The Design and Implementation of a Log-structured File System

Copyright © 1992

Mendel Rosenblum

The Design and Implementation of a Log-structured File System

by

Mendel Rosenblum

Abstract

This dissertation presents a new technique for disk storage management called a *log-structured file system*. The technique writes all file system changes in large sequential transfers to a log-like structure on disk. The key benefit is a high write performance that is independent of the workload. The large transfers also enable the efficient use of large disk arrays such as RAIDs. The technique minimizes the overhead of computing the redundancy information required by large RAIDs.

A log-structured file system achieves high write rates without sacrificing file retrieval performance. Files are read back from the log efficiently due to the indexing information that is maintained. The log structure also permits fast recovery from system crashes. Using a recovery system based on checkpoints and roll-forward the log-structured file system can quickly restore the disk to a consistent state.

An important focus of this dissertation is the technique used for free space management in a log-structured file system. The approach taken was to divide the disk into large *segments* to which the log was written. A *segment cleaner* mechanism exists to compress the live information from heavily fragmented segments. The mechanism reads in the fragmented segments, compacts the live data, and writes the data back to segments on disk. The dissertation includes a series of simulations that demonstrate the efficiency of a simple segment cleaning policy based on cost and benefit. The segment cleaner decides which segments to clean based on a function of the fraction alive in the segment and the age of the data in the segment.

I 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 and matches or exceeds 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. Unix file systems typically can use only 5-10%.

CHAPTER 1

Introduction

Computer systems research is heavily influenced by changes in computer technology. As technology changes alter the characteristics of the underlying hardware components of the system, the algorithms used to manage the system need to be re-examined and new techniques need to be developed. Technological influences are particularly evident in the design of storage management systems such as disk storage managers and file systems. The influences have been so pronounced that techniques developed as recently as ten years ago are being obsoleted.

The basic problem for disk storage managers is the unbalanced scaling of hardware component technologies. Disk storage manager design depends on the technology for processors, main memory, and magnetic disks. During the 1980s, processors and main memories benefited from the rapid improvements in semiconductor technology and improved by several orders of magnitude in performance and capacity. This improvement has not been matched by disk technology, which is bounded by the mechanics of rotating magnetic media. Magnetic disks of the 1980s have improved by a factor of 10 in capacity but only a factor of 2 in performance.

This unbalanced scaling of the hardware components challenges the disk storage manager to compensate for the slower disks and allow performance to scale with the processor and main memory technology. Unless the performance of file systems can be improved over that of the disks, I/O-bound applications will be unable to use the rapid improvements in processor speeds to improve performance for computer users. Disk storage managers must break this bottleneck and decouple application performance from the disk.

One technique for decoupling CPU and disk performance is *caching*. Caching decouples file system performance from that of the disk by using main memory to hold commonly referenced disk blocks. With caching, application read requests can proceed at memory speeds rather than disk speeds. Unfortunately, caching techniques can not solve all the problems. Caching will not catch all read requests and can do little to improve the performance of write requests. Write requests must go through to disk because the contents of main memory can be lost during system crashes. The challenge for storage manager design is to use the disk as efficiently as possible for those requests that are not satisfied by caching.

Performance of storage requests is not the only problem caused by the improvements in technology. The tremendous increase in disk capacity has broken some of the algorithms used by current storage managers. For example, one area where many current storage managers have failed to scale gracefully with disk capacity is the time required to reach a consistent file system state after a system crash. Restoration of consistency, called crash recovery, scales with the size of the disk storage. This works well for the small capacity disks of the 1970s but works poorly for the large disks of the 1990s. What is needed is a reexamination of the issues and a new design that can take advantage of and scale with the technology improvements of the 1990s.

1.1. Log-structured file systems

This dissertation describes a new disk disk storage management technique, called *log-structured file systems*, that is well suited for the technology and challenges facing the disk storage managers of the 1990s. Using a log-structured file system, storage write requests can be processed an order of magnitude more efficiently that current file systems; this allows the power of future processors to be fully utilized. The improvement in write performance is accomplished by writing all new information to disk in a sequential structure called the *log*. This approach increases write performance dramatically by almost eliminating disk seeks. By writing everything sequentially to disk, a log-structured file system uses the disk in the most efficient way possible. The sequential nature of the log also permits much faster crash recovery. Rather than scan the entire disk to restore consistency, a log-structured file system need only examine the most

recent portion of the log. Crash recovery time in a log-structured file system does not grow with the disk capacity.

Log-structured file systems are based on the assumption that files are cached in main memory and that increasing memory sizes will make caches more and more effective at satisfying read requests[1]. As a result, disk traffic will become dominated by writes. This condition is precisely the opposite of the read-dominated workloads that almost all previous disk storage managers were designed for. The changed workloads require a write-optimized rather than read-optimized disk access technique.

The notion of logging is not new; it has been used extensively in database systems and a number of recent file systems have incorporated a log as an auxiliary structure to speed up writes and crash recovery[2, 3]. 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 an efficiency comparable to that of current file systems. The result is that a log-structured file system can provide the read performance of current read-optimized storage managers while greatly exceeding their write performance.

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. Keeping large extents of space available is the most significant challenge in the design of a log-structured file system. This dissertation presents a solution based on large extents called *segments*. A *segment cleaner* process continually regenerates empty segments by compressing the live data from heavily fragmented segments. The segment cleaner acts as a garbage collector that insures that there is room to append more to the log. The file system can be viewed as a log that is continually wrapping upon itself.

The segment cleaner uses a simple but effective algorithm to segregate older, more slowly changing data from younger rapidly-changing data. The algorithm is based on cost and benefit and causes the older data to be treated differently from younger data during cleaning. Simulations show that this algorithm reduces the overhead of the segment cleaner substantially, allowing high performance for disks operating at the capacity utilization of current storage systems.

The concept of log-structured file systems is demonstrated by a prototype log-structured file system called Sprite LFS, which is in production use as part of the Sprite network operating system[4]. Benchmark programs demonstrate that, for small files, the raw writing speed of Sprite LFS is more than an order of magnitude greater than that of the commonly used Unix file system[5]. 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). Long-term measurements of Sprite LFS in production use show the inherent advantages of log-structured file systems. 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.

1.2. Focus of research: Unix office/engineering environment

The research presented in this thesis focuses on workloads in the Unix office/engineering environment. The Unix office/engineering environment is characterized by applications for document preparation, program development, simulation studies, computer aided design (CAD), and other miscellaneous programs running under the Unix operating system programming interface[6].

There are a number of advantages to using this environment as a focus for research. Unix applications make heavy use of the file system and the workloads they generate are well documented in the research literature. The storage requests of the Unix environment vary greatly in size and are dynamic in nature. This makes the Unix environment workloads particularly challenging for disk storage managers. It also puts the applications in jeopardy of becoming I/O bound on future faster processors.

The Unix environment also presents problems when mapping the disk request pattern onto disk array technology such as RAIDs[7]. RAID technology can dramatically increase the performance of the disk storage subsystem but the small disk request size of current Unix disk storage managers prevent efficient use of this technology. Some environments, such as those containing large sequential accesses, already take full advantage of RAIDs. The Unix office/engineering environment requires innovations in the disk storage manager in order to use RAIDs efficiently.

The final advantage of studying the Unix environment is that it is the environment that I and the other members of the computer science department use for day to day computing. This enabled the log-structured file system research to be used and evaluated ‘‘in-house’’.

1.3. Thesis Contributions

The main contributions of this thesis are:

- The design of a log-structured file system that achieves high write performance and fast crash recovery in a simple and efficient implementation. The design, Sprite LFS, is the only published complete design for a log-structured file system. The design fills the gaps left in the initial proposal for a log-structured file system[8] by using a segmented log structure to handle the log wrapping upon itself.
- The design demonstrates the log-structured file system as a high performance disk storage management technique that will scale well with the technology changes of the 1990s. In particular the log-structured file system enables the efficient use of RAID level 4 and level 5 disk array technologies on workloads containing small modifications. Experiments run on the Sprite LFS prototype as well as measurements taken from long-term use of the Sprite LFS provide indisputable proof of log-structured file systems as a concept and the Sprite LFS design as an implementation. The work described in this thesis provides the first published study documenting the behavior of a log-structured file system over varying workloads and long-term usage.
- Several different policies for controlling the segment cleaning mechanism of Sprite LFS are analyzed. Included in this analysis is the discovery of a cost-benefit segment selection policy that substantially reduces the segment cleaning overhead for Sprite LFS. Using this policy Sprite LFS can exploit the locality of reference in the workload to increase performance and disk space efficiency.

1.4. Relationship to Sprite and RAID

Although the research described in this dissertation was done in the context of the Sprite network operating system and RAID disk array projects, its results are not limited to Sprite, network operating systems, or disk arrays. In particular, this dissertation presents measurements of Sprite LFS running on non-arrayed disks used for a variety of applications including program development, simulations, hardware design, and virtual memory backing store. Sprite provided a convenient platform for a first implementation of a log-structured file system but the results are more broadly applicable.

Although not limited to Sprite and RAID, the research presented in this dissertation draws motivation from both projects. The large file caches of the Sprite distributed file system motivated the log-structured file system design. In addition, the log-structured design is the key to enabling the RAID technology for workloads with small updates. The large write access pattern of Sprite LFS allows workloads containing small modifications to efficiently use RAID style disk arrays.

1.5. Outline of Dissertation

The rest of this document is organized into eight chapters. Chapter 2 discusses the design techniques used in current storage managers . Chapter 3 reviews the issues in designing file systems for computers of the 1990s. It provides both the motivation and direction for this research. For each technique, the effect of current technology trends on the applicability of the technique is discussed. Chapter 4 presents the design alternatives for a log-structured file system and derives the structure of Sprite LFS. The mechanisms used to write data, locate files, clean segments, and recover from crashes are presented. Chapter 5 describes a simulation study of the Sprite LFS segment cleaning policy. The simulation results are used to develop and justify the cost-benefit segment selection policy implemented in Sprite LFS. Chapter 6 evaluates the behavior of the Sprite LFS prototype under benchmark programs and presents measurements of long-term usage by the Sprite user community. Chapter 7 presents the related work in the research literature and compares Sprite LFS to other contemporary file systems and storage managers. Chapter 8 contains concluding remarks and Chapter 9 is the bibliography.

CHAPTER 2

Disk Storage Manager Design

Disk storage manager design is a relatively old research area of computer science and many designs have been both proposed and implemented. The basic abstractions and implementations in use today date back to the 1960s[9]. This section describes these abstractions and implementation techniques with particular focus on the disk storage manager designs for the Unix office/engineering environment. A brief outline of the Unix file system implementation is presented to provide a reference point for the discussion of disk storage manager optimization techniques.

This chapter focuses on storage manager techniques that effect the performance and crash recovery time. The performance enhancing techniques presented can be divided into two classes: techniques for avoiding disk accesses and techniques for accessing disks more efficiently. The speed of a disk can be hidden from application programs by using the fast storage of main memory to satisfy storage requests. Section 2.4.1 describes techniques of read and write caching and prefetching that allow main memory to be used to increase performance.

The efficiency of disk accesses can be improved both by techniques that carefully allocate the storage on disk to decrease disk overhead time and by scheduling the disk requests. Section 2.4.2 describes techniques for controlling disk allocation such as continuous allocation and clustering of data. Section 2.4.3 describes sorting techniques that may be used for scheduling disk requests.

The last part of this chapter addresses techniques for performing crash recovery on file systems. Several techniques are presented in Section 2.5 including the Unix scavenger program that restores consistency after a crash and techniques that use a log to speed recovery. Logging techniques described in detail in Section 2.5.3 are of particular interest because they can improve both performance and crash recovery time.

The techniques presented in this chapter are dependent on three hardware technologies: processors, main memory, and magnetic disks. This chapter describes the current state of each technology and uses this description to motivate the techniques presented. Chapter 3 examines the current evolutionary trends for these technologies and the effects the trends are having on the techniques presented in this chapter. These effects provide the motivation for the design of the log-structured file system presented in Chapter 4.

2.1. Unix file system abstraction

The Unix file system interface[6] is a commonly used one in the office/engineering environment. It defines a set of abstractions and operations that must be supported by the disk storage manager. The choice of the Unix file system interface has serious implications for a disk storage manager. One of the most important implications is that applications provide little if any information about their access patterns to data stored in the file system. For example, unlike file systems such as VMS[10] and MVS, the Unix file system interface provides no information about the eventual file size, file lifetime, or access pattern that the file will have. Storage managers for this environment must operate efficiently with no ‘‘hints’’ from the application.

The basic data storage abstraction in Unix is the *file*. A Unix file is represented as a sequence of bytes. The bytes are uninterpreted by the file system; no additional record or block structure is imposed. Files also have attributes associated with them that store information about the file’s owner, access permissions, access and modify times, and size. Each file is uniquely identified by an integer called its *i-number*. The Unix file system supports a special file type called a *directory* that is writable only by the file system. Directories contain pairs of names and i-numbers and are used to assign names to files. Application programs reference files using these names and the file system translates them into i-numbers to locate the file. The entries in a directory may refer to other directories as well as files, which leads to a tree-structured

naming hierarchy. For example, the name ‘‘/a/b/c’’ refers to a file ‘‘c’’ contained in a directory ‘‘b’’ which is contained in another directory ‘‘a’’.

Unix file operations follow the traditional model of opening a file, performing some I/O transfers, and closing the file. Files are opened with the *open* system call which takes as an argument the name of the file to open. If the file being opened does not exist the *open* call can create the file with an initial length of zero bytes. The open call also allows the specification of the protection bits for the newly created file. Note that other than the protection bits no file attributes need or can be specified. In particular the file system is not given any information about the amount of space needed for the file.

For each open file a *current offset* into the file is maintained by the file system. The *write* system call allows an application to transfer an arbitrary amount of data to a file starting at the current offset. The *read* system call allows an arbitrary amount of data to be read into a memory buffer from the file starting at the current offset. The current offset is updated by both the *read* and *write* systems call. The application can change the current offset without transferring data using the *lseek* system call. By explicitly setting the current offset before each transfer operation an application can perform non-sequential accesses to a file.

Directories in Unix are created with a special system call, *mkdir*, that allocates and initializes a new directory. For each process, the Unix file systems keeps track of a *current working directory*. The current working directory is used to resolve relative pathname lookups (i.e. opens that do not specify that the search should start at the root of the naming hierarchy tree.)

The Unix interface also includes operations for retrieving a file’s attributes (*stat*) system call, and for truncating files to a specified length (*truncate*). A file is deleted from disk when the last directory entry that references the file is removed with the *unlink* system call.

2.2. Unix file system metrics

Before storage management techniques can be evaluated it is necessary to choose the evaluation metrics. Commonly used evaluation metrics for Unix file systems are performance, storage efficiency, and reliability.

Performance

Unix file system performance can be classified along three different axes: latency, bandwidth, and throughput. Latency of a request is the delay that is independent of the amount of data transferred. Since many system calls in Unix are synchronous, the latency is a fixed cost that add directly to the process’ execution time. The cost is particularly important for workloads containing requests that transfer small amounts of data. For such workloads the latency dominates the total request time. An example measure of latency is how long it takes to create or delete a file.

Bandwidth is a measure of the performance achievable by a single application. The bandwidth metric measures how fast a single application can source or sink from the file system. Bandwidth is an important metric for applications that transfer large amounts of data. An example measure of bandwidth is the average transfer rate that a 100 megabyte file can be read or written by an application.

The final performance metric of interest is the throughput or operations per second the file system can perform. For Unix systems with many processes making storage requests simultaneously, the throughput of the system is an important metric. An example of a measure of throughput is number of simultaneous I/O requests the system can process at a time.

Storage efficiency

Since Unix is used in cost-sensitive environments efficient use of the available disk storage has been an important design consideration for disk storage managers. The *storage efficiency* of a disk storage manager can be expressed as the fraction of the disk storage that is available for application data. The goal is to have a storage efficiency fraction of 1.0 but this is not possible because some disk space will be needed for metadata (data maintained by the storage manager to locate application data, such as Unix inodes, directories, and indirect blocks).

Besides space lost to metadata, disk space is lost to *internal fragmentation* and *external fragmentation*. Internal fragmentation occurs when the unit of allocation of the storage manager is not a multiple of the size of the object being stored. For example, Unix file sizes are in units of bytes while the storage is

typically allocated in blocks. This means the last block of a file may be only partially used. The extra unused space is referred to as internal fragmentation.

External fragmentation refers to small free areas on disk that are hard to use because of their small size. External fragmentation occurs in systems that have different size allocation units so that free space may exist between objects that is too small to hold the current allocation request. For example in a system that allocates whole files contiguously on disk, the space between two files that is too small to allocate a new file is external fragmentation.

A third source of wasted disk space comes from space purposefully left unallocated by the disk storage manager to increase the performance of the storage manager. For example, by intentionally leaving free a certain amount of disk space the storage manager can ease the task of finding free blocks. This space could be allocated to files but is left free for performance reasons.

Although disk space cost has been dropping dramatically for the last part of the 1980s and early 1990s, disk storage managers still must avoid wasting space to be competitive with existing designs. The initial Unix file system used small fixed sized blocks to limit internal fragmentation and eliminate external fragmentation. Existing designs such as the Berkeley Unix fast file system reserve 10% of the disk space to increase allocation performance.

Reliability/Crash Recovery

Applications in the Unix environment have evolved to tolerate the mediocre reliability of the initial Unix file system implementation. The system is assumed to function correctly unless there is an abnormal event such as a system crash. Traditionally, recently-written data may be lost during a crash. The modifications are guaranteed to survive crashes if they have lived longer than a specified timeout that is typically set at 30 seconds. Later versions of Unix have extended the interface to provide system calls that force modifications to disk. Many applications have been modified to use these calls to decrease the chances of losing data in a crash. Disk storage manager designs should match if not exceed this level of reliability.

Beside the loss of data, crashes may leave the Unix disk structures in an inconsistent state that requires repair before they can be accessed again. After a crash a special crash recovery program *fsck[11]* must be run on the file system before it can be accessed. As described in Section 2.5.2, fsck is time consuming and can greatly increase the restart time of the system after a crash.

Tradeoffs

One problem with using the three metrics of performance, storage efficiency, and reliability to evaluate storage manager designs is that they are not independent of each other. It is both possible and desirable to have designs that exploit the tradeoffs between the metrics. When evaluating storage manager design techniques it is important to be aware of the tradeoffs being made and how the tradeoffs affect application programs. For example, storage managers designed for environments in which performance is of the utmost importance might reduce the storage efficiency in order to increase the performance. The Berkeley Unix fast file system makes this tradeoff by reserving 10% of the disk space to ensure high performance allocation.

It is also possible to make tradeoffs between reliability and performance. Unix file systems have traditionally tolerated some loss of data at a crash in order to increase performance. In other environments such as a bank's database system reducing reliability to increase performance would be unacceptable.

2.3. Unix file system implementation

The initial Unix file system implementation[6] used very simple disk storage management techniques and contained few of the optimizations presented in this chapter. Nevertheless it is interesting to study because it forms the base for many of the existing implementations of the Unix file system interface. This section presents disk storage management techniques, using as examples the improvements made to the Unix file system implementation by systems such as the Berkeley Unix fast file system. The section starts with a description of the initial Unix implementation.

The initial Unix file system divided the disk into three main regions: the *super-block*, the *i-list*, and the *data block region*. The super-block region contained the description of the file system including the

locations of the other regions and other file system parameters such as the number of files. The i-list was an array of *inode* data structures. The array was indexed by the file's i-number and there was one inode per file or directory in the file system. The inode contained the file's attributes, such as owner and size, as well as some pointers to the data blocks of the file. The data blocks themselves resided in the data region of the disk. The data region can be viewed as an array of 512-byte blocks from which files and directories were allocated.

The blocks of a file was located using a tree-structured block index rooted at the file's inode. Figure 2-1 shows the format of the tree. The inode contained 13 disk addresses that pointed at blocks in the data block region. The first 10 addresses pointed at the first 10 blocks of the file. For files that required more than 10 blocks, the 11th disk address in the inode pointed to a block of disk addresses pointing to 128 additional data blocks. This block of pointers is known as an *indirect block*. Even larger files require using the 12th disk address as a pointer that points to a block of 128 pointers to indirect blocks. This additional indirection block is called a *doubly indirect block*. Still larger files use the 13th disk address in the inode as a pointer to a *triple indirect block* that contains 128 pointers to *doubly indirect blocks*. Using this tree-structured block index, the Unix file system can quickly access any block in a file even for very large files. Figure 2-2 shows the on-disk format of a small Unix file system.

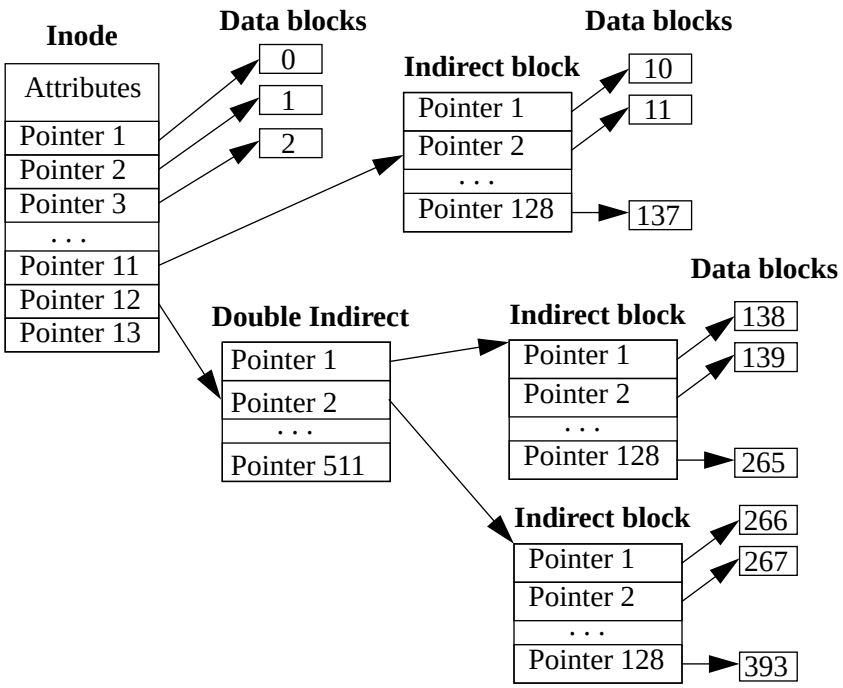

Figure 2-1 — File block index format of the Unix file system
This figure shows the tree-structured format of the index used to locate the blocks of file in the initial Unix file system. The file's inode contains pointers to the first 10 data blocks. The eleventh pointer in the inode points at an indirect block that points at the next 128 data blocks. The twelfth pointer points at double indirect block that points at 128 indirect blocks. The thirteenth pointer (not shown in figure) points at a triple indirect block that points at 128 double indirect blocks.

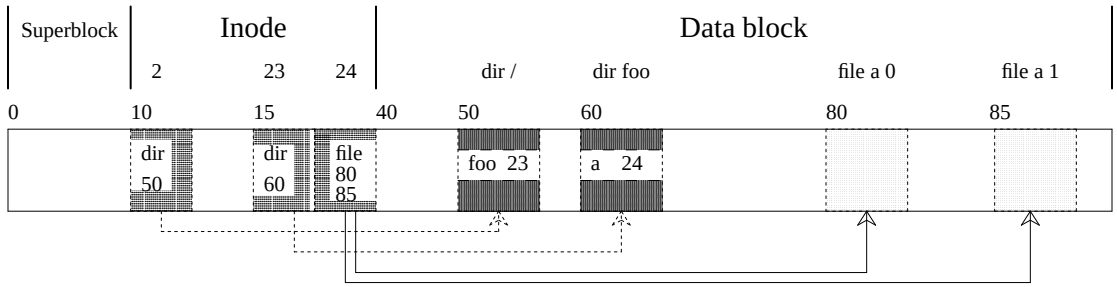

Figure 2-2 — Disk layout of a Unix file system
This figure shows the disk layout of the a small Unix file system containing the directories "`/`" and "`/a`" and the file "`/a/foo`". Each directory and file is given an inode under its i-number in the inode region. The directory "`/`" is created under i-number 2 when the file system is created. `/a` was given i-number 7 while the file `foo` was given i-number 23. The inodes contain the attributes and pointers to the blocks of the file or directory.

Blocks in the data region that are not currently allocated to a file or directory were kept on a linked list called the *freelist*. The freelist starts with a pointer in the superblock that points at the first free block in the data region. Each block on the freelist contains the disk address of 50 unallocated blocks. Having many disk pointers per block reduces the number of block fetches needed to do allocation. Blocks are allocated when the *write* or *mkdir* system call needs space from the data region on disk. Blocks are deallocated and returned to the freelist when files are truncated or deleted.

To improve performance, the Unix file system reserved a portion of main memory for use as a cache of recently accessed disk blocks. Disk read requests that were found in the cache avoided the overhead of disk I/O. Disk write requests modified the block in the cache. The actual on-disk copy of the block was not updated until later. Every 30 seconds a process performs a *sync* system call that writes all the modified blocks in the main memory cache to their home on disk.

2.3.1. Berkeley Unix Fast File System

In the early 1980s a major revision of the Unix file system was done at Berkeley[5]. The resulting file system, called the Berkeley Fast File System (Unix FFS), has become one of the most widely used and studied Unix file systems. Unix FFS changed the on-disk layout in a number of ways that improved performance. Changes included increasing the block size, changing the disk region layouts, and a new block allocation algorithm. This section presents the changes made in Unix FFS while the next section will review the storage management techniques that motivated the changes.

Unix FFS removes the limit that blocks were only 512 bytes in size. Blocks in the Unix FFS file system can be any power of two starting with a minimum size of 4096 bytes up to the limits imposed by the disk controller and drives. Unix FFS also adds the notion of a block *fragment* that only occupies a portion of a file system block. To reduce internal fragmentation the last block in a file is allowed to be one or more fragments rather than an entire file system block. The fragment size is set to be one fourth or one eighth the size of a block. Although having fragments complicates the design, fragments are necessary to avoid large amounts of disk space being lost to internal fragmentation. Studies showed that with a 4096 byte block as much as 50% disk space would be lost on normal workloads[5].

Unix FFS keeps the basic on-disk format of inodes and indirect blocks that was present in the initial Unix implementation. However, the inodes are no longer clustered together in their own region. Instead, Unix FFS breaks the disk into regions of adjacent cylinders called *cylinder groups*. Each cylinder group has its own inode region and data block region.

Unix FFS also has a new block allocation algorithm and data structures. The freelist data structure is replaced by a *bitmap* data structure that records the allocated state for each block and fragment on disk.

The allocation algorithm was changed to attempt to allocate blocks of a single file together and allocate files in the same directory in the same cylinder group.

Even with these changes, the disk layout of the Unix FFS is still very similar to the disk layout of the initial Unix file system pictured in Figure 2-2. Rather than representing the entire disk, Figure 2-2 can be viewed as a single cylinder group of a Unix FFS disk. The inode and directory structure are still the same. The differences are that blocks are larger (4096 or 8192 bytes in Unix FFS versus 512 in Unix), the distance between the inodes and the data blocks is smaller (a maximum of a few cylinders in Unix FFS rather than the entire disk in Unix), and the blocks of a file are more likely to be close together in Unix FFS.

2.4. Design techniques for high performance

Over the last 20 years a number of techniques have been developed to improve the performance of disk storage managers. This section discusses several of the techniques, focusing on those that benefit the Unix office/engineering environment. The first techniques presented involve the use of main memory to hide from the application the much slower magnetic disks. Techniques presented include caching and prefetching of data.

The second form of performance improvement techniques presented focuses on reducing the disk delays by accessing the disk with more efficient access patterns. Techniques of this form include optimizing the layout of data on disk to promote locality of storage that is likely to be requested together. Contiguous allocation and clustering are examples of such disk layout techniques. Disk efficiency can also be improved by ordering the requests sent to disk using the disk scheduling techniques presented in Section 2.4.3.

2.4.1. Main memory techniques

The goal of high performance disk management techniques is to minimize the time applications spend waiting for disk accesses to complete. One common technique for doing this is to use main memory to satisfy storage requests rather than forcing the request to go to the slower disk storage. The attraction of using main memory to service storage requests comes from the dramatic difference in access time between main memory and disk technology. For example, using technology common on machines in the 1980s, the access time of a 512-byte block of storage in main memory is measured in hundreds of microseconds while disk access times are in the tens of milliseconds. This two order-of-magnitude difference in access times is known as the *access gap*.

Three techniques are used by disk storage managers to lessen the performance impact of the access gap: *read caching*, *prefetching*, and *write caching*. The idea behind these techniques is to have enough of the storage requests satisfied from main memory that the storage system operates closer to memory speeds than disk speeds.

Read caching

The technique of read caching is used by the storage manager to keep frequently accessed storage in main memory. Application read requests can be satisfied by accessing a copy stored in memory with zero time spent waiting for the disk. The Unix file system makes extensive use of read caching. All disk resident items such as files, directories, and metadata blocks can be cached in the disk cache. The Unix disk cache is maintained using a least recently used algorithm that exploits locality and reuse in file system access patterns. Much research literature exists on read caching[1, 12-14]. Studies have found cache hits rates of 80-90%. This rate is high enough to significantly reduce the disk delays experienced by Unix applications.

From the storage manager's view there is no disadvantage to read caching. The more memory devoted to caching, the higher the cache hit rate and the fewer disk accesses that are needed. From the whole system perspective there are disadvantages to read caching. Memory used to cache files can not be used for supporting the application's other memory requirements. Care must be taken not to give so much memory to the storage manager that applications can not perform acceptably. Nelson's PhD thesis[13] contains a study and suggested solution of trading off space between virtual memory and the storage manager's cache.

Prefetching

Prefetching is a technique that, like read caching, can improve the performance of read requests by using main memory. Prefetching works by bringing into memory the data that the application might reference in the future. Should the application actually request the prefetched data the application will not have to wait for the requests to be processed from disk. By predicting the reference and fetching the data ahead of time, the storage manager can reduce the time an application spends waiting for the disks.

The Unix file system monitors the access to a file; whenever it sees sequential accesses to adjacent blocks of a file it assumes a sequential access mode and begins prefetching. Unix also implicitly prefetches data that is smaller than a block in size. For example, rather than fetching a single inode from disk the entire block of inodes is fetched into memory.

Prefetching works well when the access patterns can be consistency predicted and works poorly on workloads that can not be predicted. Prefetching in Unix works well because sequential access is the most common access pattern[15].

Write caching

Just as read caching uses memory to improve the performance of reads, it is possible to use memory to speed writes using a technique called *write caching*. Write caching, also known as write buffering, write behind, or delayed writes, has the storage manager return control to an application before the data of a write request is transferred to disk. Main memory is used to buffer the write requests before sending them to disk. Write caching improves performance in several ways including reducing the time spent by an application waiting for disk I/O's to complete, reducing the amount of data written to disk, and allowing the disk transfers to proceed more efficiently. The first two benefits of write caching are presented in this section and the third benefit is presented in Section 2.4.3, which is on disk scheduling techniques.

The original Unix and Unix FFS file systems make extensive use of write caching. Application write system calls return after the data has been transferred into the disk cache and before the data was been transferred to disk. As long as there is space in the disk cache application write requests execute at CPU and memory speeds rather than disk speeds.

A second benefit of write caching occurs if multiple application writes are made to the same block. By delaying the transfer of the modified block, multiple application writes to the same block can be combined together and transferred to disk in a single I/O. Write caching is particularly helpful when applications write data sequentially in units smaller than a disk block. The multiple small transfers to fill in an entire block can be combined in memory and transferred to disk with a single I/O.

Write caching suffers from data integrity problems that make it unacceptable in some environments. Because the memory contents of most computer systems are lost when the system fails, write caching opens a window of vulnerability between when an application's write requests are performed and when they are safely on disk. Should the system crash during this window an application could ''lose'' modifications it made. For this reason, applications that depend on storage writes being permanent cannot use write caching. Prominent examples of such applications are data base management systems[16]. Data base systems need a guarantee that once the modification request returns, the changes will not be lost.

Even in an environment such as Unix office/engineering where some data loss in crashes is been tolerated, limits on write caching are needed to reduce the chances of lost data. Unix limits write caching by disabling write caching for some modifications and by limiting the time that a write may be cached. The *sync* system call that occurs on 30 second intervals in Unix provides one such limit on write caching. Modifications are held no longer than 30 seconds before the *sync* forces all changes in the disk cache to disk. Because of this, any changes that are older than 30 seconds will not be lost in a crash. This provision allows the file system to get some of the benefits of write caching without arbitrarily long exposures to data loss.

