cleaned. The figure also shows two reference points: ''Unix current'', which represents typical Unix file systems today, and ''Unix best possible'', which is our estimate of the best performance possible in an improved Unix file system. Write cost for the Unix file systems is not sensitive to the utilization.

file system to outperform current Unix systems; the utilization must be less than .5 to outperform an improved Unix system.

It is important to note that the utilization discussed above is not the overall fraction of the disk that is alive; it is just the fraction of live blocks in segments that are cleaned. Variations in file usage will cause some segments to be less utilized than others, and the cleaner can choose the least utilized segments to clean; these will have lower utilization than the overall average for the disk.

Even so, the performance of a log-structured file system can be improved by reducing the overall utilization of the disk. Log-structured file systems provide a cost-performance tradeoff: if disk space is underutilized, higher performance can be achieved but at a high cost per usable byte; if disk utilization is increased, storage costs are reduced but so is performance.

The key to achieving high performance at low cost in a log-structured file system is somehow to force the disk into a bimodal segment distribution where most of the segments are nearly full, a few are empty or nearly empty, and the cleaner can almost always work with the empty segments. This would allow a high overall disk utilization yet provide a low write cost. The following section describes how we achieve such a bimodal distribution in Sprite LFS.

3.5. Simulation results

We built a simple file system simulator so that we could analyze different cleaning policies under controlled conditions. The simulator's model does not reflect actual file system usage patterns (its model is much harsher than reality), but it helped us to understand the effects of random access patterns and locality, both of which can be exploited to reduce the cost of cleaning. The simulator models a file system as a fixed number of 4-kbyte files, with the number chosen to produce a particular overall disk utilization. At each step, the simulator overwrites one of the files with new data, using one of two pseudo-random access patterns:

Uniform Each file has equal likelihood of being selected in each step.

Hot-and-cold Files are divided into two groups. One group contains 10% of the files; it is called *hot* because its files are selected 90% of the time. The other group is called *cold*; it contains 90% of the files and they are selected only 10% of the time. Within groups each file is equally likely to be selected. This access pattern models a simple form of locality.

In this approach the overall disk utilization is constant and no read traffic is modelled. The simulator runs until all clean segments are exhausted, then simulates the actions of a cleaner until a threshold number of clean segments is available again. In each run the simulator was allowed to run until the write cost stablized and all cold-start variance had been removed.

Figure 4 superimposes the results from two sets of simulations onto the curves of Figure 3. In the

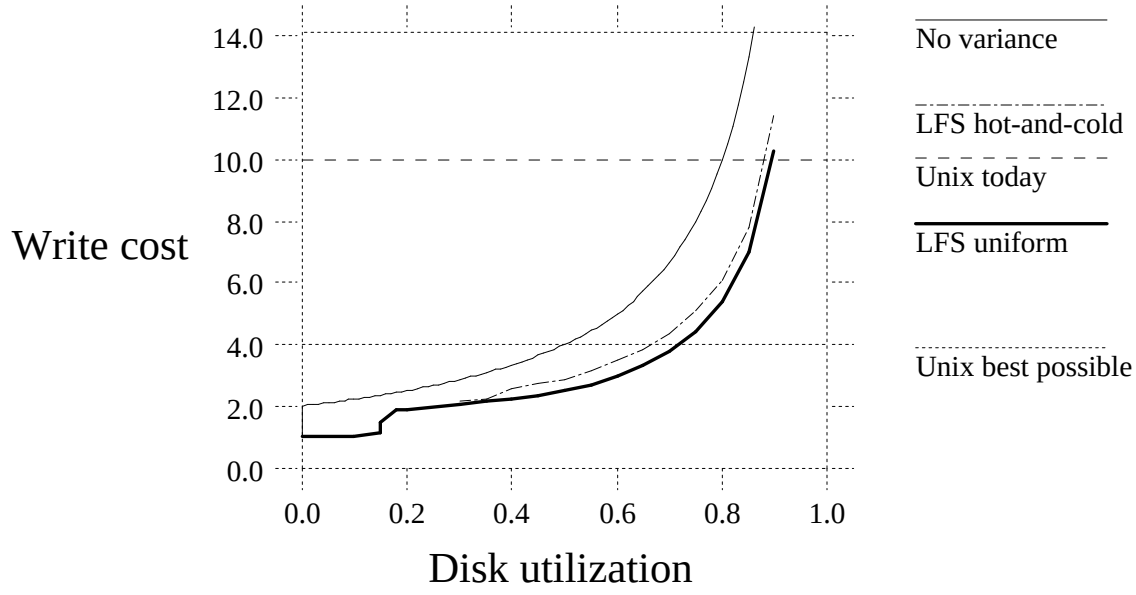

Figure 4 — Initial simulation results.
The curves labelled ‘‘Unix today’’ and ‘‘Unix best possible’’ are reproduced from Figure 3 for comparison. The curve labelled ‘‘No variance’’ shows the write cost that would occur if all segments always had exactly the same utilization. The ‘‘LFS uniform’’ curve represents a log-structured file system with uniform access pattern and a greedy cleaning policy: the cleaner chooses the least-utilized segments. The ‘‘LFS hot-and-cold’’ curve represents a log-structured file system with locality of file access. It uses a greedy cleaning policy, and the cleaner also sorts the live data by age before writing it out again. The x-axis is overall disk utilization, which is not necessarily the same as the utilization of the segments being cleaned.