Unix FFS systems further limit write caching to ensure that data can be correctly recovered after a crash. Modifications to file system data structures such as directory blocks and inodes are not write cached. These operations use a special write routine internal to the Unix file system that modifies the block cache, starts the disk write, and waits for the disk write to complete. Write operations of this type are called *synchronous writes*. Section 2.5.1 describes the synchronous writes of the Unix FFS in more detail.

To provide additional guarantees to applications, the Unix FFS implementation added a new system call, *fsync*, that allows an application to force all modifications to a file from the disk cache to disk. Once an fsync call returns, an application knows that all modifications to the file have made it safely to disk. Applications such as text editors have been modified to use the fsync call to avoid loss of data during crashes.

2.4.2. Disk optimization techniques

Once the main memory of the computer has been exploited to improve performance, the next area of focus is to minimize the time an application spends waiting for the requests that have go to disk. Disk request times can be reduced by reducing overheads such as time spent moving the disk head (disk seeks) and time spent waiting for the storage to rotate underneath the disk head (rotational delay). This section describes the general approach used by storage managers to reduce overheads in disk requests. The first approach is to control the layout of storage on disk so that requests can be processed with the least overhead. The second approach is to schedule the application's disk requests to reduce the overheads.

2.4.2.1. Magnetic disk characteristics

Before discussing disk storage layout and disk scheduling techniques it is necessary to understand the characteristics of magnetic disks that influence their performance. Magnetic disks vary in performance characteristics and organizations. In this section a disk popular during the 1980s, the Fujitsu Eagle, is described as an example of a ''typical'' disk organization. Figure 2-3 shows the organization a magnetic disk. More advanced disk organizations are described in Section 3.1.2.

The Eagle is organized as a stack of rotating *platters* that contain data storage which is broken into 512-byte *sectors*. The platters rotate at 3600 RPM. Data is accessed by a *disk arm* that contains 20 *read-write heads*. Each disk head is held by the disk arm over one side of a platter so that it can access at any given time a circular ring of 67 sectors on the platter. This circular ring is called a *track* and all the tracks that are under all the disk heads for one disk arm position are called a *cylinder*. The disk contains 840

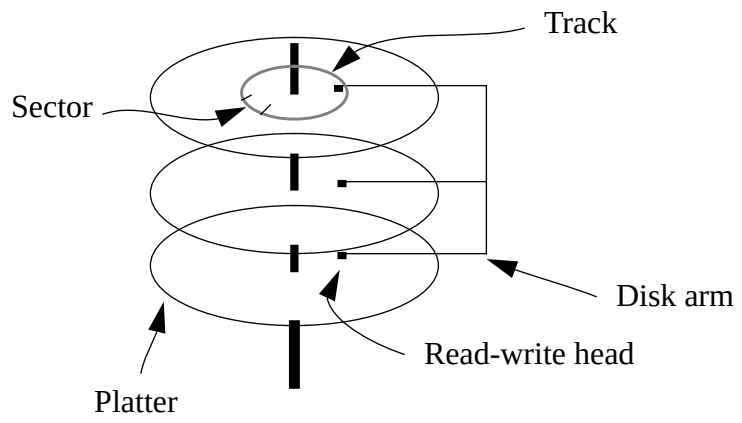

Figure 2-3 — Magnetic disk organization
This figure shows the organization of a ''typical'' magnetic disk.

cylinders, for a total storage capacity of about 540 megabytes.

A disk transfer request specifies a cylinder, head, starting sector, and the number of sectors to transfer. To process a request the disk first positions the arm to the correct cylinder using an operation called a *seek*. The time it takes to move the disk arm to the correct cylinder is called the *seek time*. For the Eagle, seek times take from 4.6 milliseconds for a one-cylinder seek to a maximum of 35 milliseconds for the longest seeks. The average seek time is 18 milliseconds. Once the seek positions the disk heads over the correct cylinder, the disk selects the correct disk head and waits until the starting sector rotates under the disk head before starting the transfer of the data. The time spent waiting for the starting sector to rotate under the disk head is called the *rotational latency*. Since the disk takes 16.6 milliseconds for a full rotation, random disk transfers must wait half a revolution on average or 8.3 milliseconds. The time spent reading the data from the sectors is called the *transfer time*. With data passing under the disk head at 1.96 megabytes per second, the transfer time for a 512-byte sector is around .25 milliseconds.

For the Eagle disk, the best case for accessing a 512-byte sector is when the disk head is positioned over the correct cylinder and the disk is rotationally positioned in front of the starting sector. In this case the transfer can complete in as little as .25 milliseconds. In the worse case, with a full 35-millisecond seek followed by a full rotation of 16.66 milliseconds, the transfer would take almost 52 milliseconds. An average access takes about 27 milliseconds to complete.

2.4.2.2. Contiguous allocation

The two order-of-magnitude difference between the best case and average case disk access times provides motivation for disk storage managers to allocate storage so that common access patterns achieve best-case access times. One such technique is *contiguous allocation*. Contiguous allocation works by allocating objects (i.e files) to consecutive sectors on disk so that sequential accesses can be performed with minimal delay between blocks. When objects are read sequentially, the request for the first block suffers a delay but all blocks after that proceed at the best-case rate.

Another advantage of contiguous allocation is that the size of an index used to find the blocks of an object is small. The location of an entire object can be described by a single disk pointer to the start sector of the object. The index is much smaller than the Unix technique that has at least one pointer per block. Smaller indexes mean that less data needs to be stored and examined to locate blocks of an object.

Contiguous allocation works best for large multi-block objects that are accessed sequentially. The delay suffered by the first block can be amortized over the rest of the blocks that are transferred. Since the blocks after the first are transferred with minimal delay, the average delay when transferring many blocks is low. Large objects can achieve transfer rates that are close to the maximum possible from the hardware.

Contiguous allocation is less beneficial for workloads with mostly small objects or small non-sequential transfers to large objects. With these workloads the transfers are dominated by the overhead to move to the start of the transfers. Because of this, contiguous allocation does little to increase disk efficiency for workloads with a high concentration of small objects such as the Unix office/engineering environment.

A more serious problem with contiguous allocation is external fragmentation. In an environment such as the Unix office/engineering environment with files of widely different sizes and much creation and deletion of files, contiguous allocation can result in much space being wasted to external fragmentation. Few general purpose storage managers require objects to be contiguously allocated and those that do either require frequent defragmentation programs to be run or else they restrict the operations that can be performed.

One such system that does contiguous allocation is the Bullet[17] file system. Contiguous allocation gives Bullet high performance on sequential accesses but the cost is high. Bullet requires that the file system be ‘‘repacked’’ at night to reclaim space lost to fragmentation. In addition, Bullet has restrictive access semantics that prohibit updating or file growth. Files in Bullet are immutable: all modifications create new copies of a file.

Extents

To reduce the external fragmentation problems of contiguous allocation some file systems allocate files in a small number of contiguous pieces. These pieces, commonly known as *extents*, share many of the

benefits of contiguous allocation and reduce the external fragmentation problems. If the extents are large enough, they act like contiguous allocation because the disk delay for the first block of an extent can be amortized over the rest of the blocks in the extent to keep the disk overhead low.

Several extant-based systems have been built and used including VMS[10], DTSS[18], MVS, and others[19]. These system allocate large files in a small number (e.g. 16) of large extents. When the file grows and fills one extent, another extent is added to the file.

Although extent-based storage managers ease the storage allocation problems of contiguous allocation they do not totally eliminate the problem. Unless all files are a multiple of the extent size, much space can be lost to the internal fragmentation of the files' last extents. Large extent sizes allow better sequential performance but cause worse internal fragmentation. For example, in a system with random sized files and an extent size of one megabyte, an average of half an extent or 512 kilobytes will be lost per file.

To reduce fragmentation, many extent-based systems require the application program to explicitly set the file's extent size. This is not compatible with the Unix file system interface. The Unix office/engineering environment and the Unix file system interface make extents particularly difficult because at creation time there is no information about the ultimate size of a file and most files are small. There is no interface to inform the file system of the file's ultimate size or a good extent size for the file.

Block index

The block index structure of Unix maintains a pointer for each block of a file. This structure does not preclude the implementation of contiguous allocation for files. It is certainly possible to allocate the blocks of a file contiguously on disk to achieve the same disk layout as an extent or contiguous allocation file system. Although this is possible, the use of the freelist data structure and the allocation algorithm of the original Unix file system made it unlikely that contiguous allocation could be achieved.

In the original Unix file system, file blocks were allocated from the freelist. When a file system was first created, the freelist contained most of the blocks in the data region and was initially sorted by increasing block number. Since blocks were allocated to files in the order that they appeared on the freelist, files were initially allocated contiguously. Unfortunately under workloads with frequent allocations and deallocations, the free space became fragmented and the freelist no longer was sorted by increasing disk address. Instead, the freelist degenerated into a list of random blocks that were free. Once the file system had been running for awhile, the blocks allocated for large files were spread randomly over the disk. The result was poor sequential access performance.

The Unix FFS file allocation algorithm improved on the original Unix file system by using a bitmap data structure for block allocation. Using a bitmap allows the algorithm to locate contiguous regions of free blocks. The allocation policy assigned blocks to files based on the location of the previous block in the file. Unless conflicting allocations occurred, the Unix FFS file system tends to allocate files contiguously or nearly contiguously. These modifications allow the file system to approximate the sequential access performance of contiguous allocation without resorting to large extents which would be wasteful for small files[20].

The Unix FFS allocation algorithm works well until the disk becomes nearly full. The reason for this is that as the disk fills it becomes harder to do contiguous or near-contiguous allocation. With few free blocks on the disk the algorithm can rarely do good allocation so it switches to a mode in which any available block is used. To limit the amount of performance degradation due to the block allocation mechanism, Unix FFS imposed a limit on how much of the disk space can be allocated. The file system always keeps 10% of the disk space free to prevent the block allocation algorithm from degenerating to random allocation.

The Unix FFS's technique of reserving disk space to keep acceptable performance indicates a general problem faced by all storage managers. Storage mangers can not operate efficiently unless there is some free space on disk. It is not possible to have high performance allocation and use every byte on the disk. The more free space, the easier it will be to find large contiguous regions. The question becomes ''how much of the disk can a file system use and still obtain high performance?''

2.4.2.3. Clustering

Contiguous allocation fails to help performance with small files because the files are too small to amortize the access to the data. In order to use the disk more efficiently with small files the storage manager must exploit locality in the access patterns between the files. This can be done using a technique called *clustering*. Using clustering the storage manager places files together (clusters) that are likely to be requested together. If the applications request files that are clustered together, the overheads of accessing the files on disk are smaller.

Storage managers typically implement clustering based on some static attribute of the data with the hope that the attribute will be a predictor of locality in the access patterns. For example, database systems cluster records based on the value in a particular field of a record. Queries that have locality based on this field will suffer less disk delays. File systems can cluster the files in a single directory with the hopes that there is locality within the references to a directory.

The Unix FFS clusters files based on the directory in which the file is created. This is implemented by allocating the file's inode and data block in the same cylinder group as the parent directory. By allocating to the same cylinder group, files created in the same directory will be within a short seek distance of each other. All the inodes and data blocks of the files in a directory can be accessed with no seek delay longer than the size of a cylinder group.

To cluster based on some attribute, objects that differ on that attribute must be spread apart. This means that clustering also implies spreading some data items a part. For example, in order for Unix FFS to cluster files in the same directory, different directories must be allocated to different cylinder groups.

Clustering works best when access patterns are predictable and consistent. If access patterns are constantly changing then clustering of data will not help performance. If the access patterns can not be predicted then the correct clustering can not be known. Unfortunately many interesting environments have unpredictable access patterns. The office/engineering environment has access patterns that are generated by many simultaneously running applications. The access streams of the applications are likely to be interleaved in ways such that large increases in disk efficiency using clustering will not be possible.

2.4.3. Disk Scheduling

Allocation techniques such as clustering and contiguous allocation are not the only techniques that disk storage managers can use to control disk access patterns. If more than one disk request is outstanding at a given time then the storage manager can process the requests in an order that reduces disk overheads. For example if the disk is presented with 10 requests that alternate between disk track 0 and disk track 100 the storage manager can reorder the requests so that all the requests are processed on track 0 before moving to track 100. This would result in only two disk seek delays, one to track 0 and one to track 100, rather than ten seek delays that would occur if the requests were processed in the order given. This scheduling of the disk, called *disk scheduling*, can reduce the seek delays, number of seeks, and rotational delays.

Disk scheduling algorithms have been carefully studied and many different algorithms can be found in the literature. They use techniques such as sorting the requests by shortest seek time first or scanning of the disk processing the requests as they are encounted. Recent studies[21, 22] that use modern disk characteristics found that sorting large numbers of random requests can increase the disk efficiency from around 7% to 25% of the disk's maximum bandwidth. Achieving 25% of the maximum disk bandwidth required a thousand requests to be queued and running of a complex, CPU intensive sort function that considered both seek delays and rotational latency.

Disk scheduling works best in environments where many requests are queued for a single disk. Because of the small number of threads of control in the office/engineering environment, large numbers of queued read requests are unlikely. Applications typically submit read requests and block until the requests return. The number of outstanding read requests is limited by the number of active processes in the system. Using disk scheduling techniques on small numbers of requests provides little benefit.

Disk scheduling techniques do work well in environments that can tolerate write caching such as Unix. Write caching can generate many write requests queued simultaneously. Careful disk scheduling can transfer the data to disk much more efficiently. Both the original Unix and Unix FFS file systems use disk scheduling in precisely this manner.

2.5. Fast Crash Recovery

The previous section focused on techniques for high performance disk storage management. This section examines techniques for recovering the storage after a system crash. The next chapter will argue that the reliability of disk hardware will be improving in reliability and availability due to technology such as disk arrays. The major cause of failures of systems will be software crashes[23]. It is the job of the storage manager to recover quickly from these crashes and ensure high availability of the storage.

The definition and semantics of crash recovery vary depending on the environment being supported. In transaction processing environments, guarantees provided by transactions give a precise meaning to crash recovery. Modifications made by all committed transactions must be present after a crash and modifications made by transaction that have not committed must not be present.

In the Unix environment transactions are not often used and crash recovery has a much less precise meaning than in the transaction processing environment. The use of write caching in Unix has meant that some data loss during crashes must be tolerated by applications. Unix file system crash recovery means restoring the disk storage to a state in which the metadata is consistent and ‘‘reasonably’’ up to date after a crash. Traditionally the crash recovery in Unix need only ensure that any data that is fsync’ed or older than 30 seconds survives.

The implementation techniques for performing crash recovery can be classified into two general classifications: careful updating and scavenger programs. The careful updating techniques, presented in Section 2.5.1, require the storage manager to apply the changes to the disk data in such a way that at any point the disk is in a consistent state. Careful updating greatly speeds and simplifies crash recovery because no actions to restore consistency are required after a crash, but it may impact the normal operating performance of the system. The scavenger program techniques, described in Section 2.5.2, use a special program that is run after a crash to check and correct any inconsistencies in the disk storage. When using scavenger program recovery a disk storage manager may let the storage become inconsistent because the scavenger will repair the inconsistencies after any crashes.

The final section (2.5.3) on crash recovery presents logging techniques that combine careful updating and scavenger programs to both improve recovery speed and normal operation performance. By keeping track of the modifications going to disk, the file system can speed the restoration of consistency. Logging also allows write caching of data to be used without data loss.

The file system operations in Unix that cause inconsistencies at crashes are those that require updating multiple disk blocks. An example is file creation that requires updates to both the directory containing the file and the inode of the file. The file name must be added to the directory block and the inode initialized. Furthermore, the file creation also modifies the bitmap of free blocks and the count of allocated files. Should only some of these modifications make it to disk because of the crash, the file system will be in an inconsistent state when the system is rebooted.

2.5.1. Careful updates

Careful update recovery techniques work under the principle that the simplest way to restore consistency after a crash is to never let the disk storage reach an inconsistent state. This is done by controlling the updates to disk so that each modification transforms the disk to a consistent state. Careful update techniques lead to very fast recovery time because typically nothing needs to be examined or changed before the storage manager can start accessing the disk again after a crash.

Two techniques are used to implement careful updates: ordered writes and shadow pages. Using ordered writes the storage manager orders all write requests to disk so that certain constraints are preserved. For example, the Unix FFS file system uses synchronous writes to order the operations that modified both a directory and an inode. For file creation example this means that the inode is always marked as allocated before it is added to the directory. The disk will never enter an inconsistent state with a directory pointing at an unallocated inode. As mentioned below, this type of ordering still can leave some other inconsistencies.

Ordered writes can be implemented by assigning an ordering to modified buffers that is used when writing to disk. An alternative way of ordering writes is to have the disk storage manager perform synchronous disk operations. Both of these implementation techniques can reduce file system performance. Ordering write requests limits the freedom to use disk scheduling techniques to improve disk efficiency.

The additional order limits the sorting of requests that can be done. This problem is especially severe when synchronous disk operations are used to enforce order as in done in Unix FFS.

A second disadvantage is that maintaining consistency is difficult when operations such as file creation can modify multiple objects (e.g. the directory entry and the inode.) Rather than always keeping the disk consistent, file systems typically use ordered writes to avoid the more destructive types of the inconsistency. For example using the file creation example above, the order between the inode and directory allows an inconsistency that causes space to be lost to an allocated but unreferenced file rather than having the inconsistency of a reference in a directory to an unallocated file. This ordered writes in Unix FFS limit the inconsistency on the disks but does not remove them altogether.

Another technique for keeping the disk data structures consistent is to not update the structures in place. Shadow pages[24-26] is one such technique that maintains disk consistent by having modifications create a new consistency copy rather than updating the old one in place. Figure 2-4 shows the use of shadow pages. Because both the old and the new copy of the object exist on disk, the storage manager can keep the disk storage consistent by maintaining the pointer to the old copy until the new copy has been totally written. If a crash occurs before the new copy is complete, the system will fall back to use the old

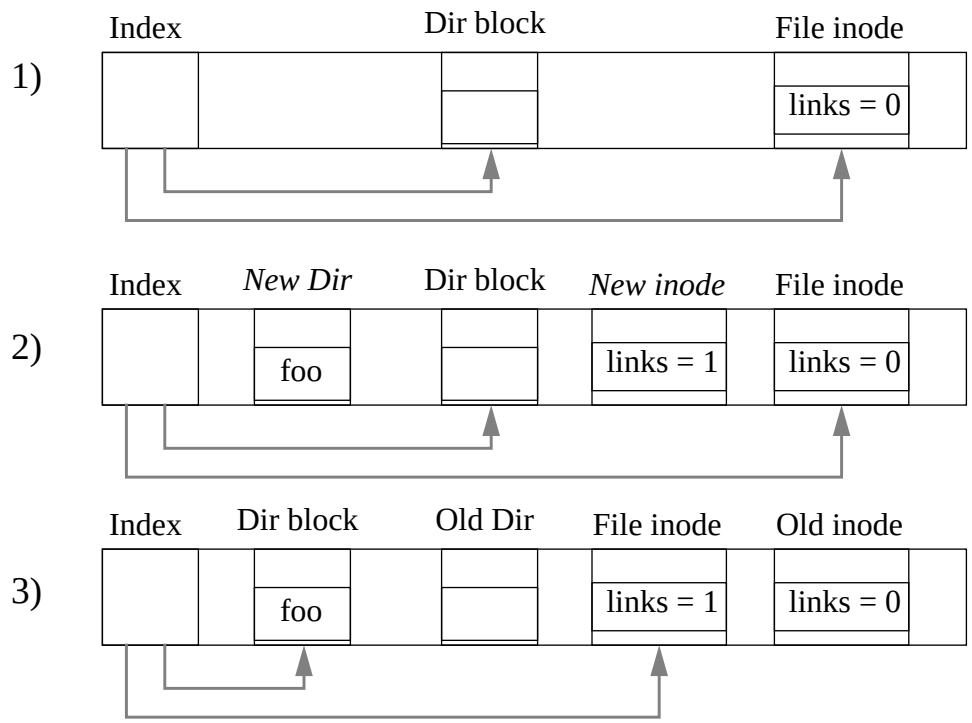

Figure 2-4 — Consistent updates using shadow pages.
This figure shows the use of shadow pages to keep the data structures on disk consistent while adding a file to a directory. The addition of the file requires modifying both the directory block (Dir block) and the file's inode (File inode). Step 1) shows the initial condition with the index block (Index) pointing at both the directory and inode. Step 2) shows the writing of the new directory (New Dir) and new inode block (New inode). Step 3) updates the index block to point at the new copies of both data structures.

copy of the object.

Shadow pages depend on the property that a disk block can be updated atomically. A single disk write can take the system from the old state to the new state. For the file creation example a single disk write must update the file system's idea of the current location of block the directory entry and the file's inode.

Because the update is not being done in place, shadow pages have the advantage that they impose less constraints on the disk scheduling and write caching techniques than ordered writes. Nevertheless, shadow pages still require some ordering of the writes in the update of the metadata. For example, once the new copy of an object has been written, the index for the object must be carefully updated to switch from pointing at the old to pointing at the new. For the small object size of the office/engineering environment the extra I/Os needed to update the metadata will have a strong negative influence on performance.

An additional problem with shadow pages is that making a separate copy tends to break any contiguous allocation or clustering done by the storage manager. For example in large files allocated contiguously, the shadow copy of a modified block can not be contiguous with the rest of the file. This reduces the benefits achieved by these disk allocation techniques. Furthermore, creating a separate copy can require much disk space for updates of large files. Because of this, shadow pages reduce the amount of disk space that is available for file allocation.

2.5.2. Scavenger program recovery

Scavenger programs remove the disadvantages of careful update by allowing the disk storage to become inconsistent and patching the inconsistency after a crash. Disk storage managers that utilize scavenger programs are free of the limitations on write caching and disk sorting imposed by the ordered writes. This allows the storage manager to take full advantage of these performance enhancing features.

For example, the Unix file system uses a scavenger program *fsck[11]* to restore consistency after a crash. Fsck checks for the inconsistencies such as disk blocks claimed by more than one inode and/or the free list and corrupted or incorrect directory entries. In order to check for these inconsistencies fsck must examine all metadata structures on disk including all inodes, indirect blocks, and directories. By scanning these data structures and correcting any inconsistencies, fsck can ensure that a consistent file system is always mounted.

The major disadvantage of disk scavenger programs is that the scanning of the metadata can take a long time. Operations such as examining every directory in the file system require many non-sequential read requests. For large disks with many files, the time to read and examine all the metadata can take many minutes. For example running on the hardware specified in Section 6.2, Fsck on a 300 megabyte disk with 23971 files take 117 seconds to complete. File servers with many disks can take tens of minutes to check all the disks. As disk get larger in capacity, scavenger-based recovery schemes will take longer and longer to recover.

The fundamental problem of scavenger programs is that they require examining the entire metadata structure regardless of how much of the metadata is inconsistent. Normally only a few metadata structures are actively being updated and possibly inconsistent at any given time. Scavenger programs would run much more quickly if they had knowledge of what objects were being updated and the locations of the possible inconsistencies. Keeping this information around to direct the recovery program is the principle behind the logging techniques presented in the next section.

2.5.3. Logging techniques

To track the locations of the latest modifications, logging systems reserve a portion of the disk to store an append-only data structure that lists the changes being made to the storage. This data structure, called the *log*, can be used by the scavenger program to quickly detect and correct any inconsistencies in the disk data structures. The crash recovery time of a logging system depends on the number of entries in the log and not the capacity of disk as is the case with scavenger programs. Since the storage manager can control the number of entries in the log it can control the recovery times.

Logging systems are usually implemented using *write ahead logging* in which the record of the change is written to the log before the changes are written to the storage. The log is considered the most up to date ‘‘truth’’ about the state of the storage. The data storage itself is allowed to lag behind the log. In the

Unix file creation example the changes made to the directory and the inode would be first written to the log and later to the data storage itself.

The lag between the modification written to the log and the modification written to the data storage represents work that must be done during crash recovery to bring the storage up to date. This work also determines the speed at which recovery can be performed. Recovery time is a function of the number of operations that must be checked by the crash recovery program. Logging systems limit the number of entries in the log that must be examined by implementing a *checkpoint* operation that ensures that the data storage is up to date and consistent with the log. Checkpoints work by writing to disk all data objects that contain modifications recorded in the log. After a checkpoint is complete the entire log contains redundant information and can be ignored during crash recovery. With checkpoints, crash recovery consists of starting at the last checkpoint and scanning forward in the log applying the recorded modifications. When the end of the log is encountered during the scanning, the data storage will have been brought up-to-date with the log. This process of moving the data storage forward to match the log is called *roll forward*.

A logging system can control the recovery time and performance of the system by controlling the frequency of checkpoint operations. Frequent checkpoints mean that few operations will need to be scanned during recovery and this results in fast crash recovery but performance impact during normal operation. Long checkpoint intervals result in longer recovery times but less performance impact from doing the checkpoints.

After a checkpoint the data storage matches the information in the log. Since the log is redundant, it can be truncated and the log space can be reclaimed without information being lost. Space occupied by the log is unavailable for storage allocation so limiting the size of the log allows more space for data storage.

If log space is not important, the log can be keep around by the storage manager to increase the reliability of the system. The log itself is a compact history of all changes that have occurred to the data storage. This history can be used to reconstruct the data storage should it become corrupted or lost due to hardware failure. The log can also be sent to another site to recover from site distasters that destroy all the storage at one site.

Logging can also improve performance as well as crash recovery time. The performance advantage is particularly noticeable for workloads with small objects and those workloads that cannot tolerate write caching. Consider the case of a workload of random writes to large files in an environment where write caching is not permitted. Without logging, each write request would cause a random disk write request. Disk scheduling techniques can not be effectively used. With logging the write request would be written first to the log. Since the log can be allocated sequentially on disk, all log writes are sequential. These log writes will be much more efficient than the random write done without logging.

Now that the data was been safely written to the log, the modification to the data storage can be delayed to improve performance. Using write caching, multiple writes to the same block can be combined together and sent to the data storage in one I/O. Furthermore, large numbers of blocks can be write cached, which makes disk scheduling techniques most effective. The result is that write caching can both reduce the number of blocks written and increase the efficiency of the actual writing.

Using two efficient disk writes per application write request (less if write caching combines requests) instead of one inefficient disk write per request allows logging to achieve higher performance. Performance can be further improved by allocating the log on a disk separate from the data storage and compressing the data written to the log. Having a dedicated disk for the log allows writes to the log to proceed in parallel with requests to the data storage disks. The log disk avoids seek delays because all disk requests are sequential log writes.

Because the log is read only during recovery, its format does not need to be efficient for reading. For example, only the changed bytes of a block need be written rather than the entire block itself. This optimization increases performance by causing less to be written to the log.

Although logging increases performance on some workloads, it also degrades performance for other workloads. The performance loss comes from the logging system writing of modifications twice, to both the log and the data disk. In particular, workloads that can tolerate write caching and workloads that already use the disk efficiently will have lower performance on a logging system because of the double writes. Clearly if write caching is acceptable then the first write to the log is overhead that slows the logging system. If the log is on a separate disk then the performance will not be reduced but the cost will be

higher because the of extra disk.

Workloads that already use the disk efficiently without employing write caching will also suffer write performance degradation using logging. For example, sequential write access to files allocated contiguously or in extents can achieve near 100% disk bandwidth. A logging system under such a workload will reduce write performance by a factor of two. Half the bandwidth used in the system will go to writing the data to the log and half to the data storage.

Logging is most commonly used in commercial database management systems. These systems are used in an environment where write caching is not acceptable and small updates are common. Logging has also appeared in several general-purpose file systems including Alpine[27] and a reimplementation of the Cedar file system[3]. Several recent implementations of the Unix file system interface have used logging to speed crash recovery and improve performance. Storage managers such as Episode[2] and AIX's JFS[28] use write ahead logging on changes to metadata structures to avoid the ordering and fsck problems in Unix FFS. The Echo storage manager[29] uses logging on all modifications to improve performance and reliability of user modifications. The databases and file systems using logging will be discussed in more detail in Chapter 7.

2.6. Summary

This discussion of current storage management techniques brings out three overall points that impact the design of log-structured file systems. First, the most difficult workload of storage managers are those containing small non-sequential accesses or many small objects. Currently used techniques are limited to disk efficiency of less than 25% for these objects. Second, Techniques such as contiguous allocation and extents allow disk storage managers to provide nearly perfect efficiency for large objects with sequential access patterns. Any performance improvement for workloads with such access patterns will need to come from the hardware. Third, Using techniques such as logging, storage managers can improve write performance, reliability, and at the same time speed crash recovery.

CHAPTER 3

Motivation

This chapter presents the motivation for new research in disk storage management techniques. The basic argument is that current disk storage manager design techniques are inadequate for the technology of the 1990s. More specifically, the current technology trends will cause many applications to bottleneck on the disk storage manager. While waiting for the disk storage requests to be processed, applications will not be able to fully exploit the other technological improvements such as faster processors.

Although the performance of all applications that make disk storage requests will be adversely affected by the current technology trends, this chapter identifies applications that make modifications to small objects as being the most seriously impacted. This observation motivates a storage manager design such as the log-structured file system that is optimized to handle workloads containing writes to small objects.

Storage manager design is governed by two general forces: workload, which determines a set of operations and how they should be carried out, and technology, which provides a set of basic building blocks. Workload and technology are not independent of each other. Computer system technology is designed to efficiently support application workloads within the constraints of the materials available. Workloads are determined by application program request patterns which are frequently designed to map efficiently onto the hardware technology.

The approach taken in this dissertation is to treat both the technology and the workload as unalterable forces driving the storage manager design. In addressing the challenges for disk storage managers, the log-structured file system design relies on neither specialized hardware technology nor alterations in application generated workloads. The following sections will examine both hardware technology and common application workloads.

3.1. Technology for Disk Storage Managers

Three components of technology are particularly significant for disk storage manager design: processors, disks, and main memory. This section examines the technology changes occurring in these technologies. Table 3-1 shows examples of the progression in computer systems technology described in this section. From these trends it is argued that many applications will become I/O bound and will not be able to use the faster processors as they are developed. I will also argue that workloads containing modifications to small objects will be the most adversely affected by technology changes.

3.1.1. Processor technology

Processor technology is significant to disk storage managers because the speed is increasing at a nearly exponential rate and these improvements seem likely to continue through much of the 1990s. This can be seen in Table 3-1 where processor speeds seem likely to increase by a factor of a 500 over the period from 1985 to 1995. Such an explosive growth in processing power places pressure on the other computer system elements to speed up so that the system does not become unbalanced.

Consider an example of a computer system with a one-MIPS CPU, 30 millisecond average disk access time, and an application that does one disk I/O for every 100 milliseconds of processor time. Assume that on this system the application program takes 600 seconds to complete. If the one-MIPS CPU was replaced by a 100 MIPS CPU, the application would complete in 154.8 seconds. The application only realizes a speed up of 3.9 times even though the CPU is 100 times faster. When the 100 MIPS CPU is replaced by a 1000 MIPS CPU, the application will complete in 120.5 seconds. The second increase of CPU power by a factor of 10 resulted in a speed up of less than 1.3.

Technology trends				
Technology		Year		
		1975	1985	1995
CPU	speed	1 MIPS	2 MIPS	1000 MIPS
Memory	speed	1 us	100 ns	1 ns
	capacity	.1 MB	10 MB	10000 MB
Disk	speed	60 ms	30 ms	8 ms
	capacity	100 MB	300 MB	3000 MB

Table 3-1 — Technology trends effecting storage managers
This table shows the trends in the technologies that effect storage manager design for mid-range machines used for engineering purposes. It includes data for the last 15 years and predictions for the next five years. The numbers from 1975 and 1985 are taken from disk storage manager design papers published in those years. The numbers for 1995 are generated from predictions and future product announcements in the literature.

This example illustrates what has come to be known as Amdahl's law[30]. If part of a system is sped up and part is not, the overall performance will be held back by the part not improved. For computing systems of the 1990s, the part that is improving is the processor and the part that is not is the disk. This leaves the application's performance bound by the disks.

3.1.2. Disk technology

The previous example assumed that disk technology is not improving. In fact disk technology is improving, but not as fast as CPUs. The improvements in disk technology have been primarily in the areas of cost and capacity rather than performance. Table 3-1 shows that the increases in disk capacity have tracked CPU performance but disk access times have not. The reason behind the lack of rapid performance improvement is the mechanical nature of the disk. Spinning a disk faster or moving a disk head more rapidly requires faster mechanics that cost more to manufacture and consume more power to operate. It is not surprising that the mechanics of disks have and will continue to lag behind the semiconductor technology in CPUs and main memory.

There are two main components of disk performance: access time and transfer bandwidth. The access time of disk is the time it takes for an individual storage request to finish. It is important for applications that must wait for storage requests to finish before proceeding. The transfer bandwidth is the measure of how fast data can be transferred to and from the disk. Transfer bandwidth is important to the performance of large requests where most of the request's time is spent transferring the data.