‘‘LFS uniform’’ simulations the uniform access pattern was used. The cleaner used a simple greedy policy where it always chose the least-utilized segments to clean. When writing out live data the cleaner did not attempt to re-organize the data: live blocks were written out in the same order that they appeared in the segments being cleaned (for a uniform access pattern there is no reason to expect any improvement from re-organization).

Even with uniform random access patterns, the variance in segment utilization allows a substantially lower write cost than would be predicted from the overall disk utilization and formula (1). For example, at 75% overall disk utilization, the segments cleaned have an average utilization of only 55%. At overall disk utilizations under 25% the write cost drops below 2.0; this means that some of the cleaned segments have no live blocks at all and hence don’t need to be read in.

The ‘‘LFS hot-and-cold’’ curve shows the write cost when there is locality in the access patterns, as described above. The cleaning policy for this curve was the same as for ‘‘LFS uniform’’ except that the live blocks were sorted by age before writing them out again. This means that long-lived (cold) data tends to be segregated in different segments from short-lived (hot) data; we thought that this approach would lead to the desired bimodal distribution of segment utilizations.

Figure 4 shows the surprising result that locality and ‘‘better’’ grouping result in *worse* performance than a system with no locality! We tried varying the degree of locality (e.g. 95% of accesses to 5% of data) and found that performance got worse and worse as the locality increased. Figure 5 shows the reason for this non-intuitive result. Under the greedy policy, a segment doesn’t get cleaned until it becomes the least utilized of all segments. Thus every segment’s utilization eventually drops to the cleaning threshold, including the cold segments. Unfortunately, the utilization drops very slowly in cold segments, so these segments tend to linger just above the cleaning point for a very long time. Figure 5 shows that many more segments are clustered around the cleaning point in the simulations with locality than in the simulations without locality. The overall result is that cold segments tend to tie up large numbers of free blocks for long periods of time.

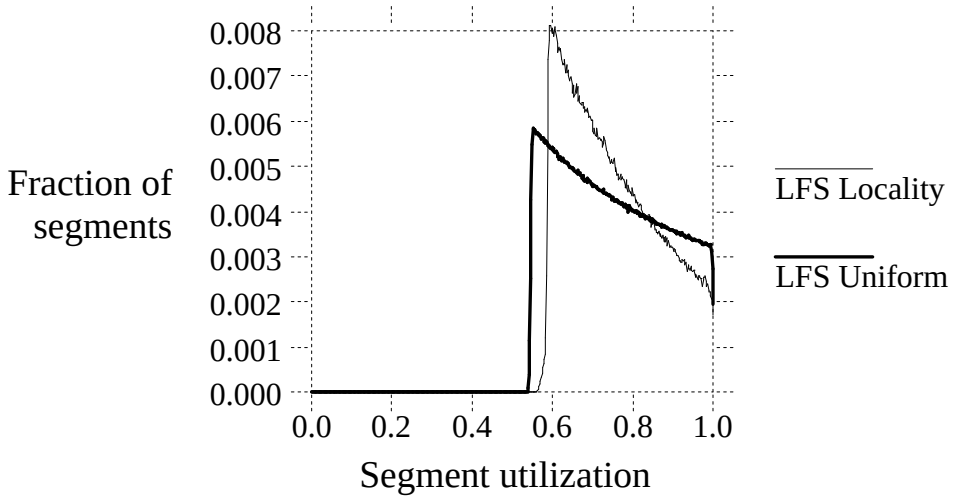

Figure 5 — Segment utilization distributions with greedy cleaner.
These figures show distributions of segment utilizations of the disk during the simulation. The distribution is computed by observing the utilizations of the segments on the disk at the points during the simulation when segment cleaning is initiated. The distribution shows the utilizations of the segments available to the cleaning algorithm. Each of the distributions corresponds to an overall disk utilization of 75%. The ‘‘LFS Uniform’’ curve corresponds to ‘‘LFS Uniform’’ in Figure 4 and ‘LFS Locality’’ corresponds to ‘‘LFS hot-and-cold’’ in Figure 4. Locality causes the distribution to be more skewed towards the utilization at which cleaning occurs; as a result, segments are cleaned at a higher average utilization.

After studying these figures we realized that hot and cold segments must be treated differently by the cleaner. Free space in a cold segment is more valuable than free space in a hot segment because once a cold segment has been cleaned it will take a long time before the cold data becomes fragmented again and re-accumulates the unusable free space. Said another way, once the system reclaims the free blocks it will get to ‘‘keep’’ them a long time before the segment ‘‘takes them back again.’’ In contrast, it is less beneficial to clean a hot segment because free space will rapidly re-accumulate; the system might as well delay the cleaning a while and let more of the blocks die in the current segment.

To test this theory we simulated a new policy for selecting segments to clean. The policy rates each segment according to the benefit of cleaning the segment and the cost of cleaning the segment and chooses the segments with the highest ratio of benefit to cost. The benefit has two components: the amount of space that will be freed and the amount of time the space is likely to stay free. The amount of

space to be freed is just 1−*u*, where *u* is the utilization of the segment. We used the most recent modified time of any block in the segment as an estimate of how long the space is likely to stay free. The cost of cleaning the segment is 1+*u* (one unit of cost to read the segment, *u* to write back the live data). Combining all these factors, we get

$$\frac{\text{benefit}}{\text{cost}} = \frac{\text{free space generated} * \text{age of data}}{\text{cost}} = \frac{(1-u)*\text{age}}{1+u} \tag{2}$$

We call this policy the *cost-benefit* policy; it allows cold segments to be cleaned at a much higher utilization than hot segments.