Throughput is a commonly used component of disk performance that is a combination of access time and transfer bandwidth. Throughput is a measure of how many requests per second the disks can handle. Throughput is important for workloads that have much concurrency in the requests.

Contrasted with the exponential growth in CPU speeds, the performance improvements of disks in terms of access time, transfer bandwidth and throughput have been very small. While CPUs have been doubling in speed each year, disk performance has taken ten years to double.

3.1.2.1. Disk array technology

Although the performance of individual disks is being held back by mechanics, disk storage performance is being improved by the use of disk arrays or RAIDs[7, 31, 32]. By spreading (striping) the data over many disks, storage managers can significantly improve the throughput and transfer bandwidth of the disk storage by accessing many disks in parallel. For example if a file is spread over 1000 disks, 1000 I/Os to the file can be processed at the same time to increase throughput three order-of-magnitudes over a single disk. Similar improvements in transfer bandwidth can be achieved if parts of the file are read from 1000 disks in parallel.

To exploit the large performance benefits of disk array technology, disk storage managers must place the data on the disks so that requests can use multiple disks concurrently. Unless multiple disks can be used to process requests concurrently, a disk array will be no faster than the individual component disks.

Storage manager design for disk arrays is further complicated by the redundancy necessary for large disk arrays. Although increasing the number of disks increases performance it also decreases the reliability of the storage because a failure of any single disk will cause part of the data to be lost. Spreading data over a large numbers of disks leads to unacceptable reliability unless redundancy is incorporated to mask disk failures. The techniques used to update the redundant information can reduce the performance advantages of disk arrays. Storage managers for disk arrays must both be able to exploit the parallelism in the array and be able to implement the redundancy efficiently.

The techniques used to implement redundancy vary widely in design and performance characteristics. This dissertation focuses on the techniques used for RAID level 4 and RAID level 5[7]. These techniques maintain blocks containing a parity value calculated over several data blocks. These parity blocks can be used to reconstruct data loss due to failure of a disk. These RAID techniques have also been proposed for workloads such as the office/engineering environment and seem to allow both high bandwidth and high throughput at low cost. Large files can be allocated across several disks in the disk array to provide high-bandwidth sequential transfers. Small files can be allocated on different disks so the files can be accessed in parallel.

The chief problem of these RAIDs is that writing a block on the disk may require as many as four separate disk I/Os. Writing a disk block requires updating the parity block which requires computing the new value of the parity from the old contents of the block being written along with the old contents of the parity block. Unless caching can be used to remove reads, each write request requires two reads and two writes. With each disk write potentially turning into four disk accesses, the benefits of RAIDs in the office/engineering environment lessen. Not only is the latency seen by individual reads no better than a single disk, the writes can be as much as a factor of four slower.

There are several things a storage manager can do to reduce the performance impact of the parity updating. By using additional memory to cache parity blocks as well as the data blocks, the number of disk I/Os may be reduced to as little as two (one to write new data, one to write new parity). The operations of read, modify, and write (read/modify/write) on the data and parity blocks can be reduced to only write operation each.. This still leaves small updates a factor of two slower because of the parity update.

The cost of parity can be reduced substantially if the storage manager can arrange write requests so that all the blocks in a stripe are modified simultaneously. When all the blocks in a stripe are updated together the new value for the parity block can be calculated from the data being written. There is no need to read the old values of either the data blocks or parity block. The parity block can be updated in parallel with the data blocks so it imposes little delay to the operation. Write transfers that involve all the blocks in a stripe are called *full stripe writes*. Full stripe writes are the key to efficient write accesses to RAIDs.

To use RAIDs efficiently, existing storage manager techniques must be modified to use striped files and ‘‘full stripe writes’’. The difficulty of these modifications depends on the workload being supported. Workloads with sequential access to large objects require only trivial modifications to storage managers to use full stripe writes. Contiguous allocation techniques and extents can allocate large objects across stripes so that sequential file writes turn into full stripe writes. Figure 3-1 shows the organization of a RAID level 4 disk array.

Full stripe writes are much harder to achieve for workloads containing small objects or small non-sequential transfers. For the office/engineering environment with small files, the updating of the inode and directory blocks effectively guarantees that full stripe writes will not happen with small files. There does not appear to be any easy way that current storage managers can be modified to achieve full stripe writes under these workloads. It is this problem that lead to the design of the log-structured file system that is presented in the next chapter.

3.1.3. Memory technology

The third technology affecting storage manager design is main memory. Like CPU technology, memory technology has benefited greatly from the improvements in semiconductors. Both the capacity and the speed of the memory hierarchy of computer systems have been improving dramatically in the 1980s. As seen in Table 3-1, the size of main memory has increased several orders of magnitude in the late 1980s

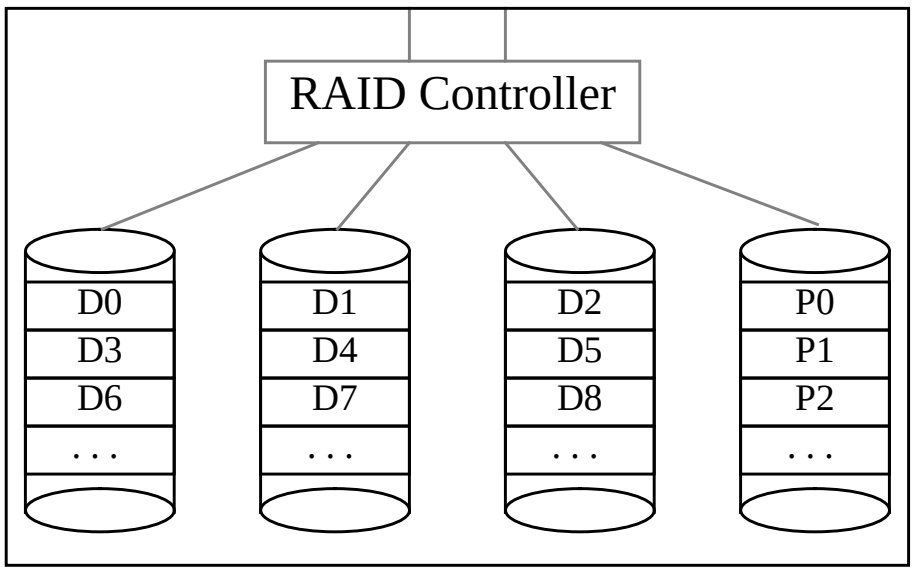

Figure 3-1 — Disk organization of RAID level 4
This figure shows the organization of a RAID level 4 disk array. The array's controller stripes blocks across three data disks and one parity disk to export the image of a single large disk. By accessing the data disks in parallel the disk array can provide higher transfer bandwidth and more transfers per second. The blocks on the parity disk contain the exclusive-or (XOR) of the corresponding blocks on the data disks. For example each byte in block P0 is the XOR of the corresponding bytes in blocks D0, D1, and D2. Should any one of the data disks fail, its contents can be reconstructed by XORing the remaining data disks with the parity disk.

RAID level 5 disk arrays differ from RAID level 4 in that the parity is not confined to a single disk. RAID level 5 rotates the parity block among all the disks so that updates to parity can be performed in parallel as well.

and early 1990s. The speed of the memory hierarchy, including the processor's registers and caches, have also scaled to match the improvements in CPU speed. As semiconductor technology advances the CPU technology, so will the memory get bigger and faster.

As main memory of computer systems gets larger and cheaper per byte, disk storage manager techniques that use main memory, such as read caching, write caching, and prefetching, will become more effective and less expensive to implement. As file caches get larger, the fraction of the read requests that can be serviced by the file cache grows. Large file caches will reduce the number of requests that must go to disk. This will effectively decouple application performance from that of the disk. For workloads that contain much locality, such as those measured in the office/engineering environment[1, 15], a file cache much smaller than the disk can have a hit rate high enough to significantly reduce the average wait time of a application and the number of read requests needed from disk storage.

Like read caching, prefetching of data into memory is helped by large main memories. With enough memory for application and file caching, aggressive prefetching can be implemented to reduce application wait time. There will be less need to worry about removing active files from memory to make space for prefetching.

Write caching techniques will also benefit from large file caches but less so than read caching. With larger file cache sizes, write caching will allow improved disk scheduling and absorption of multiple writes to the same object. The problem is that the amount of time that an object is write cache can not be increased without lowering the reliability of the system. Delaying writes for longer periods time increases

the chances that data will be lost to a crash that erases the contents of memory.

Unlike read caching with large file caches which can greatly improve performance, the benefit of write caching will be less pronounced. Most of the benefit will come from the ability to increase disk access efficiency by scheduling a larger number of write requests. The benefit from this scheduling will be only a modest increase in performance. Scheduling as many as 1000 randomly written blocks per disk only allows 25% of the maximum disk efficiency to be achieved[21]. This is much greater than the 5% to 10% achieved without scheduling but is still a factor of four less than the disk's maximum bandwidth.

With most reads being satisfied from caches in main memory and most writes still going to disk, disk I/O patterns will be fundamentally altered. Were traditionally disk have seen storage request stream with 3 to 5 times more reads than writes, large file caches will result in a request stream that contains a greater fraction of write requests. With most disk requests being writes, application performance will be determined by the rate at which the writes can be handled.

3.2. Workloads for disk storage managers

Once the technology for storage management has been established, the second factor that influences disk storage manager design is the workload. The workload determines the types of storage requests that must be processed as well as the frequency of the requests and the requirements mandated for the storage system. The request types define the operations the storage manager must support while the requirements define how fast and how reliably the requests must be processed.

The rest of this section presents the common disk storage request types and application requirements of several application areas. Before the workloads are presented several request classifications are included. These classifications allow requests to be described as either small or large in size and sequential or random in access pattern.

3.2.1. Storage request classifications

The storage requests generated by an application program can be classified as either small or large. Small requests are those in which the total disk access time is dominated by the seek and rotational latency rather than the data transfer time itself. Large requests are those in which data transfer time dominates the total time for the request.

The reason for distinguishing between small and large requests is that the access performance of small requests is unlikely to scale well with the disk technology improvements, whereas large requests will scale in performance. The access time for small requests is dependent on latencies of disk requests and is not going to be reduced dramatically in the future. On the other hand, large requests are limited by the bandwidth of the storage media which will see large improvements using disk array technology.

The actual sizes of small and large requests depends on the disk technology used. For disk of the late 1980s such as the Wren IV[33] with an average seek time of 17.5 milliseconds, 3600 RPM rotational speed, and maximum transfer bandwidth of 1.3 megabytes per second, a small request would be less than 12 kilobytes in size and a large request over 100 kilobytes. Using average seek and rotational latencies, the 12 and 100 kilobyte sizes allow the transfer time to take respectively 75% and 25% of the total access time. Disk arrays and other higher bandwidth disks will cause the sizes of both small and large requests to increase because the data transfer time of the requests will shrink.

In addition to their size, storage requests can be classified by their access patterns: sequential or random. Sequential access patterns occur when storage requests are made to an object in logically contiguous order. An example of sequential access is reading blocks from a file from beginning to end. Any access pattern that is not sequential is referred to as a random access pattern. An example of random access patterns is a large file containing bank account balances that is updated in a different place each time a check is deposited into an account.

Like request size, access patterns of requests also can be used as a predictor of future improvements in performance. Sequential access patterns will benefit more from disk array technology than random access patterns. The reason is that small sequential accesses can be combined into large sequential accesses while small random accesses can not. Random access patterns will still suffer the latencies inherit in disk mechanics.

3.2.2. Application areas

The above classifications are too broad to draw useful conclusions about which workloads will have performance problems in the future. This section refines the workload definitions further by providing example workloads from several application areas. For each application area, a brief characterization of the workload and and the effects of current technology trends is presented. Four application areas are presented: supercomputer, decision support, transaction processing, and office/engineering.

Supercomputer workloads

Supercomputer workloads are typified by the use of large data files in scientific experiments. A typical example would be a program that processes the output of sensors from a physics experiment. Access is mainly sequential to very large files[34]. Because of the high speed of the processor in this environment, the performance of the storage manager is of great importance.

The high bandwidth provided by disk array technology is a good match for the large sequential transfer of the supercomputer environment. In fact, the supercomputer environment has been a pioneer in the development of high bandwidth disk array techniques[35]. Using techniques such as contiguous allocation, current storage manager designs will achieve close to 100% of the disk maximum bandwidth on large transfers. Any additional performance improvements must come from hardware and not the storage manager techniques.

Decision support workloads

Another application area that will scale well with future computer systems is the use of database systems for decision making. In the decision support environment, data stored in a database is examined to provide information to guide decisions. An example of such as system might be a query that examines the prices of a stock over the last ten years in order to determine if a stock purchase should be made.

The workload in a decision support environment contains mostly reads with both random and sequential access patterns. Performance of these reads and availability of the system are important requirements. Because of the lack of frequent modifications, storage managers in the decision support environment can carefully allocate storage using clustering techniques and read caching to improve performance. Large memories and the parallelism of disk arrays will increase the effectiveness of both these techniques.

Transaction processing workloads

Another type of database access pattern occurs when databases are used in a transaction processing environment. A typical transaction processing workload might be the I/Os associated with cashing a check at a bank. The record containing the account balance, the bank's balance, and a history file are updated to reflect the transfer of money. Because money is involved the system must be very reliable; no data loss is permitted. Performance of the system is also important.

The workload in traditional transaction environment is characterized by many small random requests to large objects. Indexes are built and maintained to support random access to data but sequential scanning of objects is also sometimes necessary. Database systems use logging techniques to increase reliability and allow write caching and disk sorting to be used to increase efficiency.

The small random modifications of the transaction processing environment will mean that with current storage manager techniques the future technology will not scale the performance of individual transactions with processor speeds. Fortunately, much concurrency exists in transaction processing allowing the parallelism of disk arrays to be used to service many transactions simultaneously. Metrics used in transaction processing tend to stress throughput in terms of transactions per second rather than how fast an individual transaction will complete. Many applications will be satisfied by getting more transactions at the same latency.

Office/engineering workloads

The office/engineering environment includes workloads generated by applications used for program development, document preparation, interactive design, and simulations. An example of storage requests in this environment would be a compiler reading the source of a program and writing the compiled object file. 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, 15, 36, 37]. This environment also contains applications that access very large files. Most of these accesses are sequential although random access does occur and is heavily used by some applications. Request performance is of great importance in this environment; write caching and loss of same data has traditionally been tolerated for increased performance.

Because of the small file size and the high rate of update to files, the office/engineering environment causes problems for storage managers. Fortunately, file caching[26] has proven very effective for read requests in this environment. The remaining challenge for storage managers is to handle the write requests. Unlike the transaction processing environment, high levels of concurrency are typically not present in the office/engineering environment. Applications block until the current requests finishes so with current storage manager designs the parallelism of disk arrays will not help performance.

3.3. Future technology and workloads

The danger of designing a file system based on current technology trends is that some new technology will emerge and alter the technological influences on file system design in a different direction. Similarly, storage system designs based on predictions of future workloads can be upset by new applications that exhibit different workload characteristics or requirements. In the previous discussion, current technology trends were extrapolated to generate the technology of the future and the workloads of interest were assumed to be current workloads scaled to run on future technology. This section examines some alternative technologies and possible workload changes for the future.

3.3.1. Future technology

Current storage manager design is based on the attributes of magnetic disk technology. Should radical changes occur in non-volatile storage technology, the designs of storage managers could require major alterations. One of the attributes of magnetic disks that influences both current design and the log-structured file system is their access time characteristics. This section examines some of the other non-volatile storage technologies that could replace magnetic disks and alter the access time characteristics of non-volatile storage.

There has been a long tradition of alternatives to disk technology, each promising to reduce access time with similar storage cost to magnetic disks. Technologies such as CCD's and bubble memories have so far failed to displace disk technology. The most recent challenger to disk technology is the non-volatile memory used to build solid-state disks. Because these disks have no mechanical parts they offer zero seek and rotational latencies. Unfortunately, the current technology trends indicate that these solid-state disks will not be cost competitive with magnetic disk until the year 2000 at the earliest[38]. This still leaves magnetic disk as the storage media of choice for the 1990s.

It is possible that less dramatic changes will occur in storage technology that could alter storage manager design. An example would be the inclusion of fast non-volatile memory in the storage hierarchy. Having this non-volatile memory could reduce the importance of write request performance by allowing disk storage managers to use aggressive write caching without the data being lost in crashes.

Having non-volatile memory will help performance but high-write-rate systems will still need an efficient way to send the changes to disk when the non-volatile memory fills. Furthermore, write caching for long periods of time increases the changes that a software bug will corrupt the cached data before it is sent to disk. For these reasons a write-efficient storage manager will still be in demand.

3.3.2. Future workloads

Just as with technology, it is possible to use current trends in workload changes to extrapolate future workloads. A study by members of the Sprite project[15] examined changes in the office/engineering workload over a ten year period. The study found that the workload evolved to contain more I/O, greater bursts of I/O, and larger maximum file sizes.

One recent area of storage manager research that will facilitate new workloads is multimedia I/O[39, 40]. Multimedia workloads are characterized by access to continuous media[41] such as audio or video. The challenges posed to storage systems by continuous media are more stringent workload requirements rather than new workload access patterns. The continuous media requires strict guarantees on the

performance of storage managers. Processing requests faster than the guarantee does not speed the application while processing requests slower is unacceptable. The challenges of providing this additional guarantee are not addressed in this dissertation.

3.4. Research focus for the 1990s

The storage manager techniques presented in Chapter 2 applied to the workloads and technologies of the 1990s work well in the cases in which they already worked well. Sequential access to large objects will be able to exploit the bandwidth of disk arrays to improve performance. Using large memories, techniques such as read caching will improve performance as well. Unfortunately, the workloads that caused problems, such as small non-sequential modifications, will continue to lag behind. Exploiting the full benefits of disk arrays for these workloads will require new disk storage management techniques.

Barring some massive change in the current technology or workload trends, the grand challenge for disk storage managers will be to handle the workloads characterized by small object modifications. This dissertation addresses the workload of the office/engineering environment. Although the techniques developed here are applied to the office/engineering workload, similar techniques can also be applied to other environments such as transaction processing.

CHAPTER 4

Log-structured file systems

The previous chapters have presented the disk storage manager design techniques in use today and described why these techniques will have difficulty meeting the needs of office/engineering workloads of the 1990s. This chapter presents the disk storage manager design that resulted from these observations. The design, called a *log-structured file system*, is optimized for office/engineering workloads running of the technology of the mid-1990s. This chapter describes the basic concept of log-structured file systems together with a discussion of the major issues in implementing the design, including how files are allocated, written, and read.

As part of the discussion of log-structured file systems, the design of a prototype log-structured file system called *Sprite LFS* is presented. Sprite LFS was implemented in the Sprite network operating system and has been in use since early 1990. The design of Sprite LFS is presented in this chapter and some measurements from the running prototype are used to evaluate the design. A full description of the environment and workloads that generated the measurements can be found in Section 6.3.

This chapter and the next chapter, Chapter 5, contain descriptions log-structured file system technique and the Sprite LFS implementation. This chapters focuses on the choices of mechanisms for log-structured file systems while Chapter 5 focuses on the policy choices for the Sprite LFS implementation. Discussion of mechanisms is further divided into seven major sections:

1 The first section, Section 4.1, provides an overview on the concepts behind log-structured file systems. It presents the key design issues that are covered in the next two sections.

2 Section 4.2 presents the first of the design issues for log-structured file systems, the mechanism used to locate and read files. Also included in this section is the mechanism used by Sprite LFS to retrieve files.

3 Section 4.3 presents the free space management techniques used in log-structured file systems. The section details the two major choices for free space management.

4 Section 4.4 describes the segmented log structure used for free space management in Sprite LFS. This section includes several sub-sections that describe the mechanism of ‘‘cleaning’’ segments to regenerate free space from fragmented areas on the disk. The section also details the algorithms that control the layout of data on disk.

5 Section 4.5 presents the fast crash recovery mechanism implemented in Sprite LFS. This includes a description of the logging, checkpoint, and roll-forward mechanisms that make up the recovery algorithm.

6 Section 4.6 discusses the implementation of Sprite LFS in the Sprite distributed operating system.

7 Section 4.7 summarizes the chapter.

4.1. Overview of log-structured file systems

The fundamental idea of a log-structured file system is to improve write performance by buffering in the file cache all modifications to the file system and writing all the changes to disk sequentially in a single disk transfer operation. The information written to disk in the write operation includes file data blocks, file attributes, index blocks, directories, and almost all the other information used to manage the file system. By using this format, a log-structured file system can obtain high write performance regardless of the request size or access pattern of the workload.

The log-structured file system gets its name from the similarity of its disk layout to that of a log in the logging techniques presented in Section 2.5.3. In both cases the disk storage manager writes modifications sequentially to disk. The chief difference between the systems is that the log in a log-

structured file system is the only data structure on disk. No separate copy of the data storage exists.

Having the log as the only data structure on disk allows the log-structured file system to avoid the second disk write to update the data structure that is needed in systems that use write-ahead logging. Elimination of this second write allows log-structured file systems to utilize nearly 100% of the raw disk bandwidth on all workloads including those for which standard logging techniques can achieve only 25% to 50% of the 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 log-as-the-only-data-structure approach. The first issue is how to retrieve information from the log. For traditional logging systems, the log is never read during normal operations and only read sequentially during crash recovery. A log-structured file system would have unacceptable read performance if this same design were used. If a sequential search of the log were required to retrieve information, the system could suffer large delays from reads that miss in the file cache.

The second major issue of log-structured file system design is how to manage the free space on disk so that large extents of free space are always available for writing new data. One way to view this problem is that the log of the log-structured file system quickly fills the disk and wraps upon itself. During steady-state operation the log of a log-structured file system is continuously wrapping upon itself. The key issue in the design of a log-structure file system is to handle this ‘‘log wrap’’ problem as efficiently as possible.

4.2. Data location and reading

The log structure of the log-structured file system makes the allocation of data on disk trivial; data blocks are allocated and written when the log is written to disk. The challenge of file I/O in a log-structured file system is in the algorithms for retrieving data from disk. This section discusses the design techniques that allow log-structured file systems to quickly retrieve data from disk.

Although the term ‘‘log-structured’’ might suggest that sequential scans are needed for retrieval operations in a log-structured file system, this is not the case and it would lead to unacceptable performance in most environments. For example, the disk technologies currently used in the Sprite environment have capacities around one gigabyte and maximum transfer rates of two megabytes per second. Using these disks a sequential scan over the entire disk would take over eight minutes. Even with a cache read hit rate of 99.9% (which is much higher than any current file system achieves) the average read time would increase by 480 milliseconds, a factor of ten slower than the current storage manager provides. Clearly, retrieval mechanisms which increase the file cache miss cost five orders-of-magnitude are unacceptable.

The alternative to sequential scan is to have the log-structured file system maintain an index within the log which can be used to speed retrieval for requests that miss in the file cache. The index allows the storage manager to locate and read data items written anywhere in the log without searching. Every data transfer to the log will modify this index so its update must be efficient. To achieve this efficiency, the modifications to the index must be written to the log along with all other modifications.

To implement the fast retrieval needs of Unix, the Sprite LFS prototype builds index structures similar to those used in Unix FFS. For each file, Sprite LFS allocates an *inode* data structure that, like the Unix inode, contains the file’s attributes (type, owner, permissions, etc.) and the disk addresses of the first ten blocks of the file. Sprite LFS also uses the same *indirect blocks* and *double indirect blocks* that as Unix for handling large files.

The main difference between the index structures of Sprite LFS and Unix FFS is that Sprite LFS inodes are not in fixed locations on disk. This differences is key to reducing the number of seeks of workloads containing small file modifications. When the inodes are in fixed locations updates to inodes, directory blocks, and the file’s data blocks cause non-sequential and small transfers to disks; precisely the type of access patterns that cause poor performance. Figure 4-1 shows how the attributes, data, and index blocks of a file are allocated in the log in Sprite LFS and compares the disk layout of Sprite LFS with that of the Unix FFS.

Using the same indexing data structures as Unix FFS had several advantages for the Sprite LFS prototype. It was simple to integrate into the existing Sprite kernel and provided opportunities to reuse code from Sprite’s original Unix FFS-like file system. Furthermore, keeping the indexing structure the same meant that behavioral differences between Sprite LFS and Unix FFS could not be attributed to the index mechanism. This was useful in the evaluation of the log-structured file system.

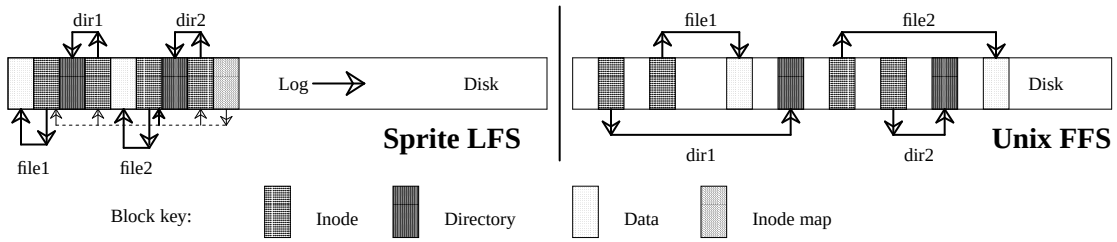

Figure 4-1 — A comparison between Sprite LFS and Unix FFS
This example shows the modified disk blocks written by Sprite LFS and Unix FFS when creating two single-block files named `dir1/file1` and `dir2/file2`. Each system must write new data blocks and inodes for `file1` and `file2`, plus new data blocks and inodes for the containing directories. Unix FFS requires ten non-sequential writes for the new information (the inodes for the new files are each written twice to ease recovery from crashes), 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 (see Section 4.2.1).

Using the same index structure also means that the number of block I/O's required to read a file from disk is identical in Sprite LFS and Berkeley FFS. This provides a bound for the maximum degradation of read performance in Sprite LFS.

By using the Unix file block index mechanism, Sprite LFS inherited both its advantages and disadvantages. Having an index pointer for each block of the file allows great flexibility for the file block allocation algorithms of the storage manager. Using this flexibility, Sprite LFS, like the Berkeley FFS, need only allocate to a file the number of blocks needed to hold the data. Like Unix FFS, Sprite LFS improved disk space efficiency by allowing the last block of the file to be a fragment of a block.

Sprite LFS also inherited the disadvantages of the Unix index structure. Having a pointer per file block means that for large files the size of the file index is large. A large file index implies more disk space to store it and more memory to cache it. On workloads where the block index is larger than the capacity of the cache, severe performance degradation can occur for non-sequential accesses. Each access might have to fetch from disk one or two index blocks to find the address of the file block to access. Each I/O could thus be 2 or 3 times slower than an extent-based system where the file map easily fits in memory.

4.2.1. Inode map

Although accessing file data given the inode is the same in Unix FFS and Sprite LFS, accessing the inode is slightly more complex in Sprite LFS because of the code needed to locate the inode. In Unix each inode is at a fixed location on disk; given the i-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.

To locate inodes quickly Sprite LFS uses a data structure called an *inode map* that maintains the current location of each inode. The inode map is structured like a large array indexed by the file's i-number. Each entry in the array contains the current disk address of one inode. To fetch an inode for a file, Sprite LFS indexes into the inode map to compute the disk location of the inode. Once the address lookup is done, file inode and data block access are identical to Unix FFS. The same inode and indirect block structure means that once the lookup is done the same code is executed and the same number of disk blocks are accessed.

Since the Sprite LFS inode map replaces what was a simple address calculation in Unix FFS, the data structure implementing the inode map should be both reliable and fast. Should the inode map become lost or corrupted, all files will be unaccessible or only available after very time-consuming sequential scans. The inode map is queried and updated frequently meaning these operations need to be fast. Ideally the

inode map lookup should take no longer than the address calculation done in Unix FFS.

To achieve high performance, Sprite LFS divides the inode map into blocks that are written to the log much like file data blocks. Writing the inode map to the log allows modifications to the map to be written to disk using the normal high write performance of the log. No extra disk seeks are needed to update the inode map.

The inode map needs to be reliably recovered after system crashes and shutdowns. To support fast startup of a file system, the locations of the inode map blocks are kept in a fixed region on disk called the *checkpoint region*. At file system attach time, the checkpoint region is read to find the location of all inode map blocks. The writing of changes to the inode map and the use of the checkpoint region for fast attaching and high reliability are described in detail in Section 4.5.

With the inode map divided into blocks, Sprite LFS can support high performance lookups in the inode map by caching frequently accessed blocks of the inode map in the same memory area used to cache file blocks. The key to high performance lookups in the inode map is to have the hit rate on the inode map be high enough so the average cost is close to zero disk accesses per lookup. To achieve this high hit rate requires that the active portion of the inode map fit comfortable in memory.

The amount of memory needed to successfully cache the inode map depends on the size of the map and the locality of the lookups. Size of the inode map depends on a file system creation parameter specifying the maximum number of inode map entries. Each inode map entry is 12 bytes in size and contains the current address of the inode, a flags field, a version number, and the time of last access of the file.

Although the inode map entry only needs the inode address to locate the file, having version number and the last access time in the inode map rather than the inode means that these fields can be examined and updated without fetching or updating the inode. This is particularly important for the access time which is updated on every read operation to the file. The use of the flag field is described below and the use of the version number is described in Section 4.4.2.

By default, the maximum number of inode map entries is set so there can be one entry for each 4 kilobytes of space. This means that the maximum size of the inode map will be equal to 0.29% of the disk space. Currently the size of main memory of the Sprite file servers is between 2% and 4% of the disk space on the machine. Thus the entire inode maps for all LFS file systems fit comfortably in main memory on the file servers.

The preceding argument about the maximum size of the inode map is based on the overly pessimistic assumption that the file systems have an average file size of 4 kilobytes (The average size on the Sprite file system range from 9 kilobytes to 60 kilobytes.) File systems with a larger average file size will have fewer inode map entries. Inode map blocks that contain no allocated entries are not given space in memory or on disks. The inode map can be as small as a single block for file systems with few files.

To allow the inode map to require even less space in the file cache, the allocation of files to i-numbers is done so that files in the same directory are given i-numbers close to each other. The flags field of the inode map entry indicates if the i-number is allocated. Directories are allocated an inode map entry at random and files created in the directory start at the directory's inode map entry and scan forward until the first available entry is found. This allocation policy means that normally all the files in a directory will reside in the same inode map block. Locality of references to files in the same directory causes locality in the references to inode map blocks. The inode map blocks containing entries from the directories currently being referenced will be in memory. The blocks containing entries for directories not currently being referenced are not needed in memory.

The combination of locality of reference and a larger average file size on the production Sprite LFS systems means the hit rate for inode map reference is very high. Hit rates range from 99.1% to 99.9% on the production file systems even though on average only 15% of the maximum inode map size is resident in the file cache (i.e only 3.6 megabytes of memory are used for caching inode map blocks of 8.1 gigabytes of disk space). Most misses occur because of cache cold-start effects when the file systems are first mounted after a reboot.

As technology scales up the size of the disk storage, the size of the inode map will increase but so will the size of the memories available to cache the blocks. Even with very large (1024 gigabyte) disk stores it seems unlikely that the inode map block cache miss rate will increase significantly. Large files and locality in reference patterns will also help ensure that the active portion of the inode map stays small

enough to fit in memory.

4.3. Free Space Management

The most challenging design issue for log-structured file systems is the management of free space. The goal is to maintain large contiguous free regions on disk so that the log can be written with large sequential transfers. When the file system is created all the free space is in a single extent on disk. By the time the log reaches the end of the disk and wraps around on itself, the free space will have been fragmented into many small extents corresponding to the files or blocks hat were deleted or overwritten. From this point on, the file system has two choices: threading or copying. Either the log can be threaded around the still live blocks or the live blocks can be copied out of the way. This section discusses the two approaches in detail and presents the combined threading-copying approach used in Sprite LFS.

4.3.1. Threading

The threading technique calls for the live blocks to be left where they are: new log blocks are written into the free spaces. Figure 4-2 shows how a log-structured file system using threading might look. The disk is treated as a large circular list of blocks and the log is written to the next free block.

The chief benefits of threading are its simplicity, the ability to operate at high disk capacity utilizations, and low free space management overheads when compared to the copying approaches. By scanning over the disk and writing to the free blocks a threaded log-structured file system can operate until the disk is 100% full. Keeping track of the allocated blocks is straightforward. A simple bitmap data structure like the one used by Unix FFS will work. The next block for writing the log can be located by scanning the bitmap in a circular fashion. The storage manager can continue operating as long as there is an unallocated block on the disk. This can be contrasted with some of the copying schemes under which it is prohibitively expensive to operate when the disk is nearly full.

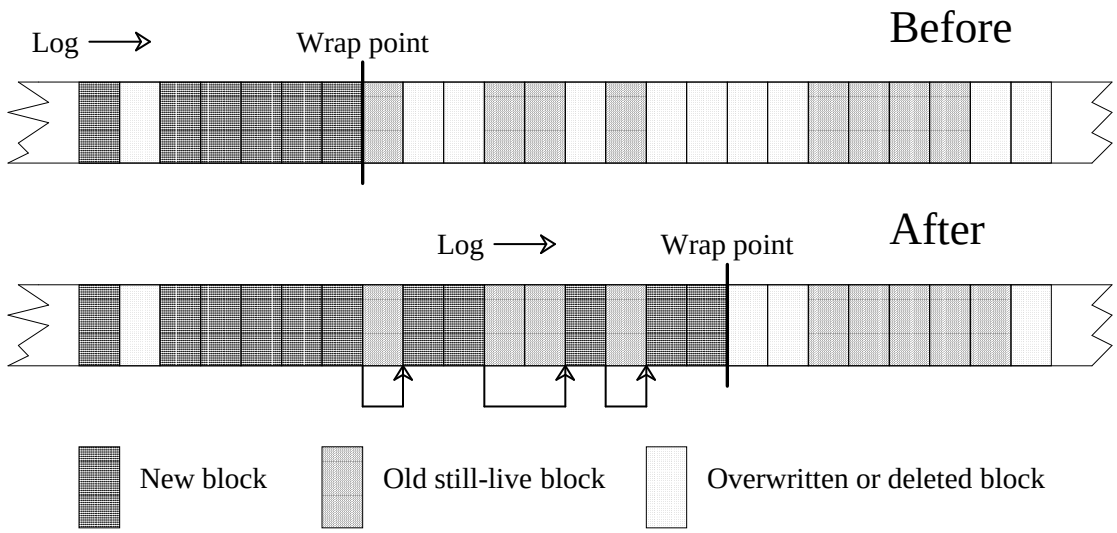

Figure 4-2 — A threaded log-structured file system
This figure shows the before and after disk image of log wrap on a threaded log-structured file system. With threading the log skips over the blocks that are still live and overwrites those blocks that have been previously deleted or overwritten. Pointers are needed to link together the disjoint pieces of the log.

The main problem with threading is that the free space on the disk will become fragmented so that large sequential transfers to the log are not possible. In a threading system, the large contiguous pieces of free disk space are overwritten with blocks of the log which are later overwritten or deleted to deallocate the space. Unless all data is overwritten and deleted, the large pieces of disk space will be broken into smaller pieces by the blocks that survive. These surviving blocks must be skipped over when the log wraps. The time spent skipping over blocks can not be used for transferring data to the log.

The reduction in performance caused by skipping over the live blocks depends on the number of live blocks and how the live blocks are clustered on disk. The number and clustering of live blocks is in turn dependent on the storage capacity utilization and workload. Operating under a low storage capacity utilization means that the disk contains few live blocks to skip over when the log wraps. For example, a log-structured file system running with only 10% of the disk allocated to live blocks will on average write 9 out of every 10 blocks that the log wraps on. Using the assumption that the disk can skip over a block at least as fast as it can write the block, write performance of at least 90% of disk maximum bandwidth should be obtainable regardless of the workload's access pattern. This example can be generalized to produce an estimate of the maximum bandwidth as a function of the disk capacity utilization.

Although a threaded log-structured file system can operate until all disk blocks are in use, the estimate for disk write performance indicates that performance will degrade as storage capacity utilization increases. Consider a disk operating at 80% storage capacity utilization, a common figure in the Sprite environment. The log-structured file system would have to skip over 8 blocks for every 2 blocks written thus achieving only 20% of the disk maximum bandwidth. As the disk fills the write performance of a threaded log-structured file system drops.

The above estimate of write performance may be overly pessimistic for systems operating at high storage capacity utilizations. It is possible that the workload overwrites or deleted files in such a way that the live blocks are clustered with other live blocks and the dead blocks are clustered with other dead blocks. This clustering would allow the log-structured file system to write in large transfers for high bandwidth and use fast seeks to skip over data. Unfortunately such clustering is unlikely in practice. The clustering of live blocks is controlled by what data is written together to the log and the clustering of dead blocks is controlled by the pattern of overwrites and deletions generated by applications. Neither of these clusterings is under the control the storage manager so the performance of threading depends on the workload.

The clustering needed for high performance threading relies on application with perfect locality in the write access patterns such that data that is written together is always overwritten or deleted together. This type of access allows the log to be threaded though the clusters of dead blocks and skipped over for the clusters of live blocks. Unfortunately, although the office/engineering environment has significant locality, it is not perfect. Most but not all files written at the same time will be deleted or overwritten together. The result is that over time the few that live will cause the free space to become heavily fragmented. The large contiguous disks transfers needed for high performance will not be available.

4.3.2. Copying

An alternative to skipping over live data during log wrap is to move the live blocks somewhere else. The live data can be read in and written back to disk in another location, allowing the log to be written to the newly cleared location. Copying allows a log-structured file system to avoid the fragmentation created by threading. By copying data out of the way, the log can always be written in large contiguous pieces. Figure 4-3 shows how a log-structured file system using copying might look.

The disadvantage of copying is that the copying overhead reduces the amount of the disk bandwidth available for application data. The time spent reading the live data in and writing it back out can not be used for reading or writing new data. Several factors control the amount of copying needed and the impact of this copying on storage manager performance. The factors include how the copying is implemented and the application workload. This relationship between workload, implementation, and performance is examined in detail in Chapter 5.

One fundamental issue that must be addressed in a copying scheme is where to copy the data. Possible solutions include copying the live data back in a compacted form at the head of the log, copying into another log-structured file system to form a hierarchy of logs, or copying the data into some totally different file system or archive. Each approach has its advantages but they all share the common disadvantage

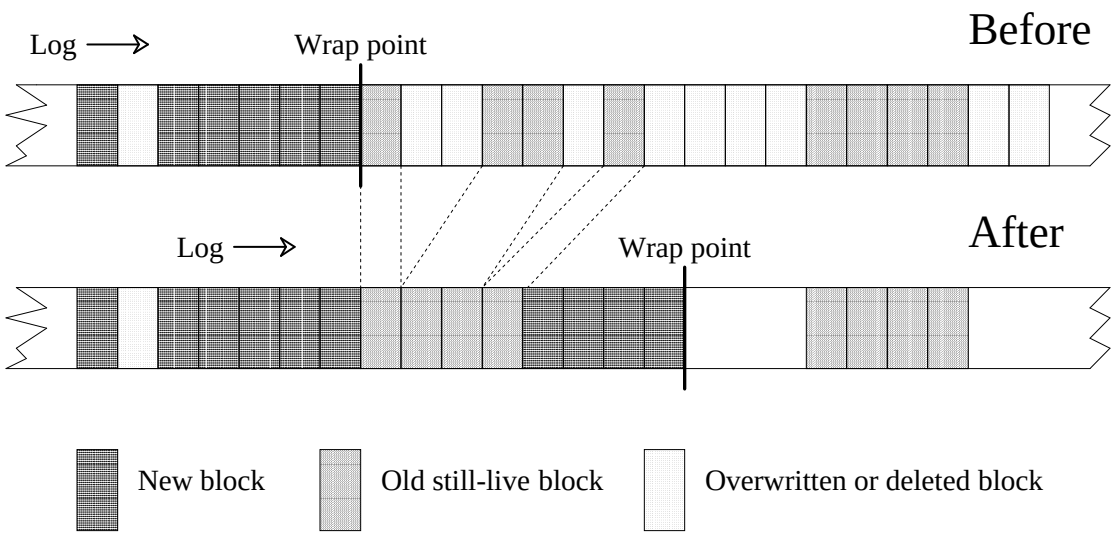

Figure 4-3 — A copying log-structured file system
This figure shows the before and after disk image of log wrap on a copying log-structured file system. Blocks that are still alive with the log wraps are read, compacted, and written back to the log.

that all long-lived data is written to disk multiple times. Valuable disk bandwidth for writing new data is lost to these multiple writes.

Copying the data back into the log is the simplest approach; only one data structure, the log, is needed. Under this approach the log wrap is dealt with by reading in the live data being wrapped on, compressing it into an unfragmented piece, and writing it back into the log. The mechanism to implement this form of copying is simple but it suffers the disadvantage that long lived data is continuously copied every time the log wraps upon it.

The other two approaches attempt to reduce the amount of copying by removing the data from the main log when it is wrapped upon. By copying the data into another log, a hierarchy of log-structured file systems can be formed with each log containing data of similar age. The hope is that the older the data in the log the slower it will be changing and the less copying will be needed. The hierarchy of logs approach lessens the problems of the single log approach because long lived data is not copied on every wrap of the main log. It still performs multiple copies of long lived data as the data moves through the layers of the hierarchy.

An alternative to the hierarchy of logs approach is to copy the data into another non-log-structured file system. Any files that live long enough to survive the first log wrap would be copied into a separate storage area. This second file system could use a format that works well for slowly changing read-mostly workloads. The hope is that the first log-structured file system would slow the rate of modifications to the point that the read-optimized file system could absorb the changes. Copying into a separate file system limits the number of copies of long lived data to two copies.

The disadvantage of this approach is that it suffers the same multiple copy problem of the hierarchy of logs. Every long-lived byte must be copied at least once. An additional complexity is caused by maintaining two different storage managers, a log-structured one and a read-optimized one.

Using a pure threading approach, a log-structured file system loses performance to fragmentation. Using a pure copying approach, a log-structured file system can control the fragmentation but loses performance to the copying overhead. What is needed is a combination of these techniques that uses copying to control the fragmentation and uses threading to control the copying overhead.

4.4. Sprite LFS - Threading and Copying

The solution to the log wrap problem used in the Sprite LFS prototype employs a combination of threading and copying. Sprite LFS copies live data back into the log to control fragmentation. When fragmentation is not a problem, a threading approach is used to quickly skip the unfragmented log sections. The design has the advantages of both the threading and copying approaches without the fragmentation overhead of threading or the copying overhead of copying. This section describes the mechanisms used to implement this hybrid approach. The policies that control these mechanisms are the topic of Chapter 5.

4.4.1. Segments

To ensure that large contiguous chunks of disk space are available for log writes, Sprite LFS divides the disk into large fixed-size extents called *segments*. The log is written to segments sequentially from the segment's beginning to its end, and all live data must be copied out of a segment before the segment can be rewritten. Segments also form the unit of the log threading. Once the log fills the current segment the log writing moves to another segment. Any segment may be chosen to be the next segment provided that it contains no live blocks.

To ensure a constant supply of segments available for writing the log, Sprite LFS uses copying to move the live blocks from heavily fragmented segments. This copying, called *segment cleaning*, generates empty segments by copying all the live blocks out of one or more of segments, combining the blocks together, and writing them to the log. The segment cleaning mechanism is described in detail in the next section.

Segments and the segment cleaning mechanism also allow slowly-changing data to be collected together and skipped over by the log. Long-lived and unchanging data living in a segment can be ignored by the log writing mechanism. This allows Sprite LFS to achieve both the benefits of threading as well as the benefits of copying.

The Sprite LFS design calls for segments to be large so 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. Thus whole-segment operations run at nearly the maximum bandwidth of the disk regardless of the order in which segments are accessed. Table 4-1 shows the achievable bandwidth versus segment size. Sprite LFS currently defaults to segment sizes of 512 kilobytes allowing Sprite LFS to achieve about 93% of the disk maximum bandwidth if the segments are transferred in random order.

Disk type	Segment size (kilobytes)								
	32	64	128	256	512	1024	2048	4096	8192
Wren IV	.45	.62	.77	.87	.93	.96	.98	.99	1.0
RAID-4	.17	.30	.45	.62	.77	.87	.93	.96	.98
RAID-8	.09	.17	.29	.45	.62	.77	.87	.93	.96

Table 4-1 — Achievable bandwidth for different segment sizes

This table shows the fraction of the maximum transfer bandwidth achievable with different segment sizes on three disk storage devices. The first device is a Wren IV disk[33] which has a maximum transfer bandwidth of 1.3 megabytes a second and an average access delay of 29.5 milliseconds (17.5 ms average seek, 8 ms average rotational delay, and 4 milliseconds controller overhead). The other devices are disk arrays with the blocks striped access either 4 (RAID-4) or 8 (RAID-8) Wren IV disks. For a single Wren disk a segment size of 512 kilobytes allows the log-structured file system to achieve 93% of the maximum transfer rate. For the disk arrays, segment sizes of 2 or 4 megabytes are required to obtain over 90% of the maximum bandwidth.

On disk arrays such as RAID level 5, segments are sized and aligned so that a segment occupies an integral number of parity stripes. This allows whole-segment writes to avoid the expensive read-modify-write on the parity blocks. Segments can be transferred with minimal parity overhead and near the maximum bandwidth of the disk array.

4.4.2. Segment cleaning

Because Sprite LFS writes entire segments at once, only segments that are known to contain no live blocks can be written. Sprite LFS refers to such empty segments as *clean segments* and only permits the log to be threaded through clean segments. To ensure a constant supply of clean segments, Sprite LFS uses a mechanism called *segment cleaning* that generates clean segments from segments containing live blocks. The segment cleaning mechanism works by copying all the live blocks out of one or more segments, combining the blocks together, and writing them to clean segments as part of the log. Once the data has been written back to the log, the original segments are clean and can be used for new data or for additional cleaning. The live data that was previously fragmented is now compacted together in the segments written during the cleaning operation.

The mechanisms used to implement segment cleaning in Sprite LFS are simple and flexible. Segment cleaning is implemented as a 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 (see Figure 4-4). This section details the basic mechanism used to implement segment cleaning. The segment cleaning mechanism is controlled by several policies that guide decisions made during the segment cleaning process. Policies

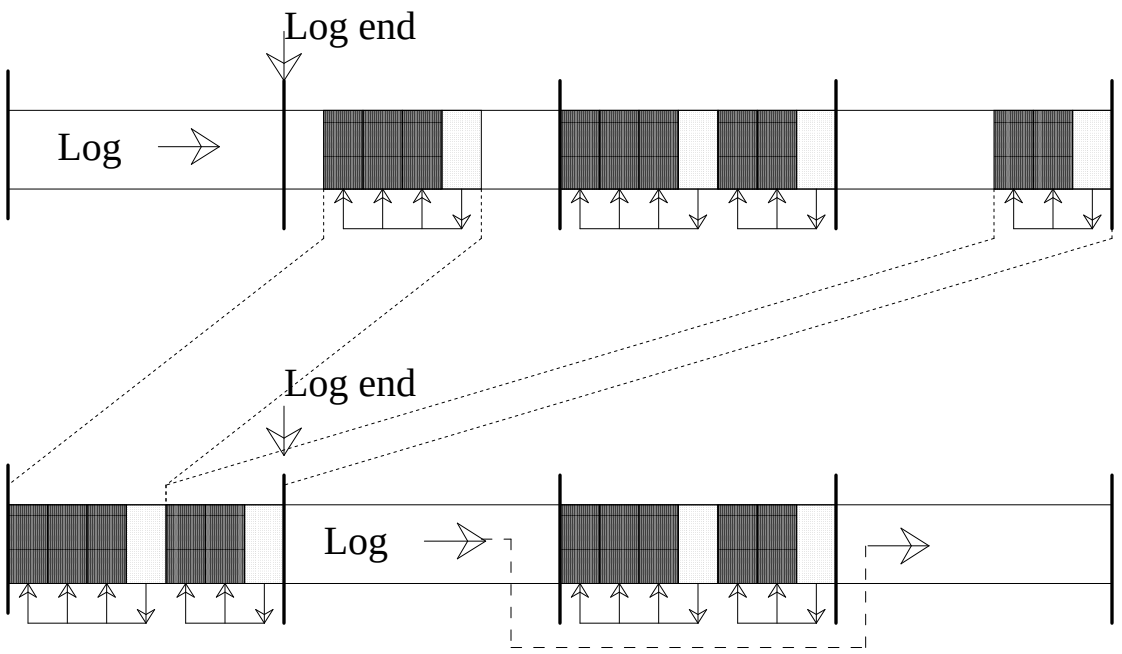

Figure 4-4 — Segment cleaning in Sprite LFS
This figure shows the use of segments and segment cleaning in Sprite LFS. In the top figure the log is seen wrapping upon itself. Then segment cleaning occurs, the fragmented segments are read, compacted, and written back to other segments. After segment cleaning the log has room to grow though the newly cleaned segments. Note that the log can skip over non-fragmented segments.

control events such as when cleaning is started and stopped, which segments are selected to be cleaned, how the live data of segments is combined together after being cleaned, and where the combined data is written. The effect of these polices is examined in Chapter 5.

The Sprite LFS implementation treats segment cleaning much as if the live blocks of the segment being cleaned were rewritten with the same data by an application program. Live blocks are brought into the file cache and written back to a segment in the log. Because of the similarity, segment cleaning in Sprite LFS is able to share much of the normal file system code.

The key difference between segment cleaning and normal file modifications is that during segment cleaning it must be possible to identify which blocks in a segment are live. For the live blocks it must also be possible to identify the file to which each block belongs and the position of the block within the file. This extra information is needed to update the file's inode or indirect block to contain the new location of the block when it is written out. Although most file systems maintain an index mapping a file's block to its location on disk, few support the reverse map from a disk block to its inode. When this reverse mapping is done such as during the disk scavenger program of the Unix file system, it is a very time-consuming process requiring scanning the inodes and indirect blocks of the file system. Using this mechanism would be unacceptably show for Sprite LFS.

Live block identification during segment cleaning is implemented in Sprite LFS by making the contents of a segment self-identifying. When a segment is written, enough additional information is added so that the segment cleaning mechanism can identify each block of a segment. For example, the additional information for a file block contains the file's i-number and block number within the file. When the block is moved during segment cleaning this additional information is used to find and update the file's inode to reflect the block's new location.

The overhead imposed by making segments self-identifying is low; all the additional information is known when the log is written to a segment. Furthermore, the additional information does not change over time so no work is required to keep the information up to date. The self-identifier information is small in size so it consumes little disk space and disk bandwidth to write. Section 4.4.6 describes how Sprite LFS formats segments to make them self-identifying and Section 6.6 shows the measured overhead of this mechanism on running systems. Besides being used for segment cleaning, the additional information simplifies the fast crash recovery algorithm described in Section 4.5.

Making segments self-identifying allows the segment cleaner to know what a block is or what it was but does not tell if it has been overwritten or deleted. The cleaner still needs a test to determine if a block is still live. Sprite LFS does this by checking the inode or indirect block to see if the block in question is still part of the file. If the block is pointed to by the file's block index it is still live; otherwise it can be ignored during segment cleaning.

One advantage of using this lookup mechanism is that it eliminates data structures such as bitmaps or freelists that are commonly used in file systems to track blocks in use. Having no bitmap or freelist benefits both the normal running of the file system and the crash recovery algorithm. During normal operation such data structures are constantly being modified, requiring CPU and disk I/Os. The crash recovery algorithm must be coded to restore the data structures to a consistent state before the file system can be used again.

However, elimination of the bitmap means that Sprite LFS does not know which blocks in a segment are live until it reads the segment. This precludes any optimizations to the cleaning mechanism that attempt to read only the live blocks of segments. With the bitmap, the cleaning mechanism could be optimized to read only the live blocks as indicated by the bitmap.

The value of this lost optimization depends on the clustering of live blocks in the segments cleaned. In order for the optimization to be a big win, the implementation must be able to read the live blocks significantly faster than reading the entire segment. For current disk technology and CPU technology, this only occurs when a segment's live blocks are clustered together into a few contiguous groups of blocks that can be read in a few transfers. Between the high disk controller overhead and the CPU time necessary to start a disk I/O, this is over 5 milliseconds of overhead per I/O. This overhead combined with missing disk revolutions between transfers means that when the number of transfers is greater than one per track it will not be faster than reading the entire segment.

A second disadvantage of the test Sprite LFS uses to determine if a block is live is that fetching the inode or indirect block may involve additional disk I/O's. If the inode or indirect block that describes a block is not in the segment being cleaned and not in the file cache, the cleaning mechanism must fetch it from disk to do the liveness test. In the worst case the lookup could cause several additional disk I/O's for each block examined. To keep the cost of cleaning low it is important that such random disk references be avoided.

In Sprite LFS, extra disk references will occur during the test only when the block's index is neither in the file cache nor in the segments being cleaned. For workloads with small files and those with large files written sequentially the index is likely to be in one or both of these places. For small files, the file data and inode data are likely to be modified and written together into the same segment. Reading the entire segment reads both the file's blocks and inode so no additional reads are required to check the block.

For workloads with large files written sequentially, the segments will contain many consecutive blocks of the same file. Checking the blocks will require few random disk I/O's because many of the blocks will be mapped by the same indirect blocks and inode. The random I/O request to fetch an indirect block will be amortized over many lookup requests using the same indirect block.

The workloads under which the lookup test causes the most random I/O's are those in which the index blocks and inode are separated from the blocks they map. This happens when large files are written non-sequentially and with little locality of reference. The data blocks mapped by a single indirect block will be scattered over the disk causing random I/O's to fetch the index when cleaning.

Using knowledge of the workload in the Sprite environment, Sprite LFS further improves cleaning performance by maintaining a file version number that is updated every time a file is deleted or truncated to length zero. The version number allows Sprite LFS to determine quickly and without additional I/O's whether a block is part of a file that has been totally overwritten or deleted. The version number for a file is kept in the file's inode map entry (see Section 4.2.1) and included in the summary information for each of the file's blocks in the segment. During the liveness test a quick comparison of the version number in the summary information and the inode map can determine if the block has been deleted.

In the workload environment of the Sprite LFS prototype, large-file random I/O without locality is rare so the lookup test performs well. With the truncate version number eliminating most of the inode lookups for deleted files, the remaining fetches of inodes require little disk I/O. Excluding the swap1 file system, only 1% to 5% of the inode fetches made during segment cleaning required any disk I/O. The needed inodes were either found in the file cache or in the segment being cleaned. The swap1 file system with its non-sequential access patterns to large files had a much higher number of disk I/Os during segment cleaning. Around 18% of the inode fetches missed in the file cache and required a disk I/O to fetch the inode block. The experience with swap1 suggests that an additional mechanism might be needed to reduce the extra disk I/Os during segment cleaning on file systems with workloads containing large files accessed non-sequentially.

4.4.3. Segment selection

Although the cleaning mechanism described above will work for any segment, some segments are better than others for efficient operation of the system. Segments that are heavily fragmented make a better choice than those that contain little free space. To guide in the selection of segments to clean and to keep track of the segments that are clean, Sprite LFS maintains a data structure called the *segment usage table*.

For each segment on disk the segment usage table contains an entry that describes the state of the segment, the number of live bytes in the segment, and an estimate of the age of the youngest block in the segment. The state bits tell if the segment is clean and therefore available for writing new data. The live byte count and the age are used by the policy for selecting segments to clean. Its usage is described in Chapter 5.

The segment usage table is modified by any operation that adds or deletes data from disk. The entry for a segment is updated when the log is written to the segment and when files are truncated or deleted. Entries are also updated when a block of a file or directory is rewritten causing the old block to become free.

Like the inode map, the segment usage table is broken into blocks that can be cached in the file cache. The size of the segment usage table depends on the segment size and the size of the disk. With the

512 kilobyte segment size of the current Sprite file system, the segment usage table occupies 48 kilobytes per gigabyte of disk space. The mechanism used to write the segment usage table to disk and recover it after a crash is described in Section 4.5.

4.4.4. Segment layout

This section presents the part of the Sprite LFS design that controls the organization of data on disks. The disk layout is important because it affects not only the read performance of Sprite LFS but also the segment cleaning overhead, crash recovery time, and space efficiency. This section describes layout of data in segments and the mapping of segments onto the disk media.

Because both write performance and cleaning overhead depend on efficient whole-segment transfers, segments need to be mapped onto the disk so that whole segment disk transfers are efficient. Segment size and alignment should be adjusted so that segments correspond to ‘‘natural’’ disk transfer units such as sets of tracks or cylinders. Such placement allows high bandwidth transfers that are not slowed by extra seeks or rotational delays. For disk arrays, the alignment and size should take the redundancy overhead into account so that segment write operations avoid the costly read/modify/write cycles needed for non-stripe-aligned accesses.

Once the segments have been sized and aligned for efficient transfers to and from disk, the Sprite LFS storage manager can focus on the placement of data inside segments. Issues that influence the layout of data in a segment include the read performance, data identification, and end-of-log detection.

Read performance
The whole-segment transfer mode used by log writes in Sprite LFS means that the write transfer performance does not depend on how the data is organized in a segment. The important performance metric driving the segment layout is the performance of read requests. Data of a segment is optimized to exploit common read access patterns to increase performance. In the Sprite environment the most common access pattern is sequential whole-file access.

Data identification
The segment must contain the additional information required for self-identification. This information should be placed in the segment so that the overhead in terms of log bandwidth and storage efficiency is low and the speed of identification during cleaning is fast.

End-of-log detection
A basic problem in a logging system is how the crash recovery algorithm detects when it has reached the end of the log. The mechanism used must work regardless of when the system dies while writing the log. For Sprite LFS this means it must be able to detect when a log write to a segment only partially completed because the system crashed.

4.4.5. Log regions and segment summary blocks

The identification of a segment’s contents and the end-of-log detection algorithm are complicated by the need to write partial segments as well as whole ones. Partial-segment writes are needed when modifications to the file system are insufficient to fill an entire segment yet still must be written to provide reliability guarantees (e.g. a client has invoked the `fsync` kernel call). It must be possible to fill in a segment piece-wise with each piece self-contained.

Sprite LFS addresses this need by dividing segments into contiguous extents called *log regions*. A log region contains zero or more data or metadata blocks combined with a *segment summary block* that contains the additional information describing the contents of the region. The log region is the smallest unit in which blocks are written to the log. It can range in size from one block to an entire segment. Figure 4-5 shows the layout of a log region containing several files. The rest of this subsection describes the contents of the log region in more detail.

Besides containing the self-identifying information for a segment, the segment summary block also contains pointers used to link log regions together to form a logically sequential log. The links are used to connect log regions in different segments as well as those within a single segment. Each segment summary block starts with a header containing the disk address of the previous segment summary block and the address of where the next segment summary block will be written (see Section 4.5.1.4 for implementation). A program can follow these pointers to traverse the log in both the forward and backward directions. Also

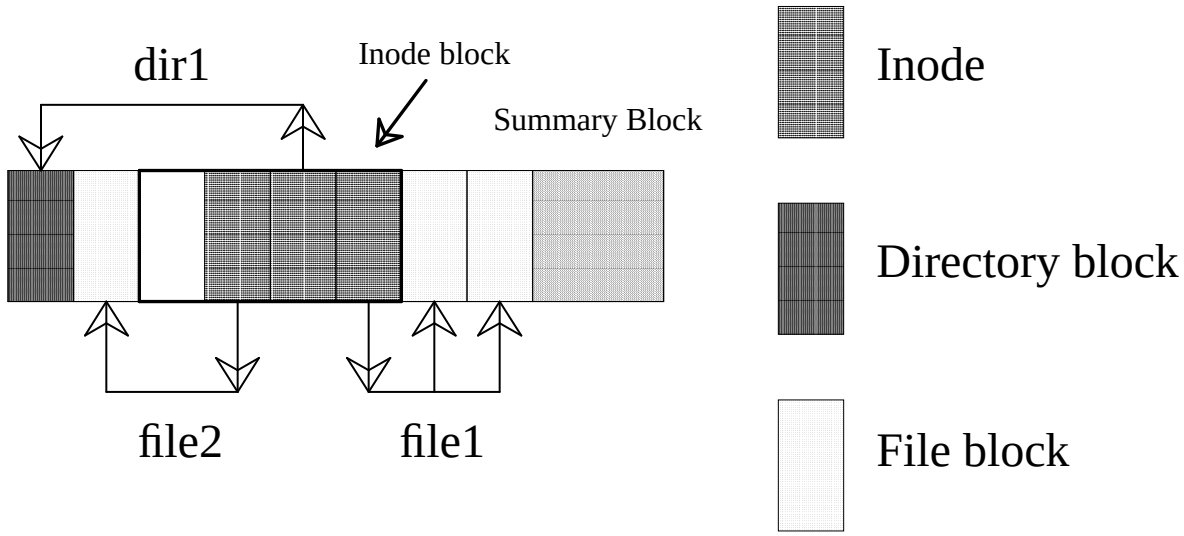

Figure 4-5 — Layout of a Sprite LFS log region
This layout shows the disk layout of a Sprite LFS log region that contains the inodes and data for two files (file1 and file2) and a directory (dir1). The log region is assembled from right to left. First a summary block allocated that will be modified to describes the contents of the rest of the log region. The summary block also contains pointers to the previous and next summary blocks. Once the summary block is placed, the data blocks of file1 are placed in the log region and an inode block is allocated for its inode. Next the file file2 and directory dir1 blocks are added to the region and their inodes placed in the inode block.

contained in the header is a monotonically increasing timestamp value whose usage is described in Section 4.5.

Using the summary block approach for data identification requires that two different blocks, the summary block and the data block, be read to identify the data block. The alternative approach would be to make every block self-identifying so the summary block would not be needed. This approach was rejected because it would require stealing bytes from file data blocks to provide the identification information. Since most application make file system block size requests, stealing as little as one bit from a block will cause application request to access multiple disk blocks.

4.4.6. Log region layout

With the segment summary blocks taking care of the identification and end-of-log detection, the remaining issue for the log layout on disk is the arrangement of the data for fast retrieval and high space efficiency. During the log writing process, Sprite LFS assembles an image of the log region in memory. It is not until this time that the disk addresses for the blocks in the log region are known. The layout code must place blocks into the region and update the indexes pointing at them. The rest of this section details the log region layout algorithms.

The questions to be answered are who decides when log region layout is to be done and what does the layout look like. The layout algorithm is invoked by a policy module in the file cache. It is the file cache that decides when and which modified file and metadata blocks need to be written to disk. The operation of this policy module is described in Section 4.5.1.

Once initiated, the log region layout algorithm runs as a separate kernel-resident process for each file system that needs to write data. The file cache code hands the layout algorithm the files whose attributes, data blocks, or block indexes have changed. The layout algorithm retrieves the modified blocks and

attributes from the cache and builds an in-memory image of the log region organized as a list of pointers to blocks in the file cache. When all the modifications are placed in the log region or the log region fills it is transferred to the current segment being written.

To help read performance, the log region layout algorithm clusters together the modified blocks from the same file. This is implemented by placing into the log region all the modified blocks of a file before moving on to the next file. For files written sequentially, this algorithm will allocate the blocks of the file contiguously and in order in the log region. Sequential read requests to the file can then be processed sequentially from disk.

For each block placed in a log region, the algorithm must update two data structures: the segment summary block and the index of the block. The addition of a block in a log region is recorded in an entry added to the summary block of the log region. This entry contains the i-number of the file, the block number being added, and the truncate version of the file from the inode map. Placing the block in the log region also determines the disk address at which the block will be written. The inode or indirect block is updated in the file cache to reflect this new address for the block.

When placing a file in a log region, the data blocks are placed first followed by the indirect blocks followed by any double indirect blocks and finally the inode. Placing blocks in this bottom-up hierarchal fashion means that blocks are never modified after being placed in a log region. Furthermore, the layout algorithm is the same for indirect blocks as it is for data blocks. Placing an indirect block causes the next higher level of index to be updated in addition to adding an entry to the segment summary block. In the entry, negative block numbers are used to distinguish the index blocks from the file's data blocks. These block numbers are assign using a depth first scan of the tree of indirect blocks (i.e. the single indirect blocks is -1, the double indirect block is -2, the children of the double indirect block are -3 through -1026, etc).

The final step of a file's layout is to place the inode in the log region and update the file's inode map to point at this new inode. This step is both important for crash recovery and problematic for the layout code. Until the inode is written and the inode map updated, the blocks written to the log region will not be visible on disk because it is not part of the file's index rooted at the inode. The crash recovery algorithm described in Section 4.5 uses this property so that the inode acts like a data base commit record that decides when to switch the on-disk version of a file to include the newly written changes.

The problem with placing the inode is caused by its small size. The inode in Sprite LFS is only 128 bytes so allocating an entire 4 kilobyte block or even a 512 byte sector will waste much disk space due to internal fragmentation. To reduce the fragmentation, Sprite LFS clusters the inodes of a log region into a block called an *inode block*. The size of the inode block is a file system settable parameter that defaults to 4 kilobytes.

Once all the modified blocks of the file have been placed in a log region, the layout algorithm places the inode of a file into the inode block and updates the inode map entry of the file to point at the inode block. Inode blocks are allocated in the log region whenever an inode needs to be placed and no space exists in the current inode block. For log regions with many small files, the files' inodes will be placed into a few inode blocks with low fragmentation. Figure 4-5 shows the layout of a log region with a single inode block. If need be, all blocks of a log region can be allocated to data blocks with no inode blocks.