We re-ran the simulations under the hot-and-cold access pattern with the cost-benefit policy and age-sorting on the live data. As can be seen from Figure 6, the cost-benefit policy produced the bimodal distribution of segments that we had hoped for. The cleaning policy cleans cold segments at about 75% utilization but waits until hot segments reach a utilization of about 15% before cleaning them. Since 90%

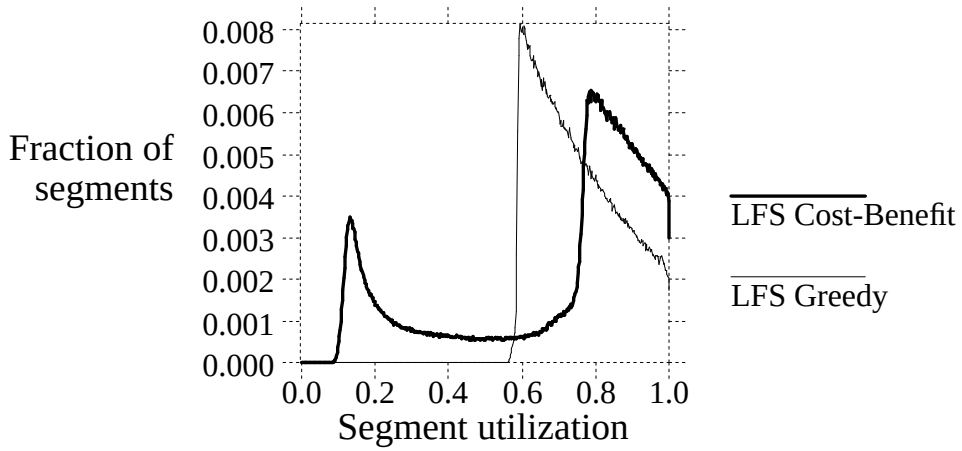

Figure 6 — Segment utilization distribution with cost-benefit policy.
This figure shows the distribution of segment utilizations from the simulation of a hot-and-cold access pattern with 75% overall disk utilization. The ‘‘LFS Cost-Benefit’’ curve shows the segment distribution occurring when the cost-benefit policy is used to select segments to clean, and live blocks being grouped by age before being re-written. Because of this bimodal segment distribution, most of the segments cleaned had utilizations around 15%. For comparison, the greedy method selection policy is shown by the ‘‘LFS Greedy’’ curve reproduced from Figure 5.

of the writes are to hot files, most of the segments cleaned are hot. Figure 7 shows that the cost-benefit policy reduces the write cost by as much as 50% over the greedy policy, and a log-structured file system out-performs the best possible Unix system even at relatively high disk utilizations. We simulated a number of other degrees and kinds of locality and found that the cost-benefit policy gets even better as locality increases.

The simulation experiments convinced us to implement the cost-benefit approach in Sprite LFS. As will be seen in Section 5.2, the behavior of actual file systems in Sprite LFS is even better than predicted in Figure 7.

3.6. Segment usage table

In order to support the cost-benefit cleaning policy, Sprite LFS maintains a data structure called the *segment usage table*. For each segment, the table records the number of live bytes in the segment and the most recent modified time of any block in the segment. These two values are used by the segment cleaner when choosing segments to clean. The values are initially set when the segment is written, and the count of live bytes is gradually decremented when files are deleted or blocks are overwritten. If the count falls to zero then the segment can be reused without cleaning. The blocks of the segment usage table are written to the log, and the addresses of the blocks are stored in the checkpoint regions (see Section 4 for details).

In order to sort live blocks by age, the segment summary information must record the time at which each block was written. At present Sprite LFS does not keep this information; it uses the time when the segment was written as the modified time for all blocks in the segment (but this is inaccurate for segments that have been cleaned). We plan to modify the segment summary information to include modified times for each block.