On the prototype Sprite LFS file systems with four-kilobyte inode blocks, on average only half the entries in the inode blocks were in use. The distribution indicated that having a smaller inode block would not have helped much because the blocks were either nearly full or contained only a few live inodes. One of the reasons for this was that the crash recovery mechanism of the Sprite distributed file system modifies the inodes of the files recently used by a client machine when the client crashes. This caused more inode modifications than Sprite LFS anticipated. In hindsight, the crash recovery version number should have been put in the inode map like the file's access time.

Clustering the inodes together into an inode block has benefits beyond reducing fragmentation. By fetching the entire inode block when a reference to an inode in the block occurs, Sprite LFS implicitly pre-fetches the other inodes in the block. Any locality of reference to inodes in the same inode block is exploited. The inode blocks are cached like file blocks in the file cache. Since the higher levels of the file system cache individual inodes, the inode blocks act as a second-level cache.

On the production Sprite LFS file systems, most access to inodes find the inode block already in the cache. Except for the swap1 file system, the cache hit rates on the inode blocks are in the range 90% to 96%. The hit rate on the swap1 file system was only 75% for two reasons. The first reason was that swap1 had relatively few files so the caching of inodes in the higher levels of the file system was very effective. Only the cold start misses referenced the inode blocks in the file cache. The second reason was that swap1 also suffered worse fragmentation in the inode blocks. The average number of inodes was only four per block and most (60%) inode blocks contain less than three inodes. The heavy fragmentation of the inode blocks meant that the caching was ineffective for swap1.

To reduce fragmentation and improve space utilization, the Sprite LFS log region layout algorithm allows the last block of a file to occupy less than a full 4 kilobyte block. The number of sectors allocated to the last block of a file is rounded to an integral number of 512-byte disk sectors. Like the Berkeley Unix FFS, allowing the last block to be a fragment greatly reduces the intra-file fragmentation in file systems with many small files that are not integral numbers of blocks. It is also possible to round fragments to even smaller units than disk sectors and achieve greater fragmentation reduction. This was not done in the current Sprite LFS implementation because addressing the disk on more than sector boundaries would require increasing the size of disk pointers.

4.5. Crash recovery

The last part of the Sprite LFS design to present is the crash recovery system. A crash in Sprite LFS is defined to occur any time the file system's memory-resident data structures are lost. Because of the write caching and multi-object updates done in Sprite LFS, crashes can leave the disk storage in an inconsistent state. Crash recovery is the act restoring the disk to a consistent state in a way that preserves the guarantees made to application programs. It is the goal of Sprite LFS for crash recovery to be fast and the impact of the recovery method on normal operations to be low.

With its log structure, Sprite LFS can use checkpoint and roll-forward recovery mechanisms similar to those used by the logging systems described in Section 2.5.3. Within these mechanisms, there are many implementation choices that effect both the recovery speed and the speed of normal operations. These choices include when and in what format data is to be written to the log. The basic choice boils down to either writing modifications to the log immediately or delaying and using the roll forward mechanism to reconstruct the changes after a crash. Section 4.5.1 describes the writing policy, which determines when data is written, and the roll forward algorithm, which determines the format. Section 4.5.2 presents the checkpoint algorithm used to limit the amount of work required during roll-forward.

4.5.1. Roll forward

The main design issues for the roll-forward algorithm in Sprite LFS are the writing policy and the log-following algorithm. The writing policy decides when data and metadata is written to disk and in what form it is written. By using a different write policy for different metadata structures, Sprite LFS is able to reduce the amount of data written to the log at the expense of doing more work to regenerate the changes should the system crash. The log-following algorithm is used to follow the log, starting at the last known consistent state (a checkpoint) and ending at the last complete log region that was written before a crash. The concern for this algorithm is to follow the possibly non-contiguous log and correctly locate the last completely written log region. The algorithm is complicated by the need to deal with crashes that cause partially written log regions.

4.5.1.1. Writing policy

Sprite LFS uses different writing policies depending on the type of data being modified. Data generated by application programs is pushed to disk in a timely fashion so as to limit the amount of application data lost during a crash. The file system metadata is written using a different policy that writes indications of the modifications rather than the modified blocks themselves. This reduces the log bandwidth consumed by metadata.

The writing policy for application data in Sprite LFS is designed to match or exceed the level of guarantees provided to applications by Unix FFS. The guarantees are that application data is assured of being written to disk once it has lived 30 seconds or once an *fsync* system call on the file has returned. To implement this policy, Sprite LFS will delay placing a file in a log region and writing it to disk until one of

four conditions is met:

1) A file contains modifications that have been write-cached more than 30 seconds.

2) A process is waiting for an *fsync* request to complete on the file.

3) A process is waiting for a sync request to complete on the file system containing the file.

4) The file cache manager detects that the caching is filling with modifications to files for the file system.

Under these guidelines, Sprite LFS delays writing a file for as long as 30 seconds unless it is requested to write the file because of an *fsync* request or because the cache is being overrun with modifications.

The writing policy used for metadata depends on the metadata involved. Inodes and indirect blocks are written using the same rules as application data. When a data block is written, the inode or indirect block that maps that block is also written. The motivation for this policy is improve read performance by keeping the inodes and indirect blocks clustered with the data that they map. Furthermore, the policy of always writing the inode and indirect blocks relieves the roll-forward code from having to deal with updating file index during recovery. For example, Sprite LFS could have further increased performance by delaying the writing of the file index and having the recovery code regenerate the index if the system crashes. This could help large file performance but small files would still need the inode to be written to disk promptly because of the attributes (permission, file ownership) it contains.

Writing indirect blocks and inodes promptly also has desirable properties for dealing with crashes during updates to a file. Since a file's inode is written to the log after all modified blocks of the file have been written, the inode acts as a commit point that switches from the old version of the file to the new version. If the system crashes between the time when the system has started writing the blocks of a file and before it completes the writes, the inode map will point at the old inode and old version of the file. For files being written during a crash, the old version or the new version of the file will appear after reboot and not some partially updated version. This is a better guarantee than provided by Unix FFS .

The inode map and segment usage table use a different write policy and recovery algorithm than is used for the inodes and indirect blocks. The reason for this is that blocks of the inode map and segment usage table are modified on every log region write. If every log region had to contain several inode map and segment usage table blocks, much of the log bandwidth and disk space would be consumed by these data structures. Consider the example of a log region containing a newly written one-kilobyte file. Writing the metadata immediately would require the log region to containing a four kilobyte inode map block as well as a four kilobyte segment usage table block. The size of the metadata would be many times that of the data.

Rather than writing inode map and segment usage table blocks in every log region, Sprite LFS writes only enough information so that the roll-forward algorithm can reconstruct the changes made to a block of the data structure since the last time the block was written to disk. In many cases, no additional information besides the summary block is needed. For example, the existence of inodes in a log region implies changes to the inode map, and the existence of files written to a segment implies changes to the segment usage table.

4.5.1.2. Directory operation log

The append-only nature of the log data structure and the Sprite LFS writing policy means that certain consistency guarantees can be made to the crash recovery program. The policy of writing the file index bottom up with the inode last means that the inode always points to the correct blocks of the file regardless of when the system crashes. Unlike traditional Unix file systems in which the inode can point to blocks that are not part of the file, Sprite LFS can assume that the most recent inode of a file is consistent.

Sprite LFS does share a problem with the Unix file system in dealing with consistency between different objects such as a file and a directory. Name-related operations such as file creation update two objects: the file's inode and the directory entry. If a crash occurs after one object has been written to disk but before the other is written, then inconsistencies can result (e.g. a directory entry might refer to an inode that appears to be free). This recovery program must correct these inconsistencies.

Inconsistency in the naming data structures consist of discrepancies between the number of directory entries pointing at an inode and the link count in the inode. Operations such as creating files and making

hard links increment the link count and add a directory entry. Operations such as unlink decrement the link count and remove a directory entry. The Sprite LFS crash recovery program must be able to detect when only one of the two modified data structures was written to disk. To accomplish this, Sprite LFS maintains an additional data structure called *directory operation log* which records all changes to directory entries. The directory operation log entries provide a single object that completely describes both parts of the operation to be performed. The writing of the entry to disk servers as an atomic commit for the operation. The rest of this subsection will detail the format and algorithms used for this data structure.

The directory operation log is a write-ahead log containing a list of directory operations in the order that they occurred. Each directory operation causes a log entry to be added to the in-memory directory operation log block. The directory operation log block is written to disk at the beginning of each log write operation. Pushing the directory operation log first ensures that a log entry is always written before either the inode or the directory block containing the changes specified in the entry. Each entry contains three fields:

Opcode field
: An opcode that specifies the type of operation being performed. The possible operation codes are create, link, unlink, and rename. These opcodes correspond to the creation of a file, adding a link to a file, removing a link to a file, and renaming of a file. The rename opcode is subdivided into two opcodes specifying the adding of the new name and deletion of the old name.

Directory location field
: The i-number and block index of the directory entry being modified.

Directory entry field
: A copy of the directory entry being modified. This consists of the file name and i-number.

The directory log entry is a variable-length data structure ranging in size from a minimum of 32 bytes to a maximum of 284 bytes. The length of the entry depends on the length of the file name in the directory entry field.

4.5.1.3. Roll forward algorithm

Sprite LFS implements roll-forward using an algorithm that takes two passes over the data written since the last checkpoint. The first pass updates metadata such as the inode map and segment usage table, and the second pass ensures that the directory structure is consistent. Doing the roll-forward as a two-pass operation simplifies the algorithm at the cost of increasing the time to takes to do recovery. The recovery algorithm caches the data read during the first pass to speed the operation of the second pass. When the amount of memory is sufficiently large to hold the entire first pass, no disk reads are needed during the second pass.

The goal of the first pass of the roll forward algorithm is to record enough information so that changes made to the inode map and segment usage table that were lost during the crash can be regenerated. During the pass the contents of all blocks written since the last checkpoint are identified. For most blocks this simply means examining the summary blocks of the segment. When inode blocks are encountered, the i-number and location of the inodes are recorded. For each i-number only information about the most recently written copy of the inode needs to be recorded. Also recorded during the first scan are the segment numbers of the segments though which the log passed. These are the segments that were marked as clean in the last checkpoint but may now contain live files.

Once the inode locations and recently written segment numbers are recorded, the in-memory copies of the inode map and segment usage table are updated to include this information. The first operation is to mark the segment table entries for the segments through which the log passes as no longer being clean. Updating this information early in the recovery algorithms allows the later stages of recovery to know which segments are clean and which contain live blocks written since the last checkpoint. With this information the recovery program can to append data to the log without fear of overwriting live data. This is important because it allows the recovery algorithm to run in less memory than would be required if it were buffering all changes in memory.

Along with the segment usage table, the in-memory copy of the inode map is updated to reflect the new inode locations recorded during the first pass. Updating the inode map also requires making changes to the segment usage table to reflect the new locations of files. Changing the location of an inode means

the active bytes count of the segment containing the old copy of the new inode must be decremented and the active bytes count of the segment containing the inode must be incremented. In addition, block pointers in the new inode and its indirect blocks are compared against those in the old copy of the inode. Any block locations that have changed require updates to segment usage table entries. Again, this means decrementing the active byte counts for the segments containing blocks of the file that have been overwritten or deleted and incrementing active byte counts for any segments containing blocks written since the last checkpoint. This comparison between the block locations of the new and old inodes is carried down the entire file block index including the indirect and doubly indirect blocks. Once all the latest copies of all files have been processed, the in-memory inode map and segment usage table will be up to date with the modifications written since the last checkpoint.

The second pass of the recovery algorithm uses the directory operation log to ensure that directory entries are consistent with i-node link counts. Any inconsistency detected during this pass will be corrected and the changes written back to the log as new versions of the files or directories. An inconsistency manifests itself by an inode and directory block pair in which one or both of the pair do not contain the changes specified in a directory log entry. To restore consistency, the recovery program can either apply the necessary changes to *roll forward* the operation or undo the changes to *roll backward* the operation. These choices allow the recovery program to move to a consistent state that either includes or does not include the operation.

The Sprite LFS recovery program opts to roll forward operations whenever it can. Always going forward simplifies the recovery because undoing one operation can sometimes cause other operations to become inconsistent and require undoing. An example is undoing the creation of a directory, which requires subsequent creations of files in that directory to be undone. The recovery program will roll forward even when the system is in a consistent state because neither of the changed objects made it to disk.

Using the information recorded during the first pass about the locations of objects, the recovery algorithm is able to quickly and simply detect inconsistencies. The writing of directory operation log entries ahead of any file system changes described by the entries means there is a relationship between the position in the log of the entry and the changes described by the entry. For each entry in the directory operation log, the recovery program can classify the inode and directory block as either being *forward* or *backward* depending on whether the most recently written version of the inode or directory block occurs in the log after or before the entry, respectively. An object classified as backward can not include the changes specified by the entry and an object classified as forward must include the changes specified by the entry.

The second pass of the roll forward scans forward through the directory operation log classifying each operand of each entry as either forward or backward. Classification of an inode operand is done by comparing its address in the inode map with the address of the log entry. Directory block classification is done by comparing the address of the directory block (i.e. a pointer the an inode or indirect block) with the address of the log entry. Given two disk addresses, the classification uses the list of segments generated from the first pass to determine which address comes first in the log. The classification process is simple but may require doing some extra disk read requests to lookup the inode or indirect block of the directory. Note that the directory and inode themselves do not need to be fetched; only the inode map and directory inode that describes their locations are needed.

Operation log entries that have both the inode and directory block classified ‘‘forward’’ are known to be up-to-date and consistent. No modifications to the file system are required for these entries. The rest of the classification possibilities require some corrective action that depends on the log entry opcode. If the inode is backward the corrective action is to update the link count. For the link opcode this means incrementing the link count by one and for unlink opcode decrementing it by one. If the unlink operation is the last link to the file, the file’s i-number is marked free in the in-memory inode map and the segment usage table is updated to reflect the deletion of the blocks of the file.

If the directory block is classified as backward, the recovery program can either add or remove the directory entry from the block. For the link opcode the directory entry in the operation log entry is added to the directory block at the specified offset. The unlink operation causes the directory entry to be removed from the directory.

The one problem for the roll-forward algorithm occurs when a file or directory creation record has an inconsistency because the inode was not written before the crash. The directory change log entry does not contain enough information to fully reconstruct the lost inode. Missing from the entry are the inode fields

of access and modify times, permissions, and owner information. Even if this information were included in the log entry the roll-forward of missing creation operations is of questionable value because the contents of the file would still be lost.

Instead of rolling these lost creation operations forward, the operation is rolled backward by removing the name from the directory. This rollback can create problems when the operation being rolled back is a directory creation. Any further file or directory creations in this directory will also have to be undone. To prevent this cascading rollback, directory creations that would cause cascading are treated differently. Rather than rolling backwards, the directory create operation is rolled forward in a special directory in the root of the file system called ‘‘lost+found’’. Performing the create in this directory allows the further creations of files or directories to use this newly created directory. For example if the inode for the creation of the directory ‘‘dir’’ is lost and the file ‘‘dir/file’’ survives the crash, the file will appear in as ‘‘lost+found/dir/file’’ after a crash. The semantics of this use of ‘‘lost+found’’ is consistent with its use under traditional Unix file systems.

One additional task performed by the Sprite LFS crash recovery program is to deal with the delete-on-close semantics of Unix. In Unix, it is possible to unlink a file that is open. The semantics are that the directory entry disappears but the file stays around until it is closed. As required by the Sprite distributed file system, the recovery programs detects these files during roll forward and links them into the directory ‘‘lost+found’’. Two additional features are required to implement this feature. First, an extra bit in the inode map is used to track files that have been unlinked while still open. This bit is preserved with the inode map during checkpoints so it is available to the recovery program. Secondly, the recovery program sets this bit when the roll forward code finds an inode with a zero link count. This step allows the recover program detect any file unlinked-while-open since the last checkpoint.

4.5.1.4. End-of-log detection

The roll-forward algorithm assumes it can follow the log until it reaches the last completely written log region. This task is more complicated than it sounds because a partially written log region caused by a crash can mislead the roll forward code into mistakenly interpreting the previously written data at the end of the log as part of the current log. For example, consider a segment that is first written with a single large log region containing data blocks from single large file. The file is deleted so the segment is reused. This time the segment is written with several small log regions and a crash occurs after a few regions have been written. A problem can occur if the old contents of the large file looked enough like a log region that the roll forward code found more log regions than were actually written.

To avoid this problem, Sprite LFS organizes the log so that when a log region is written, its segment summary block will always point to a segment summary block known to have an older timestamp. The log following algorithm simply follows the segment summary block chain until it encounters a segment summary block with a timestamp less than the previous block. This rest of this section describes the organization of the log region chain and the assumptions required of the storage device in order to make it work.

To simplify the crash recovery system, Sprite LFS makes several assumptions about the disk storage device. The first assumption is that a crash while updating a disk sector will result in the sector containing either the old contents, the new contents, or a detectable error. This is a standard assumption made by many disk storage managers.

Another assumption is that putting the segment summary block at the end of a log region allows it to be used as an indicator that the entire log region was written. This means that the disk storage device must write the last sector of a multi-sector transfer last. This assumption fails on some disks. On such devices the assumption can be enforced by making the software explicitly write the last sector last.

Using these assumptions, Sprite LFS implements a combination of reserved blocks on disks and erasing of previously written data to implement the crash-proof chain of summary blocks. The last block in every segment is reserved to be a segment summary block. When the file system is created all these reserved blocks are initialized. Each time the segment is rewritten, the last disk sector transferred to the segment will overwrite this reserved block with a new summary block. Using this technique, a segment summary block that points to a new segment will always point to this reserved block. Until it is overwritten, it will be the end-of-log marker since its timestamp is old. Should a new log region be totally written to this segment, the reserved block will be overwritten and this log region becomes part of the recovery chain.

The technique of reserving the last block in the segment fails when multiple log regions are written to the same segment because the last block in the segment is not always the current end of the log. With multiple writes to a segment, a segment summary block in the middle of the segment may be the current end of log. In this case, adding a log region to the segment requires erasing the current end-of-log summary block after the new log region has been transferred. To efficiently implement erasing of the next summary block, Sprite LFS writes multiple log regions to a segment in reverse ‘‘disk order’’. The first log region written to a segment is placed in the last portion of the segment. This first log region’s summary block overwrites the reserved summary block location in the segment. The second log region written to the segment starts further back in the segment and ends with a segment summary block that just overwrites the segment summary block that was at the beginning of the first region. This pattern continues until the segment is totally filled in.

By writing the segment backwards, Sprite LFS can erase the next summary block by adding it to the beginning of the transfer. Figure 4-6 illustrates this operation of writing multiple log regions to a segment. Each write to disk first updates the next summary block and then writes the log region pointing at this summary block. The key feature of this backward fashion is that it allows the location of the next segment summary block to be written in the same transfer that writes the current log region. No additional disk request is needed to erase the block.

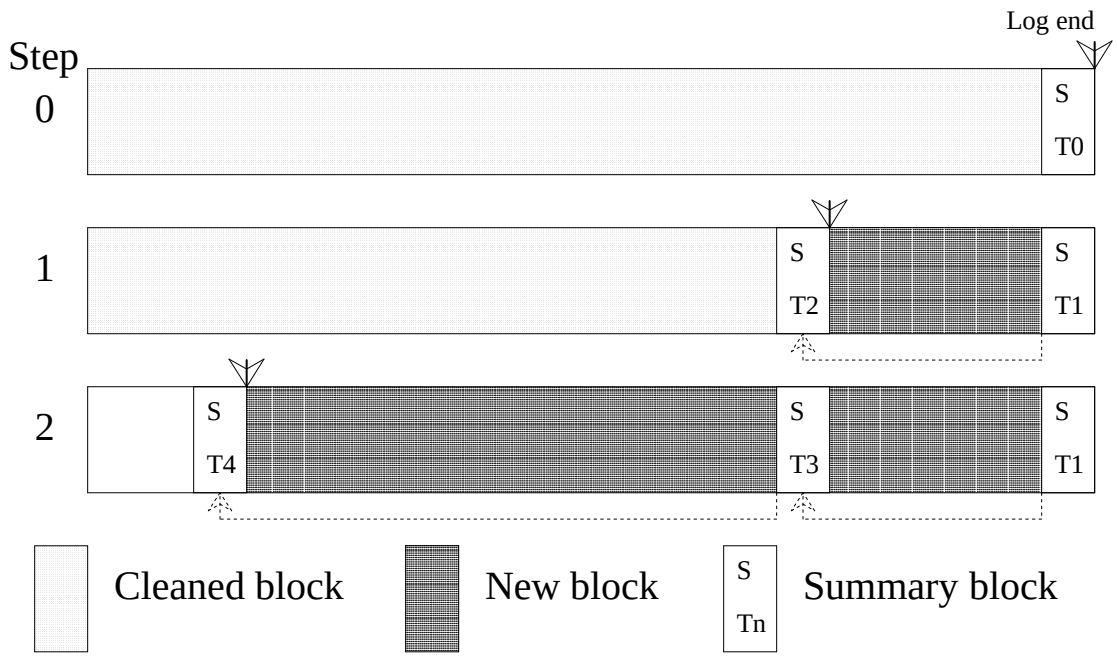

Figure 4-6 — Multiple log writes to same segment
This figure shows the disk transfers done when transferring multiple log regions to the same segment. The example shows the writing of two log regions to a previously cleaned segment. At Step 0 the segment has been cleaned. The only block with a known value is the reserved summary block at the end of the segment which is known to have an old timestamp; T0 in this example. Step 2 shows the placement of the first log region written. The write adds a new summary block for the end of log (timestamp T2), transfers the new data, and updates the reserved summary block as the last act. Once this transfer completes, the new end-of-log is the summary block T2. Step 3 shows the placement of the second log region. It writes a new summary block and as its last transfer overwrites the previous summary block marking the end of the log. Note that any incompletely written log region will not be found by the end-of-log detection algorithm, since the old end-of-log summary block will not have been overwritten.

Although writing the segment backward can be slower than writing it forwards, poor performance only occurs for very small log regions in which the overhead of skipping backward is large compared to the data transfer time. Such small log regions only occur when there are insufficient modifications to fill large log regions. During such times high write bandwidth is not important.

For this scheme to work, storage devices must guarantee that the last sector of a transfer will always be written last. This assumption holds for most disks in the Sprite environment but fails for some storage devices. Devices that fail require the storage manager to explicitly write the segment summary block once it knows the rest of the log region has been written.

4.5.2. Checkpoints

The final part of the Sprite LFS crash recovery design presented is the checkpoint mechanism. Its purpose is to establish a consistent image on disk so that no information written before the checkpoint need be scanned during roll-forward. Furthermore, a checkpoint should provide fast access to key metadata structures such as the inode map and segment usage table, so that file systems can be mounted quickly.

The Sprite LFS approach for fast mounting of file systems is to store the information necessary for the file system mount in a fixed known location called the *checkpoint region*. The checkpoint region allows the file system mount code to quickly restore the in-memory metadata structures needed to process storage requests. The checkpoint region also points at a consistent state of the file system that the roll-forward algorithm can start with.

The key contents of the checkpoint region are the addresses of the blocks of the inode map and segment usage table data structures. The inode map and segment usage table are broken into fixed-size blocks and written to disk in a log region. The addresses of these blocks are stored in two arrays, one for each structure. The arrays are indexed by block number and updated when blocks of the structure are written to disk. Mounting a Sprite LFS file system at boot is very fast since only these block indexes and not the metadata blocks themselves must be read in at startup. The metadata blocks are fetched upon demand using the index to locate them.

Besides the block pointers for the segment usage array and inode map, the checkpoint region contains other items needed to mount the file system. These include:

- The location of the last log region written. This is used to locate the end of the log, which becomes the starting location for the next log write.
- Information needed by the ‘‘df’’ command[42], a command that returns disk free space and other information about the file system. This information includes the number of free blocks on the disk, the number of allocated files, and the number of clean segments.
- A monotonically increasing timestamp that identifies the time when the checkpoint region is written. This is the same timestamp used in the segment summary block headers.
- Counters of interesting events that have occurred on the file system. The counters are used to produce some of the measurements presented in Chapter 6.

Although some of these data structures, such as the number of allocated files and number of free blocks, can be computed from the metadata structures, Sprite LFS stores them precomputed in the checkpoint region to avoid slowing the mount code with the recomputation of the values.

Sprite LFS implements checkpoint using a two-phase process. During the first phase it writes out all modified file system information to the log, including all modified file data and the blocks of the inode map and segment usage table. Once all modifications are written to the log, the second phase writes a new checkpoint region. After the second phase completes the checkpoint region will contain the pointers to the newest copy of inode map and segment usage table blocks.

4.5.2.1. Synchronization

Writing all in-memory modifications to the log as is done in the first pass of the checkpoint operation does not guarantee that the on-disk structures will be consistent. Operations that modify multiple objects such as file creation and deletion will cause the in-memory image to be inconsistent for short periods of time during the operation. Should the modifications be written during this period of inconsistency the on-disk format will be inconsistent. Furthermore, the disk can be left in an inconsistent state by additional

modification requests that arrive once the writing has started.

To ensure checkpoints are consistent, the checkpoint mechanism imposes extra synchronization during the writing of the modifications. Checkpoint operations are prevented from starting while directory operations are in progress and directory operations are blocked while checkpoints are in progress. This synchronization means that the directory operation log can be truncated after each checkpoint. All directories and inodes are consistent with each other.

Additional synchronization is also implemented to stop all modifications to the data in the file cache while the write back is in progress. This synchronization ensures that inconsistencies are not generated and the checkpoint will finish in a prompt fashion. Without this synchronization, an application could add new modifications to the file cache as fast as the checkpoint can write them out, preventing the checkpoint from finishing.

The current implementation of Sprite LFS does more synchronization than is strictly necessary during checkpoints. There are more complicated techniques[43] of getting consistent checkpoints without blocking all modifications. These techniques work by forgoing the synchronization and checkpointing the data to disk. The algorithms detect in operations that could cause inconsistencies during the writing and record these operations so that a consistent checkpoint can be generated after a crash. Because of the relative infrequency of checkpoints, the simpler approach was taken in Sprite LFS.

4.5.2.2. Checkpoint region update

Crash recovery in Sprite LFS requires that the system tolerate crashes that occur any time, including during the checkpoint operation. The append-only nature of the log means that data in the log is never updated in place. The Sprite LFS log deals with crashes by ignoring any partially-written log regions. The checkpoint region is located at a fixed location and updated in place so a crash during a checkpoint region update could leave the structure in an inconsistent state. In order to withstand crashes while updating the checkpoint region, Sprite LFS allocates two separate regions and alternates checkpoints between them. Each checkpoint region ends with a trailer record that records the current timestamp of the region and checksums the region's contents. This trailer record is used to detect checkpoints that were only partially written. By alternating between two regions and having the ability to detect corrupted checkpoints, Sprite LFS can always find a complete and consistent checkpoint after a crash.

To reduce the amount of data written to the checkpoint region, Sprite LFS contains another data structure called the *superblock*. The superblock contains the file system description parameters that do not change during normal operation. These parameters include the location and size of checkpoint regions, the locations of the segments on disk, and other parameters set at file system creation and never changed. The superblock is read when the file system is attached and is never written except when the disk is reformatted.

The Sprite LFS mount code first reads the superblock and then reads the two checkpoint regions. The newest complete region as indicated by the timestamp and the trailer is used to build the in-memory data structures. The entire attach operation uses only a few disk transfers.

Since Sprite LFS uses the checkpoint region for mounting the file system, the roll-forward part of crash the recovery is optional. It is safe to restart a file system after a crash by simply reading the latest checkpoint region. This effectively discards any data in the log that was written after the checkpoint. The result is instantaneous recovery with any data written since the last checkpoint being lost. The system was used in this mode with a 60 second checkpoint interval for an entire year before the crash recovery algorithm was installed. The only part of the crash recovery program missed was the collection of unreferenced files into the ‘‘lost+found’’ directory. Periodic cleanup of these files by a special program was necessary.

Checkpoints are initiated on a Sprite LFS file system under three circumstances. The first occurs when the file system is unmounted or the *sync* system call is issued to synchronize file system on disk. This checkpoint ensures that fast attach without data loss is possible after unmounting or normal shutdown of the system.

Checkpoints are also initiated at periodic intervals whose length is specified in the file system's superblock. The purpose of this checkpoint is to limit the maximum amount of roll-forward necessary after a crash. A short interval of 60 seconds means that at most 60 seconds worth of log traffic would have to be

scanned over during roll-forward after a crash. A longer interval results in less checkpoint overhead at the expense of a longer maximum crash recovery time. In hindsight, specifying the interval length in seconds was a mistake. Because of its closer correlation with recovery times, the amount of log traffic would have been a better choice.

The last event that causes a checkpoint is the completion of segment cleaning. Although the segment cleaning mechanism copies the files and index structures from a segment during cleaning, the segment summary blocks in the segment may still be needed by the crash recovery algorithm to follow the log during roll-forward. Segments can not be overwritten while they are still needed during crash recovery. Although it would have been possible to mark as clean the segments that were not needed during roll-forward, always doing a checkpoint before reuse eliminated the need to keep track of which segments were needed.

4.6. Implementation

The implementation of the design presented in this chapter is split between code running in the Sprite kernel and code run as user-level programs. The code that implements file system mounting, allocation, I/O, and segment cleaning is contained in the Sprite kernel under a special interface for disk storage managers. The code for crash recovery and new file system formatting runs as user-level programs.

Two disk storage managers are implemented in Sprite: Sprite LFS and a Unix FFS-like storage manager called the original Sprite file system (OFS). Having two disk storage managers implementing the same interface in the same kernel allows a comparison of Sprite LFS with more traditional designs. Using the object code size on the SPARC architecture as a measure of implementation complexity, Sprite LFS is slightly more complex than OFS. For the kernel resident code, the Sprite LFS implementation takes 38 kilobytes of object code and OFS takes 23 kilobytes. Most of the difference comes from the logging and synchronization code needed to support crash recovery code and more modular implementation of Sprite LFS.

Without the crash recovery and modular implementation, the code would be the same between the two storage managers. Sprite LFS has a very simple allocation algorithm and a complex segment cleaning mechanism while OFS has the complexity in its allocation algorithm and no cleaning mechanism.

Although the current Sprite LFS implementation is closely tied to the internal structure of the Sprite kernel, the design described is not limited to being implemented in Sprite. The vnode interface commonly found in Unix systems[44] provides a point at which the Sprite LFS design could be added to the Unix kernel.

4.7. Summary

This chapter presented design and implementation techniques for log-structured file systems. Two main design problems are identified: the index mechanisms used to locate files in the log and the techniques used to handle log wrap. The Sprite LFS solution to the file location problem was to keep the same inode and indirect block structure as Unix and to add a data structure called the inode map that. The inode map tracks the location of the inodes as they are written to the log. The Sprite LFS solution to the log wrap problem was to divide the disk into segments and use garbage collection to compact fragmented segments.

Also presented in this chapter was the fast crash recovery system of Sprite LFS. Using a combination of logging, checkpoints, and roll-forward, Sprite LFS allows fast recovery regardless of the size of the disk storage. Recovery time of a Sprite LFS file system depends on the amount of data written since the last checkpoint.

This chapter discussed the basic mechanisms of log-structured file systems and Sprite LFS. Sprite LFS supports the Unix file system abstraction while using a disk layout that is very different from traditional Unix storage managers. The mechanisms of Sprite LFS provide many tunable parameters and policies that can effect the performance but not the correctness of the systems. The next chapter, Chapter 5, describes the effect of different policies on the segment cleaning mechanisms. Chapter 6 follows with experience gained from using the Sprite LFS prototype.

CHAPTER 5

Sprite LFS cleaning policies

The previous chapter described the mechanisms for implementing segment cleaning in Sprite LFS. This chapter describes the policies that control the cleaning mechanisms. The cleaning mechanism is implemented by a three-step process that reads in the segments, identifies the live contents, and writes the live data blocks to segments in the log. The mechanism is controlled by five basic policy questions:

(1) What size segments should be used? The mechanism supports segments that are of any size greater than one block. For high-performance operation the segment size should be large enough so that whole segment transfers can amortize the seeks to the start of the segments. The policy question remaining is how large and what is the effect of segment size on performance.

(2) When should the segment cleaning mechanisms be invoked? The mechanism allows segment cleaning to happen at any time during file system operation. Some possible policy choices are for it to run continuously at low priority in the background, or only at night, or only when disk space is nearly exhausted.

(3) How many segments should be cleaned at a time? Cleaning many segments at a time provides an opportunity to reorganize data on disk; the more segments cleaned at once, the more opportunities to rearrange. The disadvantages of cleaning large numbers of segments at once are that it requires memory to hold the files being copied and it delays the availability of any clean segments for a longer period of time. Furthermore, cleaning too many segments increases the chances of cleaning segments whose files would otherwise be overwritten or deleted soon after the cleaning. In this case it would be less overhead to delay cleaning the segment until the space had already been freed.

(4) Which segments should be cleaned? The mechanism allows any segment to be cleaned. An obvious choice is those that are most heavily fragmented. Cleaning these segments reclaims the most free space with the least work. As will be shown in this chapter, this policy is not the best one. Other possibilities include using information about the contents or age of the data in segments to decide which ones to clean.

(5) How should the live blocks be grouped when they are written out? When files are read in during segment cleaning the file system can reorganize them when they are written back to disk. 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 try to reduce cleaning overhead by grouping blocks of similar age together into new segments.

To study these policies there needs to be a metric under which the policies can be evaluated and compared. The choice of metric decides which policies are good, and different metrics can lead to different good policies. The metric used in this dissertation is called *write cost* and is defined in Section 5.1. Write cost is a measure of how efficiently the file system uses the disk for writing data. Write cost was chosen to reflect the importance of write performance for future storage managers. Other possible metrics include request response time and request throughput.

Once a metric is chosen, the job of evaluating and selecting good choices for each policy is difficult. The large number of possible policy combinations makes it infeasible to examine all possibilities. Studying each policy individually does not work because the policies are not independent of each other. For example, the benefit from reorganizing the live blocks written out during cleaning depends on the number of segments cleaned at a time.

To further complicate the evaluation of these policies, good policies are workload dependent. A policy that causes large improvements in performance for one workload may hurt performance for another workload. For example, a policy that tries to exploit locality of reference in the workload will fail if there is no locality.

The approach I have taken in this chapter is to evaluate the above polices using simulation techniques and observations from the running prototype. The chapter starts with the definition of the write cost metric in Section 5.1. Section 5.2 describes the methodology used in the study and Section 5.3 describes the setup of the simulation study. Section 5.4 presents the results.

5.1. Write cost

The *write cost* metric is defined to be the average amount of time the disk is busy per byte of new data written, including all overheads. This can be viewed as a measure of how efficiently the file system can use a disk for writing. The write cost is expressed as a multiple of the time that would be required if there were no 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 can be written at the full disk bandwidth and there is no cleaning overhead. A write cost of 10 means that only one-tenth of the disk's maximum bandwidth is actually used for writing new data; the rest of the disk time is spent in seeks, rotational latency, or cleaning.

In a Sprite LFS file system with a large segment size, seek and rotational latencies are negligible for both writing and cleaning. 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 steady state the segment 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, or faction of live data, in the segments and $0 < u < 1$). This creates N*(1−*u*) segments of free space for new data. Thus

$$\text{write cost} = \frac{\text{total bytes read and written}}{\text{new data written}} = \frac{\text{read segs} + \text{write live} + \text{write new}}{\text{new data written}}$$

$$= \frac{N + N*u + N*(1-u)}{N*(1-u)} = \frac{2}{1-u}$$

The above formula uses 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. If a segment to be cleaned has no live blocks ($u = 0$) then it need not be read at all and the write cost is 1.0. So the write cost of Sprite LFS is

$$\text{write cost} = \begin{cases} \infty & \text{if } u=1 \\ \dfrac{2}{1-u} & \text{if } 0<u<1 \\ 1.0 & \text{if } u=0 \end{cases} \qquad (1)$$

The write cost metric for traditional Unix file systems such as the Unix FFS is not as easy to estimate as Sprite LFS. These file systems do not have cleaning costs. Instead the overhead comes from time spent waiting for seeks or rotational delays on every I/O request. The size of this overhead depends on the workload and how well the disk space allocation algorithm is working. In the best case analysis with the allocation algorithm working perfectly and a workload consisting of large sequential accesses the write cost of the Unix FFS is between 0.5 and 1.0[†]. This occurs when large files are allocated sequentially on disk and the files are written sequentially. Such transfers proceed with little seek or rotational delays. For small-file

[†]As originally implemented, Unix FFS allocation policy would allocate a file to every other block on a track. This type of allocation is called skip-sector allocation. Skip-sector allocation allows for CPU and disk controller overhead between block read requests. Without skip-sector allocation sequential transfers to a file would delay too much between blocks and the next block would rotate past the disk head before the transfer was started. The cost of skip-sector allocation is that only half the disk bandwidth can be used. Simple clustering of sequential file requests into larger transfer units allows SunOS (a Unix FFS derivative file system) to achieve a write cost of 1.0[20].

workloads, Unix FFS does substationally worse. Measurements show small-file workloads utilize at most 5-10% of the disk bandwidth, for a write cost of 10-20 (see [45] and Figure 6-1 in Section 6.2.1 for specific measurements).

If Unix FFS were enhanced with logging, delayed writes, and disk request sorting as described in Chapter 3, the write cost of a small-file workload could probably be improved to about 25% of the bandwidth or a write cost of 4. This enhanced file system is referred to as ‘‘Unix FFS improved’’. Figure 5-1 graphs the write cost for small file workloads as a function of *u* for Sprite LFS and compares to Unix FFS and Unix FFS improved. This figure suggests that the segments cleaned must have a utilization of less than 0.8 in order for a log-structured file system to have a better write cost than current Unix FFS; the utilization must be less than 0.5 to beat Unix FFS improved.

The key to achieving a superior write cost for Sprite LFS is to make *u*, the fraction of the segment alive when a segment is cleaned, as small as possible. One way to guarantee that *u* is small is to allow only a small fraction of the disk capacity to be occupied. If only 10% of the disk blocks of a Sprite LFS file

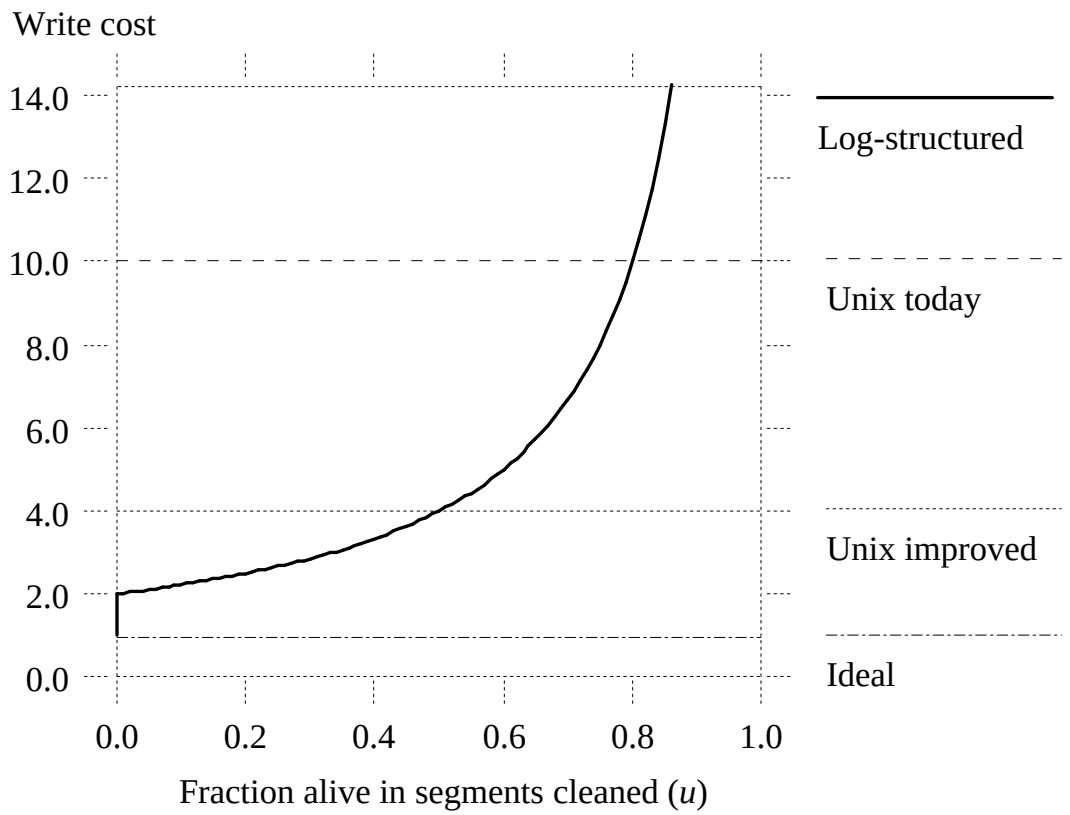

Figure 5-1 — Write cost as a function of *u*
In a log-structured file system, the write cost depends strongly on the utilization of the segments that are cleaned. The more live data in segments cleaned the more disk bandwidth that is needed for cleaning and not available for writing new data. The figure also shows three reference points: ‘‘Unix today’’, which represents Unix FFS today, ‘‘Unix improved’’, which estimates the best performance possible in an improved Unix FFS, and ‘‘Ideal’’, which shows the perfect write cost on 1.0. Write cost for Unix FFS is not sensitive to the amount of disk space in use within its operating range.

system contain live data, the cleaning algorithm can always find segments to clean with u less than or equal to 0.1. This type of system would have high performance but would also have a high cost per byte stored.

Sprite LFS provides a cost-performance tradeoff: if disk space is underutilized, higher performance can be achieved but at a high cost per usable byte; if disk capacity utilization is increased, storage costs are reduced but so is performance. Such a tradeoff between performance and space utilization is not unique to log-structured file systems. For example, Unix FFS only allows 90% of the disk space to be occupied by files. The remaining 10% is kept free to allow the space allocation algorithm to operate efficiently.

Although limiting the fraction of the disk capacity available for live blocks would guarantee a low write cost in LFS, it leads to cost-performance characteristics that are unacceptable in cost-sensitive environments such as office/engineering. Environments that typically run at 80% disk capacity utilization will need to double the available disk space to have a guarantee that Sprite LFS will outperform Unix FFS improved. Fortunately, the discussion so far is based on the overly pessimistic assumption that the fraction alive in segments cleaned (u) is equal to the disk capacity utilization. Clearly the average utilization over all segments on disk must equal the disk capacity utilization, but the write cost depends on only those segments that are cleaned. Since Sprite LFS gets to choose which segments are cleaned it can achieve a lower write cost than the overall disk capacity utilization would indicate.

For Sprite LFS to achieve superior write cost even with high disk capacity utilization, segments must divide themselves into two groups. The high disk capacity utilization requires that most segments are full or nearly full with live files. There also need to be segments with few if any live files for the segment cleaner to operate on. For example, a Sprite LFS disk with 90% of the segments being 90% alive and 10% of the segments being 10% alive means that the overall disk capacity utilization will be 82% while the (u) value can be as small as 0.1. The write cost of this system can be 2.2 even though the guarantee based on the disk capacity is 11.1. This write cost of 2.2 is much better than the value of 4 provided by the Unix improved.

The utilization of the segments on disk can be viewed as a probabilistic distribution. What is desired is a a bimodal segment utilization distribution where most of the segments are nearly full and a few are empty or nearly empty. With such a distribution the average segment utilization (the disk capacity utilization) will be high while the u seen by the segment cleaner will be small. This distribution is controlled by the workload and segment cleaning policies. It is important that the cleaning policy drive the disk into such a bimodal distribution in Sprite LFS under normal workloads.

5.2. Methodology

For a Sprite LFS file system, the write cost is a function of u which in turn is dependent on the workload and the segment cleaning policies. Two different methods were used to study the effects of the different policy choices on u. The first method was to use a simple simulator of pseudo-random workloads to study the behavior of different cleaning policies. The second approach was to use feedback from experience with benchmarks and long term uses of Sprite LFS file systems to determine effective policies.

The two methods of study complemented each other in the insight they provided. The simulation method permitted the examination of a wide range of workloads to determine the effect of alternative policies. The experience with the Sprite LFS prototype allows much less flexibility in policy measurements but provides more ‘‘real’’ workloads to be studied over long periods of time. This chapter does simulations, next chapter does measurements.

5.3. Simulation study

The simulation study examined the effects on write cost of different cleaning policies under workloads of randomly generated accesses to small files. The simulator was implemented as a simple C program that modeled the Sprite LFS segment writing and cleaning mechanisms. The output of the simulator was the average fraction alive in the segments cleaned, u, over the life of the simulation.

The study was conducted to determine the effect of each of the polices listed at the beginning of this chapter. Effects of changes in each policy were examined under a range of workloads. The insight gained from this study is reported in Section 5.4. The rest of this sub-section describes the simulator model and implementation in detail.

5.3.1. Simulator implementation

The simulator implementation was very simple. The entire program was only 1100 lines of C code (10 kilobytes of SPARC object code) which is less than 25% of the LFS kernel source size. Being simple allowed the simulator to run fast so that many combinations of workloads and policies could be examined. The simple implementation also reduced the chances that bugs in the simulator would cause incorrect results.

The simple simulator implementation also required the simulation to make several simplifying assumptions about the Sprite LFS mechanism, the disks, and the workloads. The file system was assumed to contain a fixed number of 4-kilobyte files. Only file data were written to disk. Metadata structures such as checkpoints, the inode map, segment usage table, segment summary blocks, directories, and inodes are never written to the disk. All metadata structures and files are assumed to be kept in memory so read requests never go to disk except during the reading stage of segment cleaning. The disk was assumed to behave according to the assumptions made for formula (1).

The simulator starts by placing all the files in segments with as many files as possible packed per segment. Segments that contain no files are marked as clean. At each step during the simulation, a file is selected using a pseudo-random access pattern. This file is overwritten; the previous copy is deleted and a new copy is written to the front of the log. If no clean segments are available to write the log, the segment cleaning mechanism of Sprite LFS is simulated to generate segments in which new data can be written. Section 5.3.2 describes the pseudo-random access patterns in detail while Section 5.3.3 describes the policy inputs that control the cleaning mechanism.

The simulation runs in a loop overwriting one file at each step with occasional pauses for running the segment cleaner. Counts are kept of the number of files written, the number of segments read during cleaning, and the number of live files copied out of segments being cleaned. From these counts the value of u and the write cost of the system can be calculated. By examining the state of the segments when segment cleaning starts, the simulator estimates the distribution function of the segment utilization.

The simulations were allowed to run until the average u values stabilized over a large number of segment cleanings. Typical runs simulated the writing of between 5 million and 800 million files. At the current update rate of the Sprite file systems, the second number of writes would take many years.

5.3.2. Workload generator

The workload generator of the simulator was controlled by two parameters: the disk capacity utilization and the file access pattern generator. The disk capacity utilization was specified as the fraction of the available space occupied by live files. By varying this parameter, the tradeoffs between performance and disk capacity utilization of a Sprite LFS can be evaluated. One way of viewing the capacity utilization is a specification of how much disk space needs to be purchased for the constant amount of file data. The curves in Section 5.4 that plot write cost versus capacity utilization clearly show this cost-performance tradeoff.

The second workload parameter for the simulator is the access pattern used to select files to be overwritten. Files are named with consecutive integers starting at zero and selection is done using a pseudo-random number generator to generate the sequence of file accesses. Three different random number generators were used for file selection in this study:

Uniform Each file has equal likelihood of being selected for writing at each point in time. The uniform generator provides an extreme case with no locality of reference.

Hot-and-cold Files are divided into two groups. One group is called *hot* because its files are are selected more frequently than those in the other group called *cold*. Two parameters are used to describe this distribution. The first parameter is the fraction of the files that are in the hot group. The second parameter specifies the fraction of the references going to the hot group. Within groups each file is equally likely to be selected. This approach simulates a simple form of locality. For example, tests were run with 90% of the references going to 10% of the files with the remaining 10% going to the other 90% of the files.

Exponential Files are selected using random numbers generated with an exponential distribution with a specified mean. Because of the long tail on the exponential distribution, this distribution allows for a much greater locality than the hot-and-cold distribution. The exponential

distribution present in this chapter has a mean of 0.02172. This mean causes 90% of the references to go to 2% of the data. In the results section this distribution is called ‘‘Expon90->2’’.

The distributions used to generate these workloads do not reflect common file system usage patterns. Instead the distributions were chosen to provide controlled conditions under which I could examine the effects of randomness and locality on the write cost of a Sprite LFS file system. The distributions used here are much harsher than those generated by the Sprite user community and presented in Chapter 6.

5.3.3. Cleaning policies

The final parameters for the simulations selected the segment cleaning policy. The simulator allowed for control of the following policies:

(1) The size and number of segments in the file system. The segment size ranged in values from 24 kilobytes up to 8 megabytes. Unless otherwise specified a segment size of 2 megabytes (512 files) was used. The number of segments (size of the simulated disk) was adjusted so that there were always 200 megabytes (524288 files) of data. Section 5.5.3 describes the effects of this policy.

(2) The number of live files read at a time during cleaning. The reading step of the cleaning mechanism continues until it has read this number of live files into memory to be combined and written out during the write-back step. The motivation behind the number is that Sprite LFS reserves space in the cache for cleaning. This number specifies the maximum amount of space reserved in the cache. If not specified then 10 megabytes (2560 files) were read during a cleaning operation. Section 5.5.2 examines the effect of this policy.

(3) The policy used to select the segments to be cleaned. This policy is invoked when cleaning is started and is used to select which segments are to be read in and combined. Two policies were implemented: the greedy and cost-benefit policy. The *greedy policy* chooses the segments with the least number of active bytes according to the segment usage table. The cost-benefit policy uses a function of the age and amount of data in a segment. These polices are described in Section 5.4. Unless specified the greedy policy is used.

(4) The policy specifying the order in which live data is written back to disk during cleaning. The two choices were to combine the live files in the order in which they were read from segments during cleaning or to group the files by last modify time. Section 5.5 examines this policy and the default policy is to use the order in which they were read-in.

Because of the large number of possible policy and workload combinations it was infeasible to study all combinations. The approach taken for this thesis was to study the effects of varying each cleaning policy individually. The goal was not to use the simulator to predict performance but to understand the reason behind the changes in write cost due to varying the policies. To further aid in understanding, I modified the simulator to graphically represent the state of the disk. As the simulator ran it produced an animated view of segments and allowed the effects of policy changes to be visualized.

5.4. Simulation results

The initial simulations were done to evaluate how much better the actual write cost would be over the pessimistic prediction of formula (1). The cleaning policies for the simulation were those proposed for the initial design of Sprite LFS. Segment selection during cleaning was done using the greedy policy, which always chose the least-utilized segments to clean. The (mistaken) rationale was that doing the minimum amount of work during each cleaning will result in the lowest write cost. 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. The segment size was 2 megabytes and each cleaning operation proceeded until 2560 live files were read in. Figure 5-2 superimposes the results from two sets of simulations onto the write cost curves of Figure 5-1. In the ‘‘LFS uniform’’ simulations, the uniform access pattern to files was used. The curve labeled ‘‘LFS hot-and-cold’’ used the hot-and-cold access pattern with 90% of the references going to 10% of the files.

The first observation from Figure 5-2 is that the write cost formula (1) is indeed pessimistic. Even with uniform random access patterns, variations in segment utilization allow for a substantially lower write cost than would be predicted from the overall disk capacity utilization and formula (1). For example, at

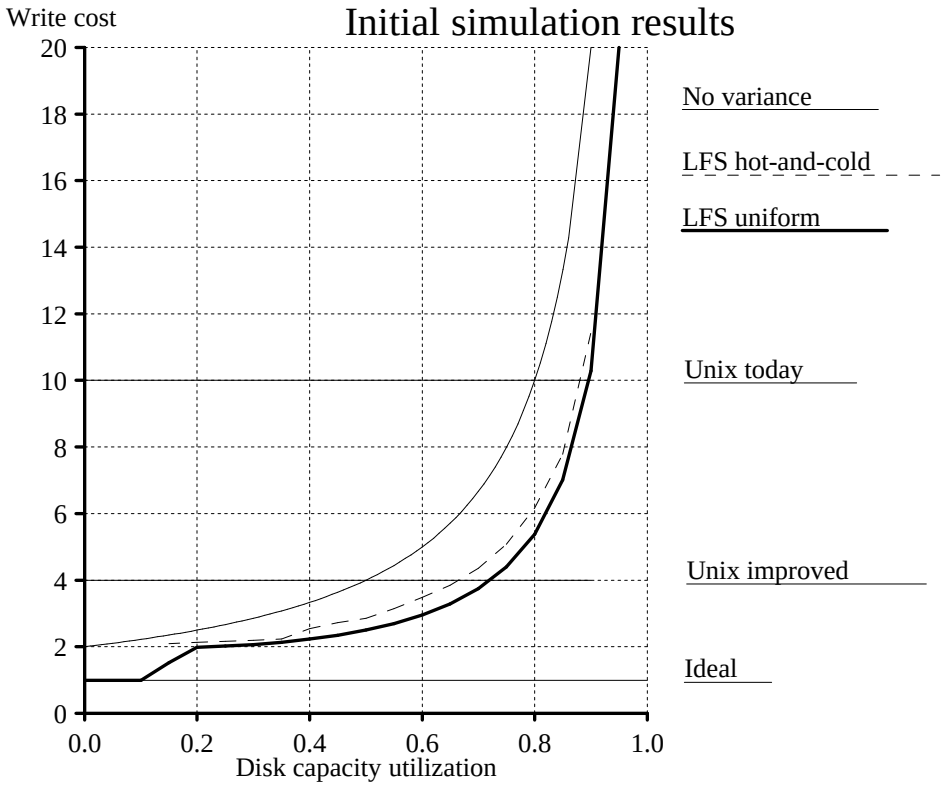

Figure 5-2 — Initial simulation results
The lines labeled ‘‘Unix today’’, ‘‘Unix improved’’, and ‘‘Ideal’’ are reproduced from Figure 5-1 for comparison. The curve labeled ‘‘No variance’’ shows the write cost that would occur if all segments always had exactly the same utilization. This is the same as the ‘‘Log-structured’’ curve in Figure 5-1. 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 90% of the references going to 10% of the files. Both simulation curves use the default cleaning policies. The x-axis is overall disk capacity utilization, which is not necessarily the same as the utilization of the segments being cleaned. Two observations can be made from this graph. The first is that chosing segments to clean that have utilization less than average can result in a write cost well below that predicted by the average utilization. Secondly, a workload with locality of reference did worse than one without locality.

75% overall disk capacity utilization, the segments cleaned have an average utilization of only 55%, which results in a write cost of 4.4 rather than the 8 suggested by Figure 5-1.

A second observation is that at overall disk capacity utilizations under 20% the write cost for LFS uniform drops below 2.0. The reason for this is that some of the cleaned segments have no live blocks at all and hence don’t need to be read in.

This second observation can also be derived analytically. For the uniform access pattern, the fraction alive in a segment is an easily computed function of the number of writes to files. The probability of a file not being overwritten on each step is $(1-\frac{1}{F})$ where F is the number of files. For a disk with capacity U, (0<U<1), it should be possible to write $\frac{F}{U}$ files before having to clean a segment. $\frac{F}{U}$ is also the size of the disk specified in number of files. The fraction alive in a segment is therefore $(1-\frac{1}{F})^{\frac{F}{U}}$ which is approxi-

mately $e^{-\frac{1}{U}}$ for large F. From the simulation with a segment size of 512 files, the number of alive files is $512*e^{-\frac{1}{U}}$ which drops below one as U moves from .2 to .15.

Using a similar argument about the probability that files are overwritten leads us to expect that locality in the file reference pattern results in a lower write cost than no locality. Consider what happens when a segment is written under the hot-and-cold workload with 90% of the references going to 10% of the files. On average 90% of the files in a segment will be from the frequently accessed (hot) set of the file and the other 10% will be cold files. With the high degree of locality, the hot files of a segment will be quickly re-referenced causing their space in the segment to be freed. If cleaning can be delayed long enough, only the cold files that occupy 10% of the segment will be left. With *u* value of near 0.1 the write cost of the system should be near 2.2 even for high disk capacity utilizations.

Surprisingly, the ‘‘LFS hot-and-cold’’ curve in Figure 5-2 has a *worse* write cost than the ‘‘LFS uniform’’ curve that has no locality. This non-intuitive behavior can be further seen in Figure 5-3 which examines several different access distributions with varying amounts of locality. Increasing the degree of locality in the hot-and-cold distribution did not result in improvements in write cost. Even with 90% of the references going to as little as 1% of the files the write cost is still worse than uniform access pattern with no locality. The much greater locality in the exponential distributions helped but still not as much as expected. With 90% of the references going to 2% of the data, 90% of each segment written should contain files from a set of 10500 files. These files should be quickly overwritten causing the faction alive in the segment to be close to 0.1.

The non-intuitive effect of locality implies that newly-written segments are being cleaned before the hot files die. This means that there are not enough clean segments around to be written before the newly-written segments are reused. The question to be answered is where has all the free space (clean segments) on the disk gone? The answer is that the segments that contain slowly-changing cold files have become fragmented and contain the free space needed for writing the hot files. The solution to the problem is to clean these cold segments before they get fragmented so the free space will be available for the log.

Figure 5-4 shows the distribution of the segment utilizations at the times during the simulation when the cleaner started, with the disk utilization at 75%. The curves are computed by recording the segment utilizations in the simulator each time the cleaner is invoked.

All the curves in Figure 5-4 have the average segment containing 75% live files and 25% free space. The difference between the Uniform case and the Hot-and-cold cases is that the Hot-and-cold has fewer segments with 80% to 100% utilization, and also fewer segment with less than 60% utilization. The more locality caused fewer ‘‘full’’ segments and more segments that are only 60% to 80% filled. Since most (90% or 99%) of the data on disk is in cold files, these segments must contain the slowly changing cold files. The free space on the disk is in segments containing around 60% cold files.

The shape of the curves in Figure 5-4 is due to the greedy segment selection policy. Under the greedy policy, a segment doesn’t get cleaned until it becomes the least utilized of all segments. The result is that a *cleaning cliff* occurs in the segment utilization distribution. Any segment whose utilization reaches the cleaning cliff is immediately selected to be cleaned. The segment’s free space is reclaimed and it is rewritten to be full of data.

The cleaning cliff of the greedy policy combined with different speed of fragmentation of segments causes the worse write cost with locality. With the hot-and-cold access patterns, segment utilizations are decreasing at widely varying rates. Recently written segments with many hot files have data overwritten rapidly so their utilizations drop quickly. Segments full of cold files have their utilizations dropping very slowly. The cleaning cliff does the wrong thing for both types of segments. For segments containing hot files, the cleaning cliff means they are cleaned too soon, before the hot files have a chance to be overwritten. For segments containing cold files, they are allowed to become too fragmented before they reach the cleaning cliff. This can be seen in Figure 5-4 where 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. With the cold segments holding the free space, the recently written segments containing mostly hot files must be cleaned to reclaim space before the files have a chance to be overwritten. The failure of the greedy segment selection policy suggests that a policy needs more information than just the utilization of

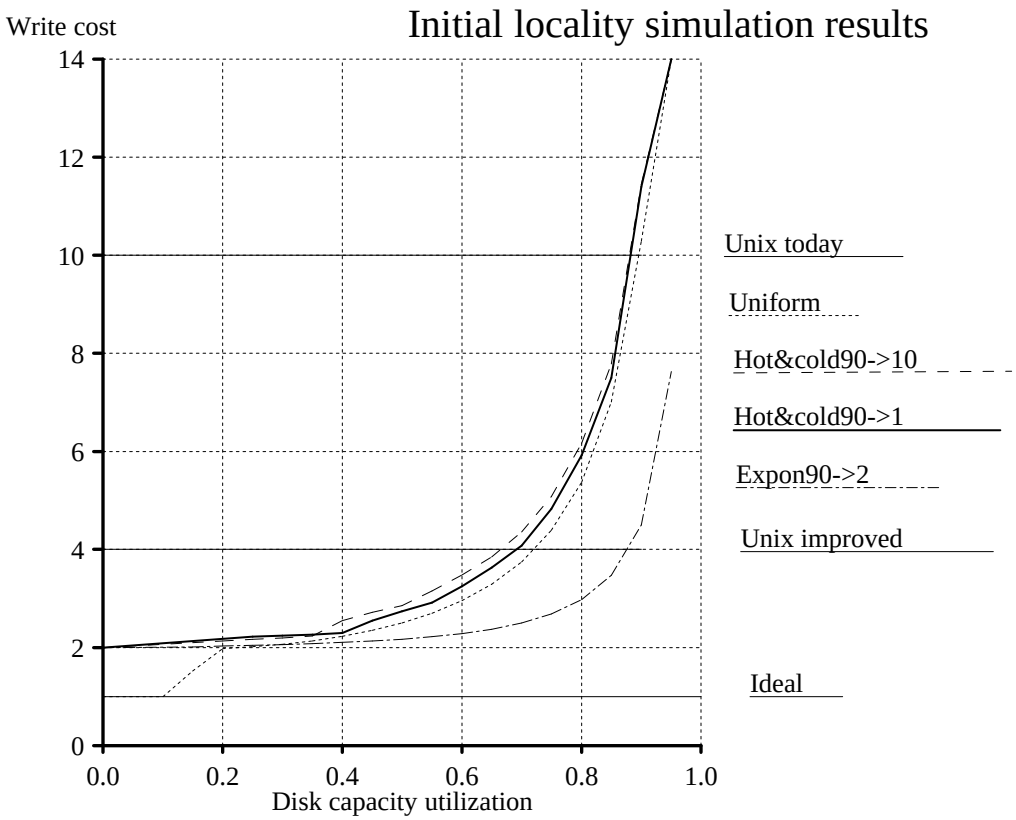

Figure 5-3 — Effect of locality on write cost
This figure shows the write cost of access patterns with differing degrees of locality. The ‘‘Hot&cold90->10’’ curve represents a log-structured file system using the hot-and-cold access pattern with 90% of the references going to 10% of the files. The ‘‘Hot&cold90->1’’ curve has even greater locality with 70% of the references going to only 1% of the files. The curves labeled ‘‘Expon90->2’’ use an exponential distribution workload generation that has 90% of the references going to 2% of the data. This access pattern has much greater locality than the Hot-and-cold access patterns. The curves labeled ‘‘Unix today’’ , ‘‘Unix improved’’, ‘‘Ideal’’, and ‘‘Uniform’’ are reproduced from Figure 5-2 for comparison. Except for the case of extreme locality of reference, locality did not result in a lower write cost.

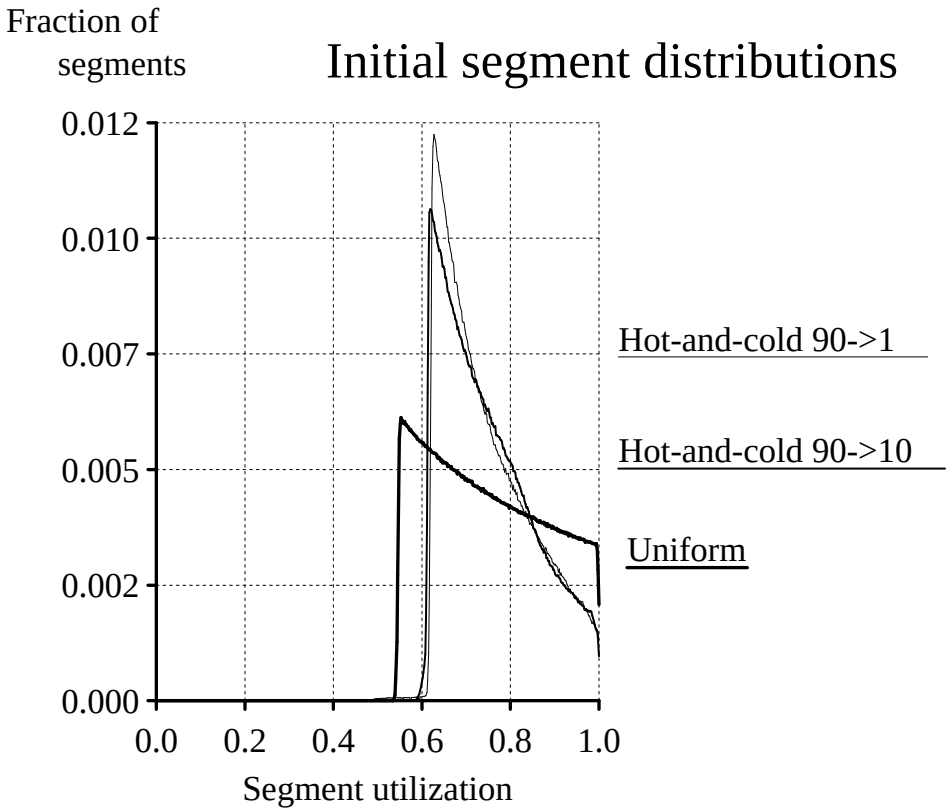

Figure 5-4 — Segment utilization distribution with greedy cleaner
This figure shows the distribution of disk segment utilizations during the simulations. The distribution is computed by measuring the utilizations of all segments on the disk at the points during the simulation when segment cleaning is initiated. The distribution shows the utilizations of the segments available to the cleaning algorithm. Each of the distributions corresponds to an overall disk capacity utilization of 75%. The ‘‘Uniform’’ curve corresponds to ‘‘LFS uniform’’ in Figure 5-2 and ‘‘Hot-and-cold 90->10’’ corresponds to ‘‘LFS hot-and-cold’’ in Figure 5-2. The ‘‘LFS hot-and-cold 90->1’’ curve uses the hot-and-cold access pattern with even greater locality; 90% of the references going to 1% of the data. 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.

each segment to make choices leading to low write cost.