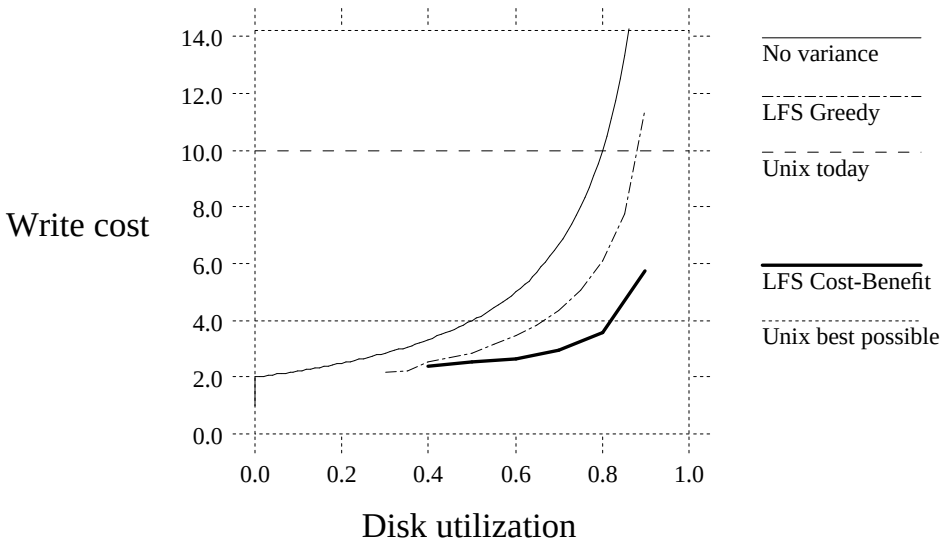

Figure 7 — Write cost, including cost-benefit policy.
This graph compares the write cost of the greedy policy with that of the cost-benefit policy, for the hot-and-cold access pattern. The cost-benefit policy is substantially better than the greedy policy, particularly for disk utilizations of about 60%.

4. Crash recovery

When a system crash occurs, the last few operations performed on the disk may have left it in an inconsistent state; during reboot the operating system must review these operations in order to correct any inconsistencies. In traditional Unix file systems without logs, the system cannot determine where the last changes were made so it must scan all of the metadata structures on disk to restore consistency. The cost of these scans is already high (tens of minutes in typical configurations) and it is getting higher as storage systems expand.

In a log-structured file system the locations of the last disk operations are easy to determine: they are at the end of the log. Thus it should be possible to recover very quickly after crashes. This benefit of logs is well known and has been used to advantage both in database systems [12] and in other file systems [2, 4, 13]. Like many other logging systems, Sprite LFS uses a two-pronged approach to recovery:

checkpoints, which define consistent states of the file system, and *roll-forward*, which is used to recover information written since the last checkpoint.

4.1. Checkpoints

A checkpoint is a position in the log at which all of the file system structures are consistent and complete. Sprite LFS uses a two-phase process to create a checkpoint. First, it writes out all modified information to the log, including file data blocks, indirect blocks, inodes, and blocks of the inode map and segment usage table. Second, it writes a *checkpoint region* to a special fixed position on disk. The checkpoint region contains the addresses of all the blocks in the inode map and segment usage table, plus the current time and a pointer to the last segment written.

During reboot, Sprite LFS reads the checkpoint region and uses that information to initialize its main-memory data structures. In order to handle a crash during a checkpoint operation there are actually two checkpoint regions, and checkpoint operations alternate between them. The checkpoint time is in the last block of the checkpoint region, so if the checkpoint fails the time will not be updated. During reboot, the system reads both checkpoint regions and uses the one with the most recent time.

Sprite LFS performs checkpoints at periodic intervals as well as when the file system is unmounted or the system is shut down. A long interval between checkpoints reduces the overhead of writing the checkpoints but increases the time needed to roll forward during recovery; a short checkpoint interval improves recovery time but increases the cost of normal operation. Sprite LFS currently uses a checkpoint interval of thirty seconds, which is probably much too short. An alternative to periodic checkpointing is to perform checkpoints after a given amount of new data has been written to the log; this would set a limit on recovery time while reducing the checkpoint overhead when the file system is idle.

4.2. Roll-forward

In principle it would be safe to restart after crashes by simply reading the latest checkpoint region and discarding any data in the log after that checkpoint. This would result in instantaneous recovery but

any data written since the last checkpoint would be lost. In order to recover as much information as possible, Sprite LFS scans through the log segments that were written after the last checkpoint. This operation is called *roll-forward*.

During roll-forward Sprite LFS uses the information in segment summary blocks to recover recently-written file data. When a summary block indicates the presence of a new inode, Sprite LFS updates the inode map it read from the checkpoint, so that the inode map refers to the new copy of the inode. This automatically incorporates the file's new data blocks into the recovered file system. If data blocks are discovered for a file without a new copy of the file's inode, then the roll-forward code assumes that the new version of the file on disk is incomplete and it ignores the new data blocks.

The roll-forward code also adjusts the utilizations in the segment usage table read from the checkpoint. The utilizations of the segments written since the checkpoint will all be zero; they must be adjusted to reflect the live data left after roll-forward. The utilizations of older segments will also have to be adjusted to reflect file deletions and overwrites (both of these can be identified by the presence of new inodes in the log).

The final issue in roll-forward is how to restore consistency between directory entries and inodes. Each inode contains a count of the number of directory entries referring to that inode; when the count drops to zero the file is deleted. Unfortunately, it is possible for a crash to occur when an inode has been written to the log with a new reference count but the block containing the corresponding directory entry has not been written yet, or vice versa.

To restore consistency between directories and inodes, Sprite LFS outputs a special record in the log for each directory change. The record includes an operation code (create, link, rename, or unlink), the location of the directory entry (i-number for the directory and the position within the directory), the contents of the directory entry (name and i-number), and the new reference count for the inode named in the entry. These records are collectively called the *directory operation log*; Sprite LFS guarantees that each directory operation log entry appears in the log before the corresponding directory block or inode.

During roll-forward, the directory operation log is used to ensure consistency between directory entries and inodes: if a log entry appears but the inode and directory block were not both written, roll-forward updates the directory and/or inode to complete the operation. The only operation that can't be completed is the creation of a new file for which the inode is never written; in this case the directory entry will be removed. In addition to its other functions, the directory log made it easy to provide an atomic rename operation.