What is needed is some indicator of the stability of the data in the segment. The segment selection policy needs to treat the cold stable segments differently than the hot rapidly changing segments. One way to view the selection is that free space in a cold segment is more valuable than free space in a hot segment. Once a cold segment has been cleaned, it will take a long time before its files are deleted and free space re-accumulates. Said another way, once the system reclaims the free blocks from a segment with cold data it will get to ‘‘keep’’ them a long time before the cold data becomes fragmented and ‘‘takes them back again.’’ In contrast, it is less beneficial to clean a hot segment because the data will probably die quickly and the 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.

The value of a segment’s free space is based on the stability of the data in the segment. Unfortunately, the stability cannot be predicted without knowing future access patterns. Using an assumption that older data is less likely to be overwritten than younger data, the stability can be estimated by the age of data.

To test this theory I 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 free space that will be reclaimed and the amount of time the space is likely to stay free. The amount of free space is just $1-u$, where u is the utilization of the segment. I used the most recent modify time of any block in the segment (i.e. the age of the youngest file) as an estimate of how long the space is likely to stay free. The benefit of cleaning is the space-time product formed by multiplying these two components. 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 results in:

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

This policy is called the *cost-benefit* policy; it allows cold segments to be cleaned at a much higher utilization than hot segments.

Using the age of the youngest file in the segment to estimate the stability of the data assumes that the contents of the segments are about the same age. To increase the accuracy of this assumption the files that are read in during cleaning are grouped by modify time when they are written-out. This allows the cleaner to segregate the hot and cold files into different segments. The effect of this age-sorting is discussed in Section 5.5.1.

I re-ran the simulations under the hot-and-cold access pattern with the cost-benefit policy and age-sorting on the live data.

I 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 5-5, the cost-benefit policy produced a radically different distribution of segments than the greedy policy. Rather than having a single cleaning cliff, the cost-benefit policy establishes a cleaning cliff for each of the two locality groups (hot and cold). The cleaning policy cleans cold segments at about 75% utilization but waits until hot segments reach a utilization of about 15% before cleaning them. The result is that hot segments are given enough time for most of their files die before they are cleaned while cold segments are not given the opportunity to become heavily fragmented.

The cost-benefit policy results in a bimodal segment utilization distribution of the sort discussed in Section 5.1. Most segments are either full of live files or only contain 20% of 40% live files.. Since 90% of the writes are to hot files, most of the segments cleaned are hot. The effect on the write cost can be seen in Figure 5-6. 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 FFS even at relatively high disk capacity utilizations. Because of this performance benefit, Sprite LFS was implemented using the cost-benefit selection policy rather than the greedy policy as specified in the initial design.

5.5. Other cleaning policies

The simulation study focused on the segment selection algorithm because it appeared to have the largest impact on the write cost. Due to lack of time, the other cleaning policies were not addressed as methodically as the segment selection algorithm. This section discusses some other policy issues, gives a few simulation results, and makes some suggestions for future studies.

5.5.1. File reorganization during cleaning

As mentioned in the previous section, files being cleaned are sorted by last modify time before being written out during cleaning. This is the policy used by Sprite LFS. It is implemented by sorting the files using the time of last modification in the file's inode. The sorted list determines the order that files are placed in segments so that similarly aged files are written back to the same segment. Figure 5-7 shows the improvement between no sorting of data and modify-time sorting. If no sorting is done the data is combined in the order that it is read from segments on disk during cleaning. The benefits of keeping the age of the data in segments homogeneous is most noticeable for high capacity utilizations. For example, age sorting reduced the write cost of the Hot-and-cold distribution at 95% disk utilization from 16.6 to 11.8. For capacity utilizations below 70% the sorting provided little benefit.

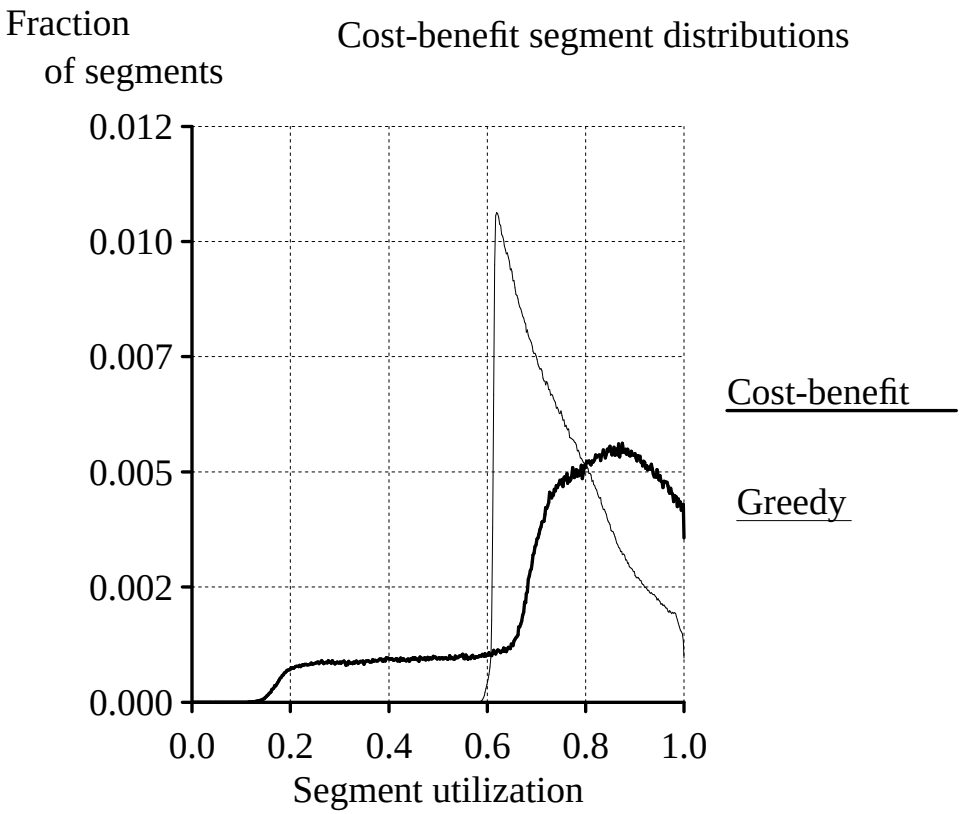

Figure 5-5 — Segment utilization distribution with cost-benefit policy
This figure shows the distribution of segment utilizations from the simulation of a hot-and-cold access pattern with 75% overall disk capacity utilization. The ‘‘Cost-benefit’’ curve shows the segment distribution occurring when the cost-benefit policy is used to select segments to clean and live blocks are grouped by age before being re-written. For comparison, the distribution produced by the greedy method selection policy is shown by the ‘‘Greedy’’ curve reproduced from Figure 5-4. The cost-benefit policy produced a segment distribution that contained segments with lower utilization than those produced by the greedy policy. This results in a lower write cost for the cost-benefit policy.

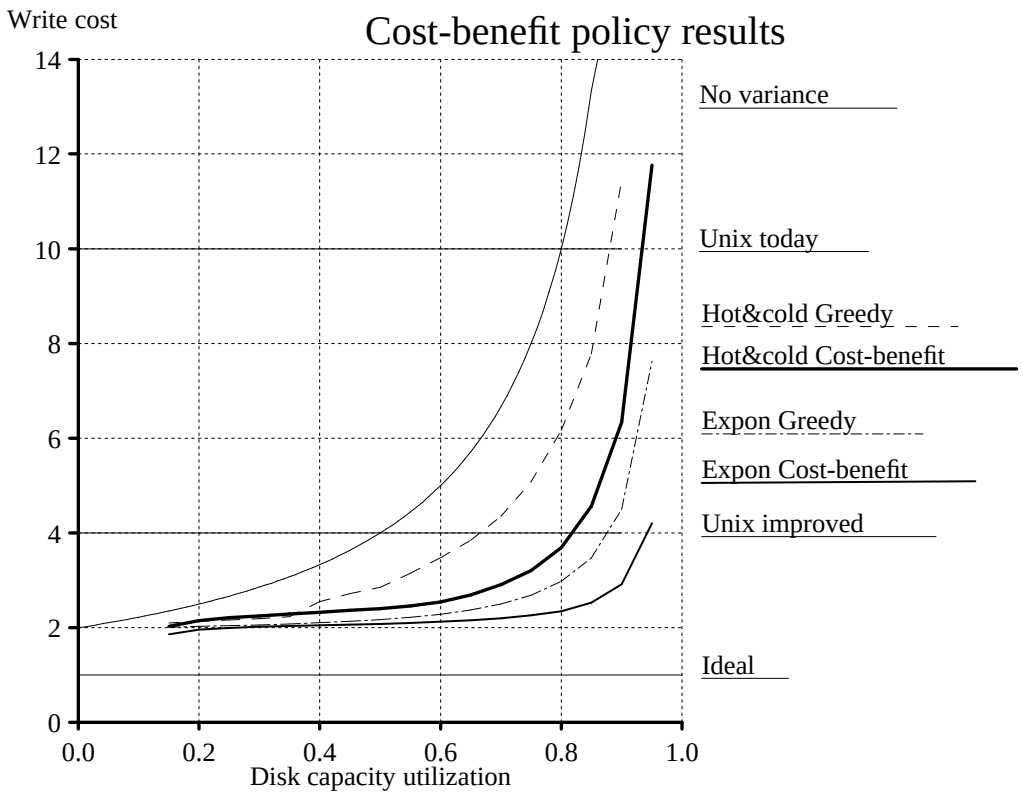

Figure 5-6 — Write cost with cost-benefit policy
This graph compares the write cost of the greedy policy with that of the cost-benefit policy for two access patterns: hot-and-cold access pattern with 90% of the references going to 10% of the data (Hot&cold) and exponential access pattern with 90% of the references going to 2% of the data (Expon). For reference the curves ‘‘No variance’’, ‘‘Unix today’’, ‘‘Unix improved’’, and ‘‘Ideal’’ are reproduced from Figure 5-2. For both the hot-and-cold and exponential access patterns, the cost-benefit policy is substantially better than the greedy policy for disk capacity utilizations above 60%. With the cost-benefit selection policy and the hot-and-cold access pattern (curve ‘‘Hot&cold Cost-benefit’’), the log-structured file system can operate at disk capacity utilizations of 80% and still have a lower write cost than the improved Unix file system. The exponential curve (‘‘Expon Cost-benefit’’) has a lower write cost than the improved Unix file for all disk capacity utilizations in which the Unix file system currently operates.

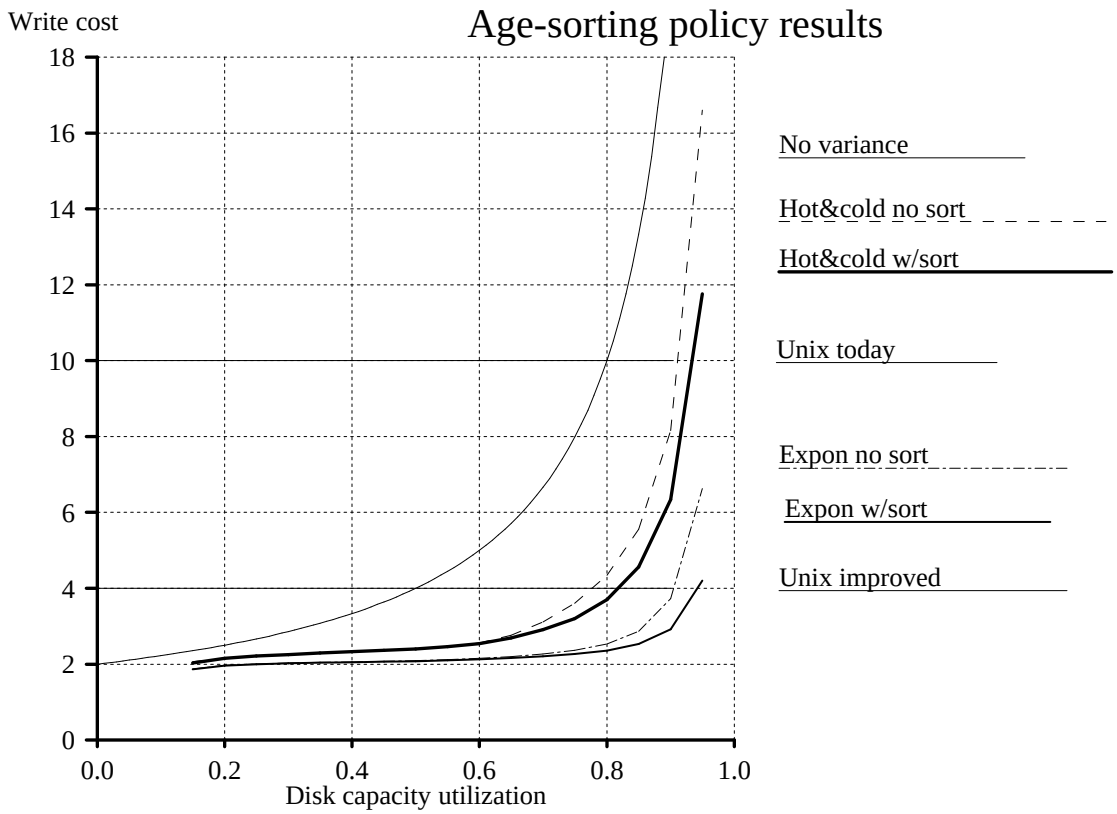

Figure 5-7 — Benefits of age sorting
This figure shows the effect of the age-sorting reorganization for the simulation results shown in Figure 5-6. The curves labeled ‘‘w/sort’’ correspond to the ‘‘Cost-Benefit’’ curves in Figure 5-6 while the curves labeled ‘‘no sort’’ represent the same access pattern and selection policy with no reorganization of the data during cleaning. For both the hot-and-cold and exponential access patterns, moderate improvements in write cost occur when the system is running at high disk capacity utilizations.

Another cleaning policy question is how much data should be cleaned at a time. One would expect improved performance by reorganizing more data at the cost of more memory space needed during cleaning. Figure 5-8 shows the write cost as a function of the amount of data cleaned at a time. As long as enough is cleaned at once so that the segment selection and age-sorting mechanism can function, the write cost is insensitive to the number cleaned. In fact, the write cost actually increased when large numbers of files are cleaned. The increase is caused by segments being cleaned too soon in the search for data to clean.

Figure 5-8 indicates that large amounts of memory are not needed during cleaning. Sprite LFS reserved 10 megabytes (about 10% of the file cache on the main file servers) for cleaning operations. When cleaning is not active this space can be used for caching files.

Rather than enhancing write performance, the reorganization during cleaning can implement a policy that attempts to improve further read performance. Recent studies[46] have demonstrated the benefits of reorganizing data on disk to improve read performance. Some reorganization is already done in Sprite LFS. Because a file is written out contiguously after cleaning, a file that is spread out on the disk will be pulled backed together if the segments containing the parts of the file are cleaned together. Files that become spread among multiple segments can be pulled back together during cleaning. One possible addition to the policy would be to combine files into segments based on the naming hierarchy so that files near each other in the naming hierarchy will be written to the same segment. For example files could be grouped together based on the directory in which they reside. The simulation study described in this dissertation did not address the issue of read performance at all.

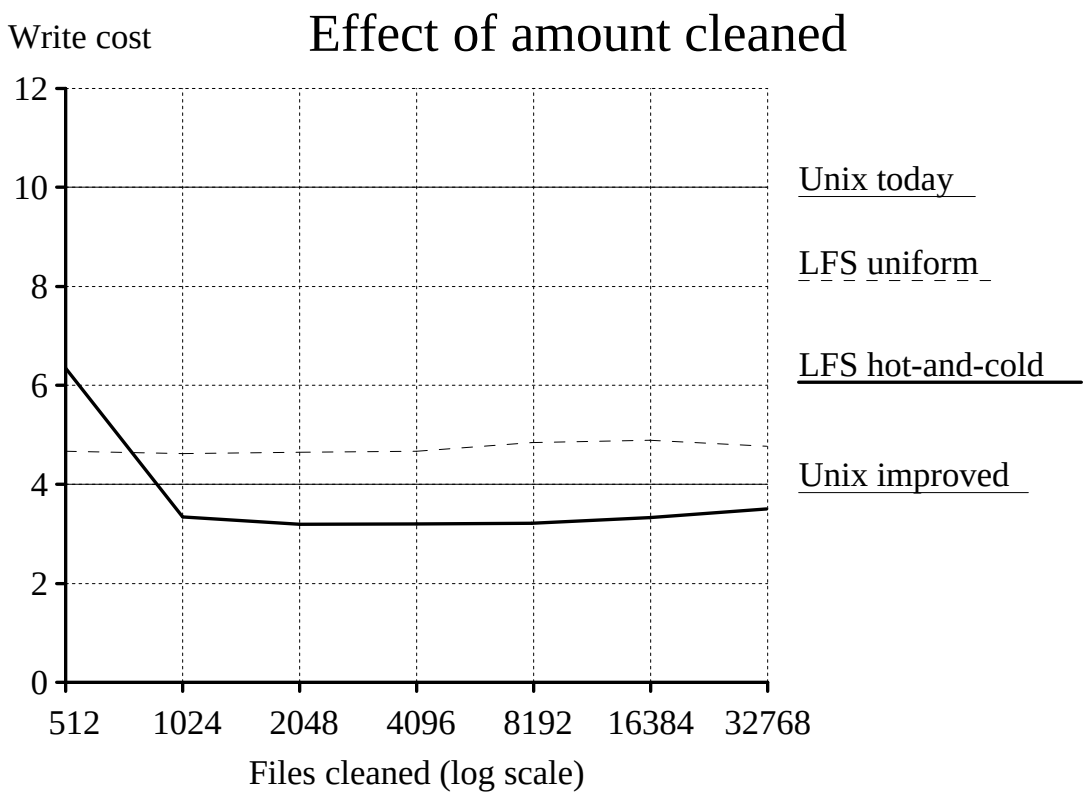

Figure 5-8 — Effect of the amount of data cleaned
This figure shows the effect on the write cost of varying the amount of data cleaned at a time. The curve ‘‘LFS uniform’’ shows the write cost of the uniform access pattern when the number of files cleaned at a time is varied from 512 (the size of a segment) to 32768 files. The curve ‘‘LFS hot-and-cold’’ shows the write cost with the hot-and-cold access pattern with 90% of the references going to 10% of the files. During the simulation, cost-benefit segment selection and file age-sorting reorganization was used and the disk capacity utilization was fixed at 75%. For reference the write cost of the ‘‘Unix today’’ and ‘‘Unix improved’’ are included. Except for very low numbers of files, the write cost is not sensitive to the amount of data cleaned.

Using the cleaning mechanism to reorganize data creates some interesting questions that have not been answered. For example, is it possible to both enhance read performance during cleaning and keep good write performance?

5.5.2. Cleaning timing and amount

Sprite LFS policy is to delay cleaning until the number of clean segments drops below a threshold value, typically a few tens of segments. The threshold value must be large enough that there is enough space on disk for the data being cleaned and the checkpoint that follows the cleaning operation. One way to ensure that the threshold value is large enough is to make it larger than the size of the file cache size. With a threshold value of this size even the largest checkpoint write back can be handled. The problem with this approach is that it requires that too much of the disk space be reserved. For example, on a 300 megabyte disk it is too much to require that the entire file cache of 100 megabytes be reserved.

Rather that requiring the threshold value to be function of the file cache size, Sprite LFS controls the flow of modifications into the file cache so that the threshold value may be much smaller than the size of the file cache. The amount of a file system’s data allowed to be dirty in the file cache is reduced as the number of clean segments approaches the threshold value.

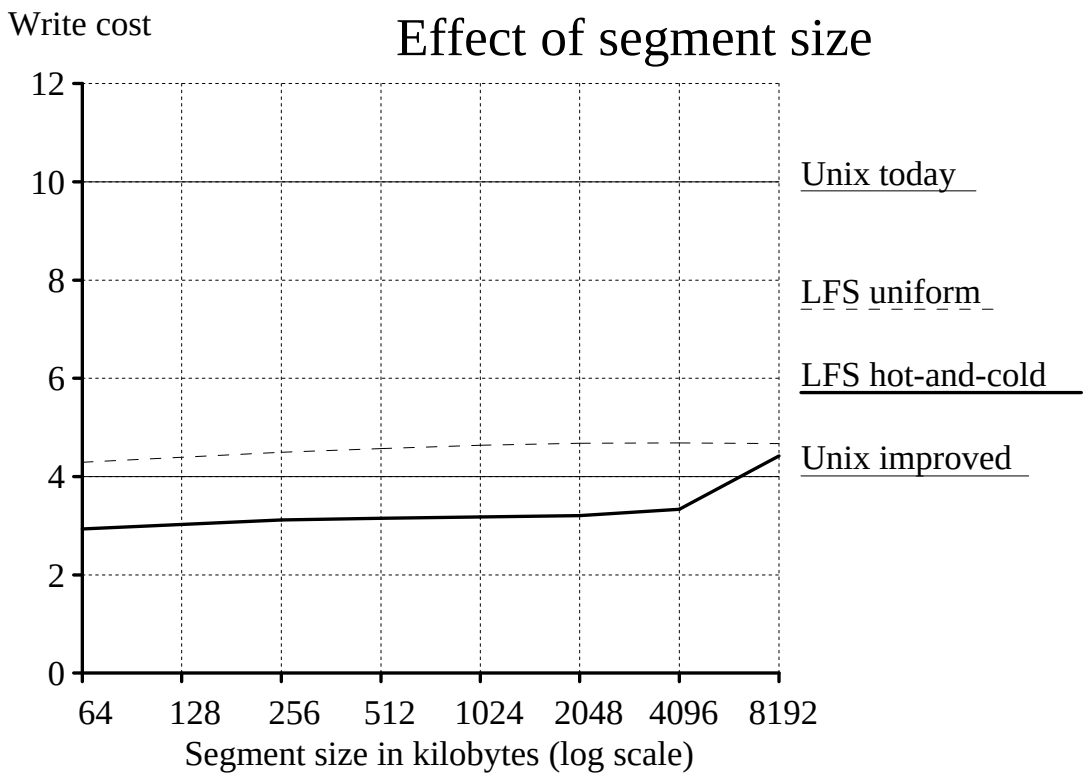

Figure 5-9 — Effect of segment size on write cost
This figure shows the write cost as a function of segment size for two access patterns. The curve ‘‘LFS uniform’’ shows the write cost of the uniform access pattern for segment sizes varying from 64 kilobytes up to 8192 kilobytes. The curve ‘‘LFS hot-and-cold’’ shows the same information for the hot-and-cold distribution with 90% of the references going to 10% of the files. The same policies as Figure 5-8 were used as well as the same disk capacity utilization of 75%. The flatness of the lines indicate than the write cost is not sensitive to the segment size.

By starting to clean when the system reaches the minimum number of clean segments allows Sprite LFS to postpone cleaning as long as possible. While this delay increases the chance that the data in segments is overwritten or deleted, thus lowering the write cost, it also leads to high variances in response times. Reading in a large number (possible 100s) of segments to fill the reserved portion of the file cache and writing the files back can take minutes. If writes are prohibited until the cleaning mechanism generates clean segments, requests can experience delays of up to a minute.

To prevent these large delays, Sprite LFS starts cleaning before the system reaches the minimum threshold. An additional parameter in the superblock specifies the number of clean segments above the minimum threshold below which cleaning will be started. By starting cleaning early, application write requests can continue to the log during the cleaning operation. Write requests are only delayed when the rate is too fast for the cleaner to regenerate live data.

Once cleaning is started, a policy is needed to determine when to stop it. Sprite LFS uses two different approaches to terminate cleaning. The first approach follows what was done in the simulation. Once cleaning starts, segments are read in until the reserved space in the file cache is full. Once the space fills, the files are sorted and written to disk. Unlike the simulator, a second threshold value (typically 50-100 clean segments) specifies how many clean segments need to be generated during cleaning. The number of clean segments generated is the difference between the number read in and the number written out during

cleaning. The motivation behind this policy is to clean enough segments to run for a little while without the disturbance of segment cleaning. Once the threshold number of clean segments has been generated, a checkpoint is performed, the segments are marked clean, and the cleaning is terminated.

A different approach to terminating cleaning is used when the file system is being overrun with modifications. This happens when modifications occur so fast that the log overruns all the clean segments and can only being written after segment cleaning completes. To reduce the delays, the algorithm is modified to clean fewer segments at once and checkpoint immediately to have the clean segments available for reuse.

5.5.3. Segment size

The last issue addresses in the simulations is the size of the segments. Figure 5-9 shows the effect of segment size on write cost. A smaller segment size leads to a sightly lower write cost in the simulation. The reason for the small improvement is that a smaller segment size for a fixed sized disk means more segments on the disk. There are 128 times more segments on a disk with 64 kilobyte segments (assuming 4 kilobyte file size) than are on a disk with 8 megabyte segments. More segments means the utilizations will have a larger variance with more low-utilization segments available for the cleaner. Smaller segment sizes also make it easier to keep files of the same age in the same segment. This explains why the write cost increases with segment size when locality is present.

Although Figure 5-9 calls for small segment sizes, the actual write cost of the system will not be minimized with small segments. As discussed in Section 4.4.1, Sprite LFS needs large segments in order to access the disk efficiently. The simulator's write cost model fails for small segments. Figure 5-9 simply means that the segment should be sized no larger than necessary to amortize the disk overheads.

5.6. Summary

Results from the simulation of the Sprite LFS log wrap provided insight into what constituted good cleaning policies. The results also indicated that a log-structured format could achieve high write performance even for cost-sensitive environments where disk capacity utilizations is high. The key discovery was that selecting segments to clean based on utilization alone caused poor write performance on workloads with locality. This led to the cost-benefit selection policy that selects segments based both on the utilization and the age of the data in the segment.

Based on the knowledge gained from the simulation, Sprite LFS was implemented and installed as the file system for Sprite users at Berkeley. This allowed for the collection of numbers and experience that are presented in the next chapter.

CHAPTER 6

Experience with Sprite LFS

This chapter reports on experiences with the Sprite LFS prototype. Two distinct groups of experiments were run to evaluate the Sprite LFS design. The first consisted of running synthetic workloads on the Sprite LFS file system to evaluate the performance on known workloads. Section 6.2 presents the results of these experiments. The second group of experiments involved observing the behavior of Sprite LFS file systems over long-term use by the Sprite user community. These measurements are presented in Section 6.3.

The two groups of experiments complement each other, providing more insight into the benefits and problems of Sprite LFS than either would have by itself. The synthetic workload generation experiments allow the behavior of Sprite LFS to be compared to other file system implementations. Using different but well defined workloads, the strengths and weaknesses of Sprite LFS relative to existing designs can be evaluated.

The chief disadvantage of synthetic workload generators is that it is difficult to write a program that generates the type of workloads presented by a large community of users. The workloads in the Unix environment tend to be complex and they vary both from system to system and over time on the same system. The metric of interest is the behavior of the file system under these actual workloads over long periods of time. Observing the Sprite LFS file system behavior under the diverse workload presented by the Sprite user community allows Sprite LFS to be evaluated much more thoroughly than what would be possible with the simulation techniques presented in the previous chapter. Because of the diverse and changing workload presented by the user community, the experiments do not allow for comparisons between Sprite LFS and other file systems. They only demonstrate the viability of Sprite LFS for the Sprite user community's workload.

6.1. History of Sprite LFS

The implementation of Sprite LFS was begun 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 the storage of of about forty users for day-to-day computing.

6.2. Synthetic workload analysis

The experiments using synthetic workloads consist of running benchmark programs that generate file system requests and comparing their performance on Sprite LFS and two other file systems. The two other file systems studied were the original Sprite file system (OFS) [47] and the Berkeley FFS running in SunOS 4.0.3 (SunOS). OFS provides an interesting comparison point for Sprite LFS because both systems reside in the Sprite kernel and they share most of the file system code. The difference between the file systems is that Sprite LFS uses a log structure while OFS uses a traditional Unix file system organization. The behavior differences between the two file systems can be attributed directly to the the disk storage manager part of the file system code.

The SunOS file system also provides an interesting comparison point because the file system and its performance characteristics are widely published in the literature[5, 20]. However, SunOS shares no code in common with Sprite LFS so differences in performance may be due to other factors than the disk storage management technique used.

The hardware used in all experiments presented in this section was a Sun-4/260 (8.7 integer SPECmarks) with 32 megabytes of memory, a Sun SCSI3 host bus adapter, and a Wren IV disk (1.3 megabytes/sec maximum transfer bandwidth, 17.5 milliseconds average seek time). For both Sprite 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 the Sprite file systems used a four-kilobyte block size. Sprite LFS used a one-megabyte segment size. In each case the system was running in multiuser mode but was otherwise quiescent during the test.

The synthetic workload experiments used two different types of workload generators: micro-benchmarks and macro-benchmarks. Micro-benchmarks are small benchmarks programs that measure the performance of the file system on simple workload mixes such as creating small files or reading a large file sequentially. Macro-benchmarks are large sets of programs that try to approximate the workload generated by a real environment.

6.2.1. Micro-benchmarks

The micro-benchmark experiments were performed using a collection of small programs to measure the best-case performance of Sprite LFS and compare it to the other file systems. The programs evaluated the performance of workloads such as those containing small files and those with different access patterns to large files. Care was taken in the design and execution of these programs to ensure that they generated disk accesses even in the face of caching and delayed writes. While these benchmarks report best-case performance and it would be incorrect to assume that they can be used to predict performance of realistic workloads, they nevertheless are useful for illustrating the strengths and weaknesses of the file systems.

The benchmarks were run on freshly created file systems. For Sprite LFS, using a newly created file system means that no cleaning occurred during the benchmark; see Section 6.3 for measurements of cleaning overhead. For the traditional Unix file system, newly created file systems means the allocation algorithms are faced with no fragmentation and thus perform best.

6.2.1.1. Small file performance

Figure 6-1 shows the results of a benchmark measuring the small-file-handling capabilities of the file system. The benchmark contained three phases that tested the speed of small-file creation, retrieval, and deletion. During the create phase, 10000 one-kilobyte files were created by opening a non-existent file with the creation option, writing one kilobyte of data to the file, and closing the file. The read phase flushed the file cache and timed the reading of the 10000 files from disk. The delete phase timed the deletion of the files.

Create phase

Sprite LFS is almost ten times as fast as SunOS for the create phase of the benchmark. This order-of-magnitude difference in performance can be attributed to two benefits of Sprite LFS: asynchronous metadata updates and sequential disk writes. Because of the small file size, the handling of file system metadata has a large effect on performance. Each file creation required modifications of a directory and the allocation of am inode for the new file. The SunOS file system synchronously writes directory and inode blocks while Sprite LFS asynchronously writes the changes. Since many of the modifications will be to the same directory and inodes allocated to the same inode block, SunOS's synchronous writes disable write caching. Each file creation in SunOS takes several disk writes to update the directory, inode block, and file data block. These multiple writes limit performance to one file creation every 60 milliseconds.

The OFS file system uses an on-disk structure like SunOS but uses asynchronous writes to the directory and inode block like Sprite LFS. Asynchronous metadata writes allow OFS to have better performance than SunOS but it is still significantly below that of Sprite LFS. The performance advantage that Sprite LFS has over the other two file systems is that changes are written sequentially to disk in large transfer units. The SunOS and the OFS file systems must write to non-sequential disk blocks with single-block transfers. The difference in disk efficiency between sequential and non-sequential gives Sprite LFS the huge performance advantage.

Figure 6-2 shows the resource utilizations of the file systems. The figure shows Sprite LFS is in a different state of operation than SunOS: Sprite LFS is CPU-bound while SunOS is disk-bound. Under Sprite LFS the disk is active only 17% of the time while under SunOS it is 85%. This is particularly impressive considering Sprite LFS is sending the files to disk 10 times faster than SunOS. Since both systems use about the same amount of CPU per file created, the CPU utilization is an indication of the number of files created. Unlike SunOS which bottlenecked on the disk, Sprite LFS was only limited by the CPU power of the machine.