The interaction between the directory operation log and checkpoints introduced additional synchronization issues into Sprite LFS. In particular, each checkpoint must represent a state where the directory operation log is consistent with the inode and directory blocks in the log. This required additional synchronization to prevent directory modifications while checkpoints are being written.

5. Experience with the Sprite LFS prototype

We began the implementation of Sprite LFS in late 1989 and by mid-1990 it was operational as part of the Sprite network operating system. Since the fall of 1990 it has been used to manage three different disk partitions, which are used by about twenty users for day-to-day computing. All of the features described in this paper have been implemented in Sprite LFS, but roll-forward has not yet been installed in the production system. The production disks use a short checkpoint interval (30 seconds) and discard all the information after the last checkpoint when they reboot. (Note to reviewers: roll-forward will be installed shortly, and we will also convert another 6-8 disk partitions to LFS, so there will be more data available for the final version of the paper.)

When we began the project we were concerned that a log-structured file system might be substantially more complicated to implement than a traditional file system. In reality, however, Sprite LFS turns out to be no more complicated than Berkeley FFS[9]: Sprite LFS has additional complexity for the segment cleaner, but this is compensated by the elimination of the bit map and layout policies required by FFS; in addition, the checkpointing and roll-forward code in Sprite LFS are no more complicated than the *fsck* code[14] that scans FFS disks to restore consistency. Logging file systems like Episode[2] or

Echo[3] are likely to be somewhat more complicated than either FFS or Sprite LFS, since they include both logging and layout code.

5.1. Micro-benchmarks

We used a collection of small benchmark programs to measure the best-case performance of Sprite LFS and compare it to SunOS 4.0.3, whose file system is based on Berkeley FFS. The benchmarks are synthetic so they do not represent realistic workloads, but they illustrate the strengths and weaknesses of the two file systems. The machine used for both systems was a Sun-4/260 (16.6Mhz SPARC CPU) with 32 megabytes of memory, a Sun SCSI3 HBA, and a WREN IV disk (1.3 MBytes/sec maximum transfer bandwidth, 17.5 milliseconds average seek time). For both LFS and SunOS, the disk was formatted with a file system having around 300 megabytes of usable storage. An eight-kilobyte block size was used by SunOS while Sprite LFS used a four-kilobyte block size and a one-megabyte segment size. In each case the system was running multiuser but was otherwise quiescent during the test. For Sprite LFS, no cleaning occurred during the benchmark runs so the measurements represent best-case performance; see Section 5.2 below for measurements of cleaning overhead.

Figure 8 shows the results of a benchmark that creates, reads, and deletes a large number of small files. Sprite LFS is almost ten times as fast as SunOS for the create and delete phases of the benchmark. Sprite LFS is also faster for reading the files back; this is because the files are read in the same order created and the log-structured file system packs the files densely in the log. Furthermore, Sprite LFS only kept the disk 17% busy during the create phase while saturating the CPU. In contrast, SunOS kept the disk busy 85% of the time during the create phase, even though only about 1.2% of the disk's potential bandwidth was used for new data. This means that the performance of Sprite LFS will improve by another factor of 4-6 as CPUs get faster (see Figure 8(b)), whereas almost no improvement can be expected in SunOS.

Although Sprite was designed for efficiency on workloads with many small file accesses, Figure 9 shows that it also provides competitive performance for large files. Sprite LFS has a higher write

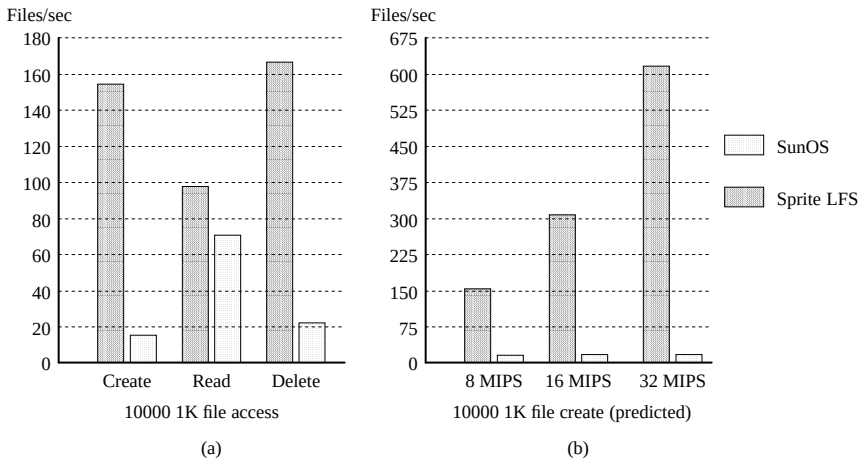

Figure 8 — Small-file performance under Sprite LFS and SunOS.
Figure (a) measures a benchmark that created 10000 one-kilobyte files, then read them back in the same order as created, then deleted them. Speed is measured by the number of files per second for each operation on the two file systems. The logging approach in Sprite LFS provides an order-of-magnitude speedup for creation and deletion. Figure (b) estimates the performance of each system for creating files on faster computers with the same disk. In SunOS the disk was 85% saturated in (a), so faster processors will not improve performance much. In Sprite LFS the disk was only 17% saturated in (a) while the CPU was 100% utilized; as a consequence, I/O performance scales with CPU speed.

bandwidth than SunOS in all cases. It is substantially faster for random writes because it turns them into sequential writes to the log; it is also faster for sequential writes because it groups many blocks into a single large I/O, whereas SunOS performs individual disk operations for each block (a newer version of SunOS groups writes [15] and should therefore have performance equivalent to Sprite LFS). The read performance is similar in the two systems except for the case of reading a file sequentially after it has been written randomly; in this case the reads require seeks in Sprite LFS, so its performance is substantially lower than SunOS.

Figure 9 illustrates the fact that a log-structured file system produces a different form of locality on disk than traditional file systems. A traditional file system achieves *logical locality* by assuming certain access patterns (sequential reading of files, a tendency to use multiple files within a directory, etc.); it then pays extra on writes, if necessary, to organize information optimally on disk for the assumed read pat-

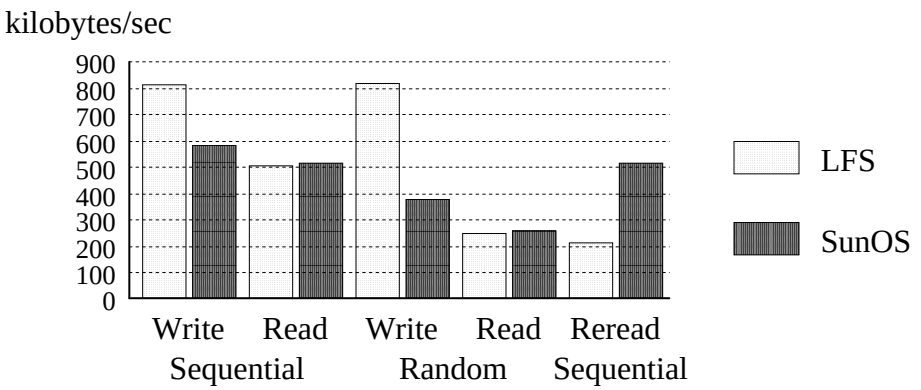

Figure 9 — Large-file performance under Sprite LFS and SunOS.
The figure shows the speed of a benchmark that creates a 100-Mbyte file with sequential writes, then reads the file back sequentially, then writes 100 Mbytes randomly to the existing file, then reads 100 Mbytes randomly from the file, and finally reads the file sequentially again. The bandwidth of each of the five phases is shown separately. Sprite LFS has a higher write bandwidth and the same read bandwidth as SunOS with the exception of sequential reading of a file that was written randomly.

terns. In contrast, a log-structured file system achieves *temporal locality*: information that is created or modified at the same time will be grouped closely on disk. If temporal locality matches logical locality, as it does for a file that is written sequentially and then read sequentially, then a log-structured file system should have about the same performance on large files as a traditional file system. If temporal locality differs from logical locality then the systems will perform differently. Sprite LFS handles random writes more efficiently because it writes them sequentially on disk. SunOS pays more for the random writes in order to achieve logical locality, but then it handles sequential re-reads more efficiently. Random reads have about the same performance in the two systems, even though the blocks are laid out very differently. However, if the random reads occur in the same order as the random writes then Sprite would have been much faster.

5.2. Cleaning overheads

The micro-benchmark results of the previous section give an optimistic view of the performance of Sprite LFS because they do not include any cleaning overheads (the write cost during the benchmark runs

was 1.0). In order to assess the cost of cleaning and the effectiveness of the cost-benefit cleaning policy, we recorded statistics about our production log-structured file systems over a period of several months. Three file systems were measured:

`/user5` Home directories for Sprite developers. Workload consists of program development, text processing, electronic communication, and simulations.

`/pcs` Home directories and project area for research on parallel processing and VLSI circuit design.

`/swap2` Sprite client workstation swap files. Workload consists of virtual memory backing store for 40 diskless Sprite workstations. Files tend to be large, sparse, and accessed nonsequentially.

Table 2 provides some summary statistics describing these file systems.

Table 3 shows statistics gathered during cleaning over a four-month period. In order to eliminate start-up effects we waited several months after putting the file systems into use before beginning the measurements. The behavior of the production file systems has been substantially better than what was predicted by the simulations in Section 3. Even though the overall disk utilizations ranged from 55-75%, more than half of the segments cleaned were totally empty. Even the non-empty segments have utilizations far less than the average disk utilizations. The overall write costs ranged from 1.3 to 1.6, in comparison to write costs of 2.5-3 in the corresponding simulations. Figure 10 shows the distribution of segment utilizations, gathered in a recent snapshot of the `/user5` disk.

Sprite LFS file systems					
File system	Size	In Use	Files	Dirs	Avg File Size
/user5	630 Mbytes	68%	22473	1865	15.5 Kbytes
/pcs	990 Mbytes	55%	44815	3937	10.5 Kbytes
/swap2	309 Mbytes	65%	3682	703	68.1 Kbytes

Table 2 — Summary of Sprite LFS file systems.
For each file system, the table lists the total capacity of the file system (Size), the overall fraction of disk space in use (In Use), the total number of files in the file system, the number of directories, and the average file size. File sizes range from zero to 100 megabytes with most files fairly small.

Write cost in Sprite LFS file systems					
		Segments		*u*	Write
File system	In Use	Cleaned	Empty	Avg	Cost
/user5	73%	17827	51%	.183	1.4
/pcs	55%	4817	77%	.130	1.3
/swap2	65%	4701	66%	.535	1.6

Table 3 - Segment cleaning statistics and write costs for production file systems.
For each Sprite LFS file system the table lists the average disk capacity utilization, the total number of segments cleaned over a four-month period, the fraction of the segments that were empty when cleaned, the average utilization of the non-empty segments that were cleaned, and the overall write cost for the period of the measurements. These write cost figures imply that the cleaning overhead limits the long-term write performance to about 70% of the maximum sequential write bandwidth.

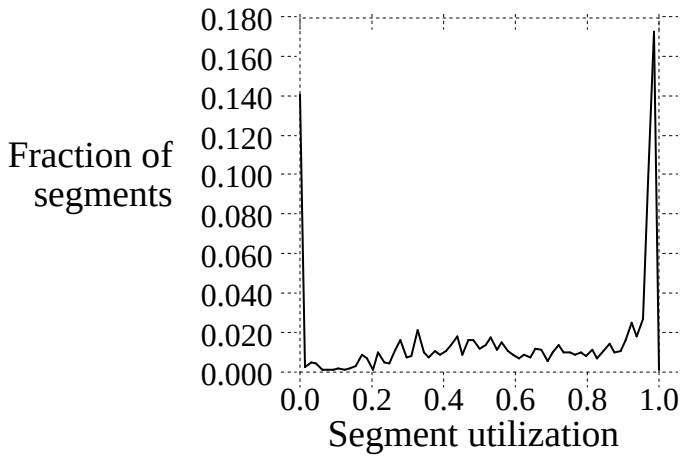

Figure 10 — Segment utilization in the /user5 file system
This figure shows the distribution of segment utilizations in a recent snapshot of the `/user5` disk. The distribution shows large numbers of fully utilized segments and totally empty segments.

We believe that there are two reasons why cleaning costs were lower in Sprite LFS than in the simulations. First, all the files in the simulations were just a single block long. In practice, there are a substantial number of longer files, and they tend to be written and deleted as a whole. This results in greater locality within individual segments. In the best case where a file is much longer than a segment, deleting the file will produce one or more totally empty segments. The second difference between simulation and

reality is that the simulated reference patterns were evenly distributed within the hot and cold file groups. In practice there are large numbers of files that are almost never written (cold segments in reality are much colder than the cold segments in the simulations). A log-structured file system will isolate the very cold files in segments and never clean them. In the simulations, every segment eventually received modifications and thus had to be cleaned.

If the measurements of Sprite LFS in Section 5.1 were a bit over-optimistic, the measurements in this section are, if anything, over-pessimistic. In practice it may be possible to perform much of the cleaning at night or during other idle periods, so that clean segments are available during bursts of activity. We do not yet have enough experience with Sprite LFS to know if this can be done. In addition, we expect the performance of Sprite LFS to improve as we gain experience and tune the algorithms. For example, we have not yet carefully analyzed the policy issue of how many segments to clean at a time, but we think it may impact the system's ability to segregate hot data from cold data.

5.3. Other overheads in Sprite LFS

Table 4 shows the relative importance of the various kinds of data written to disk, both in terms of how much of the live blocks they occupy on disk and in terms of how much of the data written to the log they represent. More than 99% of the live data on disk consists of file data blocks and indirect blocks. However, about 16% of the information written to the log consists of inodes, inode map blocks, and segment map blocks, all of which tend to be overwritten quickly. The inode map alone accounts for more than 10% of all the data written to the log. We suspect that this is because of the short checkpoint interval currently used in Sprite LFS, which forces metadata to disk more often than necessary. We expect the log bandwidth overhead for metadata to drop substantially when we install roll-forward recovery and increase the checkpoint interval.

Sprite LFS /user5 file system contents		
Block type	Live data	Log bandwidth
Data blocks*	98.0%	80.6%
Indirect blocks*	1.4%	2.4%
Inode blocks*	0.3%	3.5%
Inode map	0.2%	10.6%
Seg Usage map*	0.1%	2.1%
Summary blocks	0.0%	0.6%
Dir Op Log	0.0%	0.2%

Table 4 — Disk space and log bandwidth usage of /user5
For each block type, the table lists the percentage of the disk space in use on disk (Live data) and the percentage of the log bandwidth consumed writing this block type (Log bandwidth). The block types marked with '*' have equivalent data structures in Berkeley FFS.

6. Related work

The log-structured file system concept and the Sprite LFS design borrow ideas from many different areas of related work. File systems with log-like structures have appeared in several proposals for building file systems on write-once media[16, 17], but the write-once nature of the media made it unnecessary for the file systems to address the cleaning problem. The cleaning approach used in Sprite LFS is similar to some of the generational garbage collection schemes used in programming languages[18], but with the difference that efficient random access is possible in the generational garbage collectors, whereas sequential accesses are necessary to achieve high performance in a file system. Almost all database systems use write-ahead logging for crash recovery and high performance[12], but they do not use the log as the final repository for data, so different techniques may be used to recover log space. Collecting data in the file cache and writing it to disk in large writes is similar to the concept of group commit in database systems[19] and to techniques used in main-memory database systems[20].

7. Conclusion

The basic principle behind a log-structured file system is a simple one: collect large amounts of new data in a file cache in main memory, then write the data to disk in a single large I/O that can use all

of the disk's bandwidth. Implementing this idea is complicated a bit by the need to maintain large free areas on disk, but both our simulation analysis and our experience with Sprite LFS suggest that low cleaning overheads can be achieved with a simple policy based on cost and benefit. Although we developed a log-structured file system to support workloads with many small files, the approach also works very well for large-file accesses. In particular, there is essentially no cleaning overhead at all for very large files that are created and deleted in their entirety.

The bottom line is that a log-structured file system can use disks an order of magnitude more efficiently than existing file systems. This should make it possible to take advantage of several more generations of faster processors before I/O limitations once again threaten the scalability of computer systems.

8. Acknowledgments

Diane Greene, Mary Baker, John Hartman, Mike Kupfer, and Ken Shirriff provided helpful comments on drafts of this paper.

References

1. John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson, ''A Trace-Driven Analysis of the Unix 4.2 BSD File System,'' *Proceedings of the 10th Symposium on Operating System Principles*, pp. 15-24 ACM, (1985).
2. Michael L. Kazar, Bruce W. Leverett, Owen T. Anderson, Vasilis Apostolides, Beth A. Bottos, Sailesh Chutani, Craig F. Everhart, W. Anthony Manson, Shu-Tsui Tu, and Edward R. Zayas, ''DEcorum File System Architectural Overview,'' *Proceedings of the USENIX 1990 Summer Conference*, pp. 151-164 (Jun 1990).
3. Andy Hisgen, Andrew Birrell, Chuck Jerian, Timothy Mann, Michael Schroeder, and Garret Swart, ''Granularity and Semantic Level of Replication in the Echo Distributed File System,'' *IEEE Workshop on the Management of Replicated Data*, (Nov 1990).
4. Robert B. Hagmann, ''Reimplementing the Cedar File System Using Logging and Group Commit,'' *Proceedings of the 11th Symposium on Operating Systems Principles*, pp. 155-162 (Nov 1987).
5. John K. Ousterhout , Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch, ''The Sprite Network Operating System,'' *IEEE Computer* **21**(2) pp. 23-36 (1988).
6. David A. Patterson , Garth Gibson , and Randy H. Katz, ''A Case for Redundant Arrays of Inexpensive Disks (RAID),'' *ACM SIGMOD 88*, pp. 109-116 (Jun 1988).
7. M. Satyanarayanan, ''A Study of File Sizes and Functional Lifetimes,'' *Proceedings of the 8th Symposium on Operating System Principles*, pp. 96-108 ACM, (1981).
8. Edward D. Lazowska , John Zahorjan, David R Cheriton, and Willy Zwaenepoel, ''File Access Performance of Diskless Workstations,'' *Transactions on Computer Systems* **4**(3) pp. 238-268 (Aug

1986).

9. Marshall K. McKusick , ‘‘A Fast File System for Unix,’’ *Transactions on Computer Systems* **2**(3) pp. 181-197 ACM, (1984).
10. R. Sandberg , ‘‘Design and Implementation of the Sun Network Filesystem,’’ *Proceedings of the USENIX 1985 Summer Conference*, pp. 119-130 (Jun 1985).
11. John K. Ousterhout, ‘‘Why Aren’t Operating Systems Getting Faster As Fast as Hardware?,’’ *Proceedings of the USENIX 1990 Summer Conference*, pp. 247-256 (Jun 1990).
12. Jim Gray, ‘‘Notes on Data Base Operating Systems,’’ in *Operating Systems, An Advanced Course*, Springer-Verlag (1979).
13. A. Chang, M. F. Mergen, R. K. Rader, J. A. Roberts, and S. L. Porter, ‘‘Evolution of storage facilities in AIX Version 3 of RISC System/6000 processors,’’ *IBM Journal of Research and Development* **34**(1) pp. 105-109 (Jan 1990).
14. Marshall Kirk McKusick, Willian N. Joy, Samuel J. Leffler, and Robert S. Fabry, ‘‘Fsck - The UNIX File System Check Program,’’ *Unix System Manager’s Manual - 4.3 BSD Virtual VAX-11 Version*, USENIX, (Apr 1986). SMM:5
15. Larry McVoy and Steve Kleiman, ‘‘Extent-like Performance from a UNIX File System,’’ *Proceedings of the USENIX 1991 Winter Conference*, (Jan 1991).
16. D. Reed and Liba Svobodova, ‘‘SWALLOW: A Distributed Data Storage System for a Local Network,’’ *Local Networks for Computer Communications*, pp. 355-373 North-Holland, (1981).
17. Ross S. Finlayson and David R. Cheriton, ‘‘Log Files: An Extended File Service Exploiting Write-Once Storage,’’ *Proceedings of the Eleventh Symposium on Operating System Principles*, pp. 129-148 (Nov 1987).
18. D. Ungar, ‘‘Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm,’’ *Proceedings of the Software Engineering Symposium on Practical Software Development Environments*, pp. 157-167 (Apr 1984).
19. David J. DeWitt, Randy H. Katz, Frank Olken, L. D. Shapiro, Mike R. Stonebraker, and David Wood, ‘‘Implementation Techniques for Main Memory Database Systems,’’ *Proceedings of SIGMOD 1984*, pp. 1-8 (Jun 1984).
20. Kenneth Salem and Hector Garcia-Molina, ‘‘Crash Recovery Mechanisms for Main Storage Database Systems,’’ CS-TR-034-86, Princeton University, Princeton, NJ (1986).