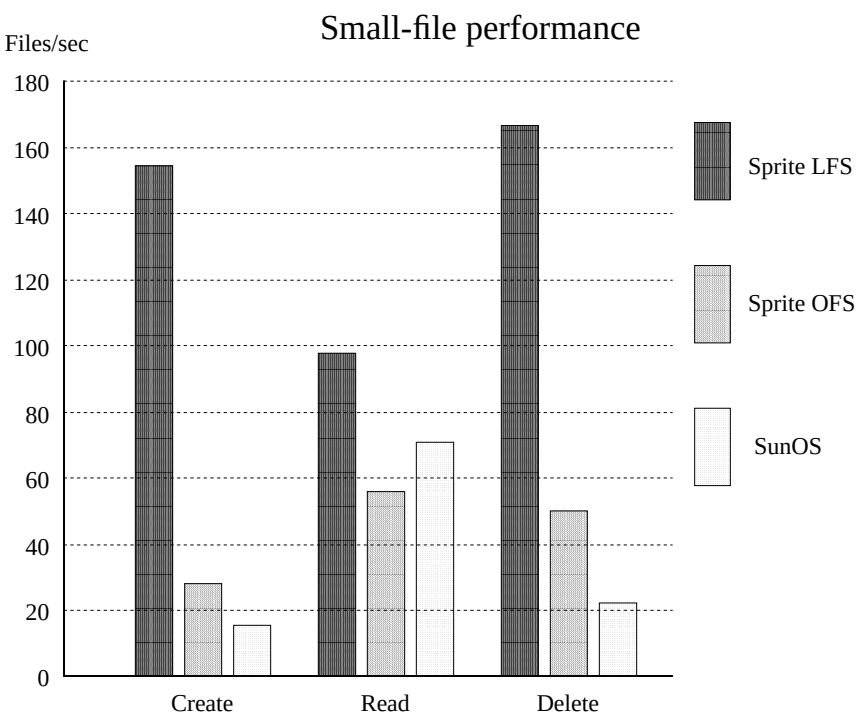

Figure 6-1 — Small-file performance
This figure shows the result of a benchmark that created 10000 one-kilobyte files, then read them back in the same order as created, then deleted them. Speed is measured by the number of files per second for each operation on the three file systems. The logging approach in Sprite LFS provides an order-of-magnitude speedup over SunOS or Sprite's original file system for creation and deletion.

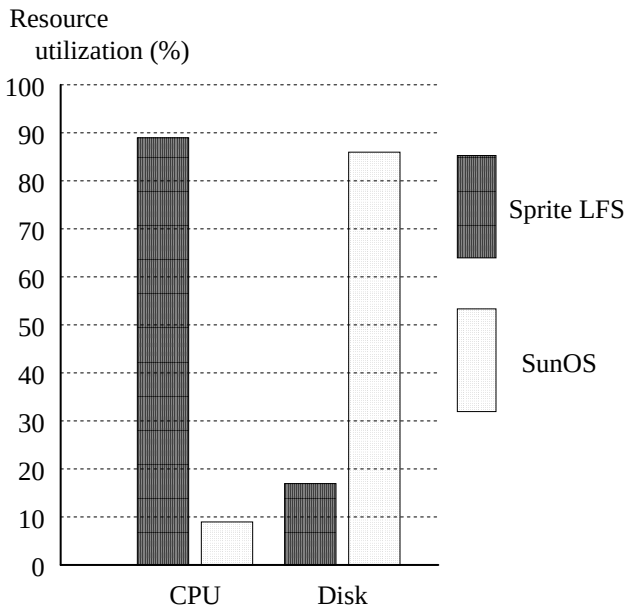

Figure 6-2 — Resource utilization during file creation
This figure shows the resource utilization of the CPU and disk for Sprite LFS and SunOS running a benchmark that created 10000 one-kilobyte files.

From Figure 6-1 and Figure 6-2 it is possible to see the difference in disk efficiencies between SunOS and Sprite LFS. SunOS transfers the files to disk at 15 kilobyte/second and keeps the disk busy 85% of the time. This means that SunOS is getting less than 2% of the maximum efficiency. Sprite LFS transfers 150 kilobyte/second using the disk 15% of the time. This is an disk efficiency rating of over 97%.

The efficient use of the disk resources will be even more important for the faster CPUs of the future. Figure 6-3 shows predicted performance of the create phase on faster CPUs. As CPUs get faster, Sprite LFS can increase its performance by another factor of 4 to 6 before the disk saturates. SunOS has already saturated the disk and can expect little improvement from faster CPUs. Overall, Sprite LFS uses the disk 40 to 60 times more efficiently that SunOS.

Read phase

The file read phase result in Figure 6-1 show less difference between the file systems. This would be expected because the file systems maintain the same index structures and access the disk in similar ways when retrieving files. The differences in performance of the file reads can be attributed to the locality in the disk layout. The log structure of Sprite LFS packs together the files with the metadata reducing the disk delays between file retrievals. Sprite LFS was particularly well suited for the benchmark because the files were read in the same order they were created which is also the same order they are in the log on disk.

SunOS and OFS suffered longer disk delays than Sprite LFS because of their allocation policy that attempts to spread files apart on disk. Files are not packed together so that they can continue to be allocated near-contiguously even if they grow. This spacing between files allows for growth but also increases the disk delays between file retrievals. The difference between the two traditional file systems SunOS and OFS is due to be better allocation policies in SunOS. OFS lacks the cylinder group technique that SunOS uses to reduce the distance between files in the same directory.

It is worth noting that had the file cache not been flushed before the read test, all the file systems would have the same read performance. Any differences in disk layout is irrelevant if read caching

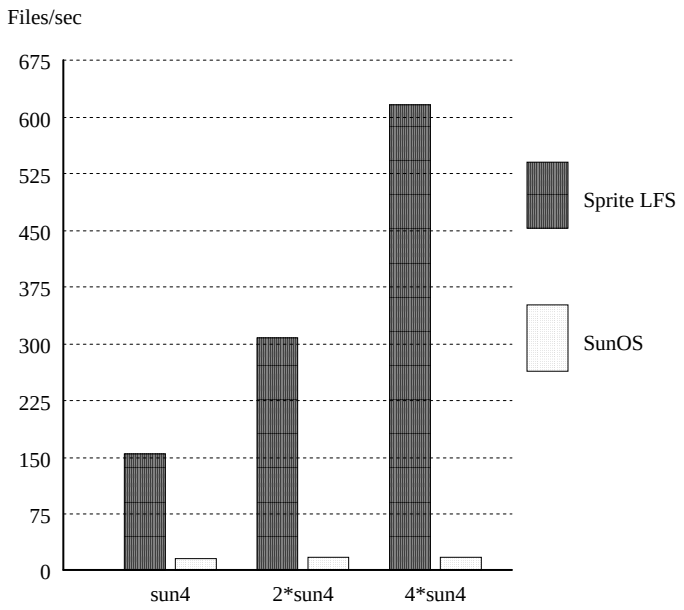

Figure 6-3 — Prediction of future small-file performance
This figure estimates the performance of Sprite LFS and SunOS system for creating files on faster computers with the same disk. The measurements over ‘‘2*sun4’’ and ‘‘4*sun4’’ represent predictions for CPUs speeds of twice and four times the sun4 described at the beginning of this section. The ‘‘sun4’’ measurements are reproduced from Figure 6-1 for reference. The predictions are based on the data in Figure 6-1 and Figure 6-2. In SunOS the disk was 85% saturated so faster processors will not improve performance much. In Sprite LFS the disk was only 17% saturated while the CPU was 100% utilized; as a consequence I/O performance will scale with CPU speed.

satisfies the read requests.

Delete phase

The delete phase in Figure 6-1 shows results similar to the create phase. This is not surprising because deleting a file modifies the same metadata structures as file creation. The name must be removed from the directory and the inode must be marked as deallocated. SunOS performs both updates synchronously while OFS gets twice the performance by only updating the inode synchronously. Sprite LFS is over 3 times faster than either SunOS and OFS. Like the create phase, the difference is due to the sequential asynchronous access patterns of Sprite LFS.

6.2.1.2. Large file performance

Although Sprite LFS was designed for efficiency on workloads with many small file accesses, Figure 6-4 shows that it also provides competitive performance for large files. The benchmark measured the speed of five operations on a 100-megabyte file. The operations were: creating the file by writing 100 megabytes sequentially, reading the entire file sequentially, randomly writing 100 megabytes of 8-kilobyte file blocks, randomly reading 100 megabytes of 8-kilobytes blocks, and sequentially reading the file again. The test was designed to measure transfer rates for different access patterns to large files.

In all cases Sprite LFS has a high write bandwidth. The log structure of Sprite LFS guarantees this high write performance regardless of the object size or access patterns. This is not true for traditional storage manager designs such as SunOS and Sprite OFS. Non-sequential write access patterns have substantially lower performance. For example, Sprite LFS is much faster on the random write test because it turns the random writes into sequential disk writes while traditional file systems must do disk seeks to

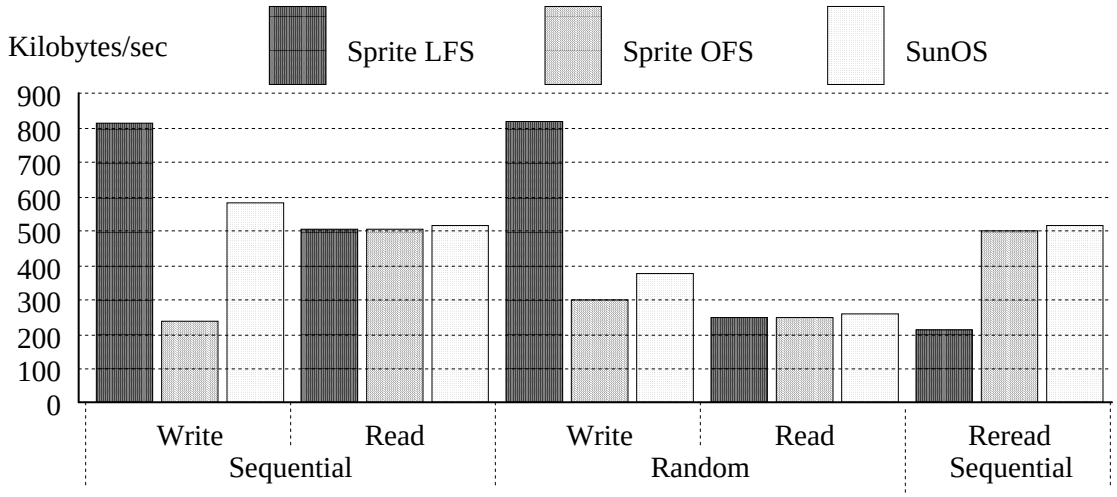

Figure 6-4 — Large-file performance
The figure shows the speed of a benchmark that creates a 100-megabyte file with sequential writes, then reads the file back sequentially, then writes 100 megabytes randomly to the existing file, then reads 100 megabytes randomly from the file, and finally reads the 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 the traditional file systems with the exception of sequential reading of a file that was written randomly. The exceptionally low sequential write bandwidth of Sprite OFS is caused by the allocation pattern and the high disk controller overheads. Sprite OFS is allocating blocks of the file on successive disk blocks but the controller overhead is too high to write successive blocks. The result is the write must wait for nearly a full disk rotation; only one disk block (4 kilobytes) is transferred per rotation.

update the file in place. The performance advantage of Sprite LFS is not as great as in the small-file creation test shown above because both systems implement asynchronous writing of data blocks. The differences between SunOS and Sprite LFS on the random write test show the improvements of sequential transfers over sorted random disk transfers.

Even when the write access pattern is sequential, Sprite LFS is still faster than the traditional file systems. The reason for this is that Sprite LFS groups many blocks into a single large I/O, whereas the others perform individual disk operations for each block. This effect can be seen in the sequential write test in which both Sprite LFS and SunOS allocate the file the same way but Sprite LFS is able to write it faster because of the larger transfer size. The performance advantage of LFS is even more pronounced relative to Sprite OFS because of missed disk revolutions in Sprite OFS.

The performance advantage of Sprite LFS on sequential writes is an artifact of the implementation. It is possible for a block-oriented file system to group requests together to improve performance. For example, a newer version of SunOS[20] groups write requests and should therefore have performance equivalent to Sprite LFS.

The sequential read following the file's creation proceeds at the same rate for each file system. Although all three file systems use different allocation policies the disk format of the file is the same for the file systems; the file is allocated sequentially on disk. Furthermore, both file systems maintain the same index structures and fetch the file one block at a time. The result is similar read performance.

The random write test causes Sprite LFS to rewrite the file to the log, this time in random order. This does not effect the performance of the random read test. Random read requests cause random disk accesses no matter what format the file is on disk.

The sequential re-reading of the file in the last phase of the benchmark demonstrates a potential disadvantage of the log-structured file system. The sequential reading of a randomly written file causes random disk read requests to retrieve the blocks of the file. In this case the read performance of Sprite LFS is substantially lower than the other file systems.

The sequential re-read test shows 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 on disk for the assumed read access patterns. 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. No form of locality helps random read requests so the file systems will have the same performance, even though the blocks are laid out very differently. However, if the nonsequential reads occurred in the same order as the nonsequential writes then Sprite LFS would have been much faster.

6.2.1.3. Crash recovery

The last micro-benchmark presented in this section tests the time it takes to recover a Sprite LFS file system after various crash scenarios. The time to recover depends on the checkpoint interval and the rate and type of operations being performed at the time of the crash. Table 6-1 shows the recovery time for different file sizes and amounts of file data recovered. The different crash configurations were generated by running a program that created one, ten, or fifty megabytes of fixed-size files before the system was crashed. A special version of Sprite LFS was used that had an infinite checkpoint interval and never wrote directory changes to disk. During the recovery roll-forward, the created files had to be added to the inode map, the directory entries created, and the segment usage table updated. Having to recreate the directory entries made this a worse than normal case scenario.

Table 6-1 shows that recovery time varies with the number and size of files written between the last checkpoint and the crash. Since file data blocks and be quickly skipped over during recovery, recovery time is much more sensitive to the number of files rather than the total amount of data. Recovery times can be bounded by limiting the amount of data written between checkpoints. From the average file sizes and daily write traffic in Table 6-2, a checkpoint interval as large as an hour would result in average recovery times of around one second. Using the maximum observered write rate of 150 megabytes/hour, maximum recovery time would grow by one second for every 70 seconds of checkpoint interval length.

Sprite LFS recovery time in seconds			
File Size	File Data Recovered		
	1 MB	10 MB	50 MB
1 KB	1	21	132
10 KB	< 1	3	17
100 KB	< 1	1	8

Table 6-1 — Recovery time for various crash configurations
The table shows the speed of recovery of one, ten, and fifty megabytes of fixed-size files. The system measured was the same one used in Section 6.2. Recovery time is dominated by the number of files to be recovered.

6.2.2. Macro-benchmarks

In everyday use Sprite LFS does not feel much different to the users than Sprite OFS, the Unix FFS-like file system in Sprite. The reason is that the machines being used are not fast enough to be disk-bound with the current workloads. For example, Figure 6-5 shows a comparison of Sprite LFS, Sprite OFS, and SunOS running the modified Andrew benchmark[45]. The Andrew benchmark was designed to measure the file system's handling of common Unix requests. Over the entire benchmark, Sprite LFS is only 5% faster than Sprite OFS and 21% faster than SunOS. The reason for the limited speedup is that the benchmark is CPU-bound. Even with the synchronous writes of SunOS, the benchmark has a CPU utilization of over 80%, limiting the speedup possible from changes in the disk storage management.

Figure 6-5 further explains the numbers by showing the times for each phase of the benchmark. The benchmark starts by creating 20 directories. Sprite resident file systems are a factor of 3 faster than SunOS because they do no synchronous writes during this phase. The second phase copies 70 files with an average file size of 4860 bytes into these directories. Because the benchmark is run several times these files

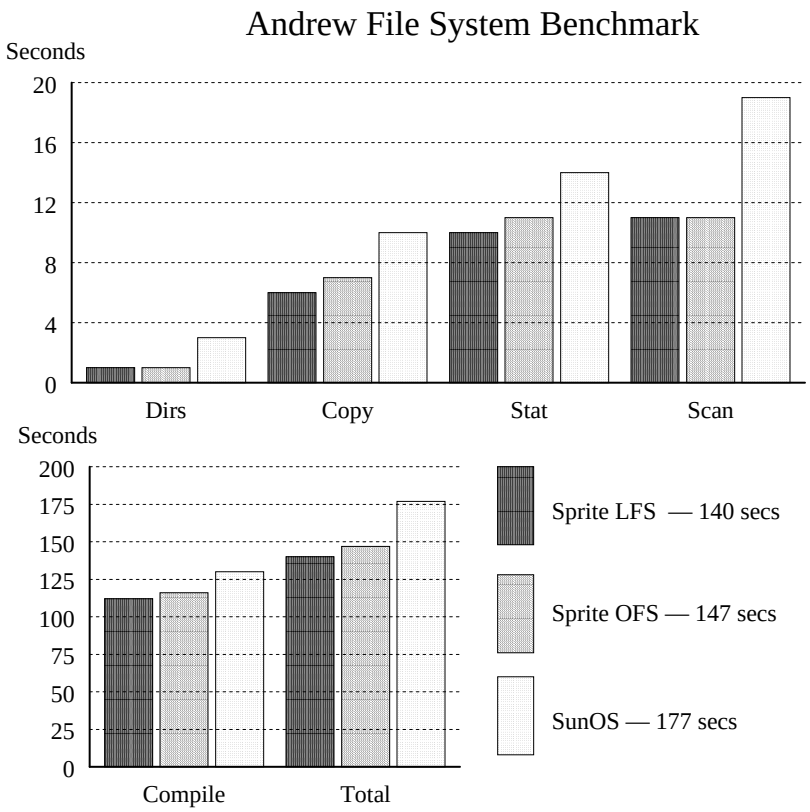

Figure 6-5 — Andrew file system benchmark results
This figure compares the results of Sprite LFS, Sprite OFS, and SunOS on the modified Andrew file system benchmark. The benchmark consists of five phases. The time in seconds for each phase is shown as well as the total for the entire benchmark. The first phase (Dirs) involves creation of directories. The second phase (Copy) copies many files into the directories. The third phase (Stat) gets the attributes of every copied file with a stat() system call. The fourth phase (Scan) reads every copied file searching for and counting words. The last phase (Compile) compiles all the copied files.

have been recently accessed so they can be read from the file cache. With a few megabytes of file cache, it is possible to complete the entire benchmark without doing any disk reads. The timing of the copy phase of the benchmark is further complicated by the write caching of file blocks done by Unix and Sprite. The copy phase is over long before the data is actually written to disk. Again, the difference in the timings is due to the synchronous metadata writes done by SunOS.

The next two phases of the benchmark get the attributes (stat) and read each file looking for a pattern (scan). Because of file caching, none of the file systems do any disk I/O during these phases. Performance is limited by the CPU speed and the number of instructions executed for each file system. The last phase compiles the files into a library. This phase took over 70% of the total time and was heavily CPU-bound. Like the other phases, no disk reads were needed. The compiler's creation and deletion of temporary files slowed the more synchronous SunOS.

The results of the Andrew benchmark are typical of most of the macro-benchmarks examined. The machines in the office/engineering environment of today are not fast enough to overload the disk subsystem and do useful work at the same time. Of course, this will change in the near future as 150 to 200 MIPS machines displace the 10 MIPS machines used above. With CPUs 10 to 20 times faster, the macro-benchmarks such as the Andrew benchmark will be heavily disk bound on current storage managers. On these much faster machines the asynchronous and sequential writes of Sprite LFS will result in a large performance gain.

6.3. Measurements from the Sprite LFS system

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). To assess the performance of Sprite LFS and especially the cost of cleaning and the effectiveness of the cost-benefit cleaning policy, statistics about the production log-structured file systems were recorded over a period of several months. These statistics allow an evaluation of Sprite LFS on the workloads presented by the Sprite user community.

6.4. Sprite environment

The following Sprite LFS file systems were studied:

`/src/kernel`	Sources and binaries for the Sprite kernel.
`/local`	Source and binary directories for locally supported computer programs.
`/swap1`	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.
`/tmp`	Temporary file storage area for 40 Sprite workstations.
`/pcs`	Home directories and project area for research on parallel processing.
`/src`	Source directory for Sprite user-level commands and libraries.
`/x11`	Sources and binaries for X window system from MIT.
`/user1`	Home directories for miscellaneous users of Sprite.
`/user4`	Home directories for researchers on hardware aspect of disk arrays.
`/user5`	Home directories for Sprite developers. Workload consists of program development, text processing, electronic communication, and simulations.
`/user6`	Same as /user5.

The measurements were gathered from the file systems over a year-long period of time. Table 6-2 shows show overall statistics on the file systems. In order to eliminate start-up effects, measurements on file systems did not start until several months after the file systems went into use. The measurements were generated from a series of event counters coded into the Sprite LFS implementation. These counters were written to disk on every checkpoint. Snapshots of the values of the counters were taken at regular intervals for analysis. The compressed size of these counters was over 600 megabytes per year of operation.

Sprite LFS file systems				
File system	Disk Size	In Use	Avg File Size	Traffic
/src/kernel	1280 MB	81.4%	31.2 KB	4.2 MB/hour
/local	500 MB	55.2%	37.1 KB	2.0 MB/hour
/swap1	633 MB	38.3%	88.3 KB	13.1 MB/hour
/tmp	287 MB	21.6%	23.7 KB	1.7 MB/hour
/pcs	990 MB	63.1%	10.5 KB	2.1 MB/hour
/src	954 MB	72.2%	16.7 KB	1.6 MB/hour
/X11	954 MB	69.8%	44.2 KB	0.8 MB/hour
/user1	633 MB	73.1%	8.9 KB	2.9 MB/hour
/user4	631 MB	78.4%	24.9 KB	1.6 MB/hour
/user6	1280 MB	73.9%	23.1 KB	4.4 MB/hour
/user5	954 MB	71.1%	20.3 KB	2.9 MB/hour

Table 6-2 — Sprite LFS production file systems

For each Sprite LFS file system the table lists the disk size, the fraction of the available disk space in use, the average file size, and the average write traffic rate.

6.5. Cleaning overheads

The cleaning performance and the associated write cost for the file systems can be found in Table 6-3. The behavior of the production file systems has been substantially better than predicted by the

Write cost in Sprite LFS file systems					
File system	In Use	Segments		*u*	Write Cost
		Empty	Avg		
/src/kernel	81.4%	89754	74.0%	0.296	1.36
/local	55.2%	78803	36.2%	0.205	1.88
/swap1	38.3%	100095	46.3%	0.281	1.81
/tmp	21.6%	13962	83.2%	0.144	1.20
/pcs	63.1%	37294	58.3%	0.177	1.53
/src	72.2%	17831	70.4%	0.280	1.41
/X11	69.8%	11569	61.2%	0.552	1.77
/user1	73.1%	34163	50.9%	0.321	1.77
/user4	78.4%	21906	56.2%	0.328	1.68
/user6	73.9%	98752	70.1%	0.210	1.39
/user5	71.1%	12155	61.2%	0.226	1.52

Table 6-3 — Segment cleaning statistics and write costs for production file systems
For each Sprite LFS file system the table shows the average disk capacity utilization, the total number of segments cleaned over the measurement 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 even with cleaning overhead Sprite LFS can achieve about 70% of the maximum sequential write bandwidth for workloads with relatively small files. This is significantly better than the 5 to 10% of traditional Unix storage managers.

simulations in Chapter 5. Even though the overall disk capacity utilizations were up to 82%, 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 1.2 to 1.8, in comparison to write costs of 2.5-3 in the corresponding simulations.

The low overhead from cleaning is reflected in the distribution of segment utilization on the disks. Figure 6-6 shows the distribution of segment utilizations, gathered in a snapshot on the disk. The segments form a bimodal distribution with most segments either fully utilized or nearly empty. The empty ones are cleaned at a low cost while the full ones incur no cleaning overhead and keep the disk capacity utilization high.

There are two reasons why cleaning costs are 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.

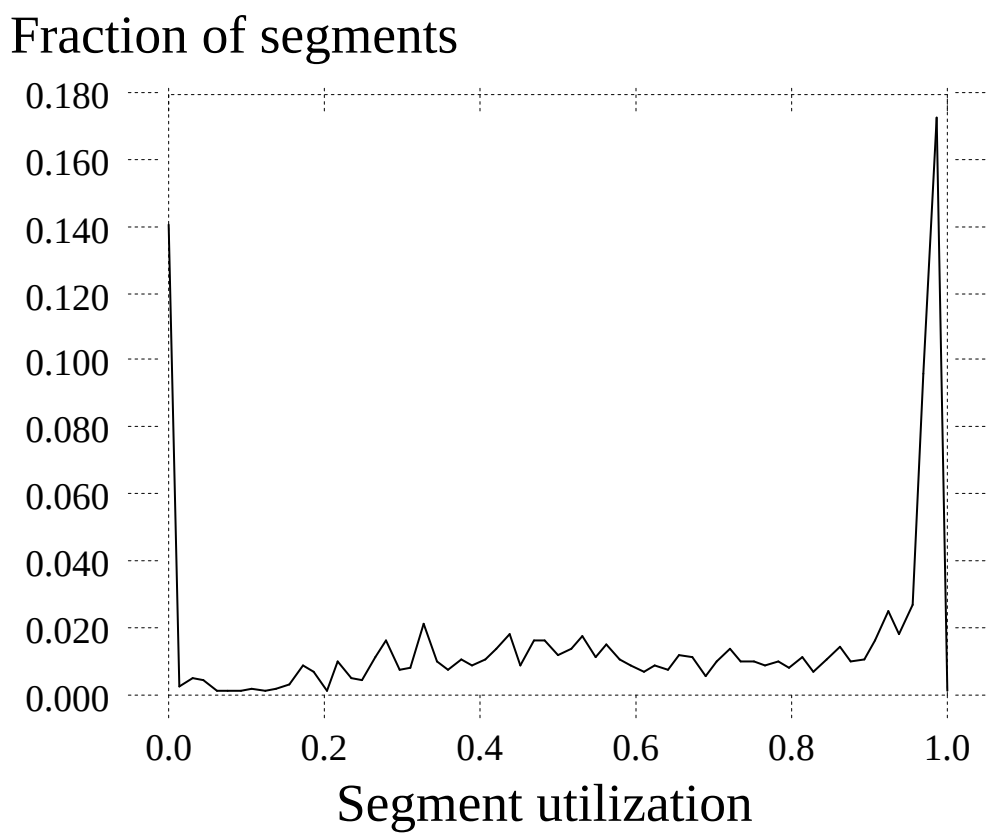

Figure 6-6 — Segment utilization in the /user6 file system
This figure shows the distribution of segment utilizations in recent snapshots of the `/user6` disk. The distribution shows large numbers of fully utilized segments and totally empty segments. The distribution is formed from the composite of many snapshots taken at random times in the day over a week long period. Distributions for the other file systems have similar shapes.

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 6.2 were a bit overly-optimistic, the measurements in this section are, if anything, overly-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. The above results show that even without this assumption, Sprite LFS can keep the cleaning overhead low.

6.6. Other overheads in Sprite LFS

Table 6-4 shows the relative importance of the various kinds of data written to disk. The table shows the blocks both in terms of how much space they occupy on disk and how much of log bandwidth they take. More than 99% of the live data on disk consists of file data blocks and indirect blocks. This means that the metadata structures unique to Sprite LFS impose little space overhead. The inode map and segment summary blocks occupy less than 1% of the allocated disk space.

In terms of disk bandwidth, the impact of the Sprite LFS unique data structures is higher. From 6% to 10% of the information written to the log is inode map and segment usage table blocks. These blocks tend to be overwritten quickly so the space they occupy can be reclaimed. The inode map alone accounts for more than 7% of all the data written to the log but only 0.2% of the disk space. I suspect that this is because of the short checkpoint interval currently used in Sprite LFS, which forces metadata to disk more often than necessary. I expect the log bandwidth overhead for metadata to drop substantially with roll-forward recovery and an increased checkpoint interval.

6.7. Summary

This chapter presented measurements and experience with the Sprite LFS prototype. Measurements of synthetic workloads were used to compare the performance of Sprite LFS with the original Sprite file system (Sprite OFS) and the SunOS file system. The synthetic workloads consisted of tests of read and write operations on small and large files. On all tests containing modifications, Sprite LFS was faster than the other file systems. For the test that created or deleted small files, Sprite LFS was as much as an order

	`/user6`		`/kernel`		`/swap1`	
Block type	Live	Log	Live	Log	Live	Log
Data blocks*	72.5%	85.7%	80.0%	90.8%	37.7%	85.1%
Free space*	25.8%	0.0%	18.4%	0.0%	61.2%	0.0%
Indirect blocks*	0.9%	1.8%	1.1%	1.5%	0.5%	4.3%
Inode blocks*	0.1%	2.5%	1.1%	1.1%	0.0%	3.1%
Inode map	0.1%	7.2%	0.1%	5.3%	0.0%	4.6%
Seg Usage map*	0.0%	2.1%	0.0%	1.0%	0.0%	1.8%
Summary blocks	0.6%	0.6%	0.3%	0.3%	0.4%	0.6%
Dir Op Log	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

Table 6-4 — Disk space and log bandwidth usage
For each block type, the table lists the percentage of the disk space in use on disk (Live) and the percentage of the log bandwidth consumed writing this block type (Log). Three different file systems (`/user6`, `/kernel`, `/swap1`) are shown. The block types marked with '*' have equivalent data structures in Unix FFS.

magnitude faster. Except for one test, Sprite LFS was as fast or faster at reading data. The one exception was the test that reads a large file sequentially after randomly writing it.

The study of the behavior of Sprite LFS under the workload presented by the Sprite user community provides indisputable evidence that a log-structured file system can provide high write performance even under the disk capacity utilizations expected for the cost-sensitive Unix environment. The length of the study, over a year in length, ensured that the effects of fragmentation over long term use were accounted for.

CHAPTER 7

Related work

Storage management is a relatively old field of study in computer systems design so it is not surprising that there exists a great deal of research that is closely related to log-structured file systems. Log-structured file systems and the Sprite LFS design borrow ideas from many other storage management systems including those used for write-once media and database management systems.

Most of the major design points of Sprite LFS have been used in the design of previous storage managers. Ideas in Sprite LFS that are also present in other systems include the use of a log to improve write performance and crash recovery time, writing data in a sequential format, and reducing fragmentation by copying and compacting data. In addition, several other designs with the stated goal of high write performance have been presented.

This chapter compares the ideas in log-structured file systems and Sprite LFS with several other storage managers. The comparison shows how the ideas were used in previous systems and how the Sprite LFS design differs from these systems. Sprite LFS is also compared with other research prototypes and commercial storage managers that have been developed concurrently with Sprite LFS. Comparing and contrasting these new designs with Sprite LFS points out the strengths and weaknesses of Sprite LFS relative to other state-of-the-art designs.

7.1. Log-structured storage systems

The first group of related work consists of systems that share the log-structured format with Sprite LFS. Storage systems that use a log as the primary data structure have appeared in several proposals for building file systems on write-once media. Examples of such systems are Swallow[48-50], Log Files[51], the Optical File Cabinet[52], and others[53]. All these systems write changes in an append-only fashion to a log-like structure and maintain indexing information much like the Sprite LFS inode map and inodes for quickly locating and reading files. The format of the data written to the log and the indexing information differ between the systems but the general concepts are the same.

The chief difference between Sprite LFS and these storage managers is that they were designed for write-once devices. The write-once nature of their media made it unnecessary for the storage managers to reclaim log space for reuse. Once a block was written physically it could not be rewritten.

Of the write-once storage managers, Swallow has the most similarities to Sprite LFS. Swallow implements garbage collection to reuse storage when run on magnetic disks. The disk storage is treated as a circular buffer with data copied to the front when the log wraps upon still live data. This can be viewed as the pure-copying solution to log wrap. Swallow also uses copying to remove all live storage from an entire WORM-disk platter so the platter can be replaced.

7.2. Main-memory storage management

The segment cleaning approach used in Sprite LFS to reclaim log space acts much like scavenging garbage collectors developed for programming languages[54, 55]. Scavenging garbage collectors work by copying the live data together leaving the deleted data in-place and available for reallocation. Early scavenging collectors[54] maintained two regions for data: old space and new space. New storage is allocated in the new space. During garbage collection the live objects in the new space are copied and compacted into the old space. After garbage collection the spaces ‘‘flip’’ with old space becoming the new space and the now empty new space becoming the old space. The copying and compaction of the live objects during garbage collection is exactly the operation performed by segment cleaning.

By using the age sorting of blocks during segment cleaning Sprite LFS separates files into generations much like generational garbage collection schemes[55, 56]. Generational garbage collection

techniques keep several regions of storage allocated in a hierarchy. All allocations are made in one region called the new space. Storage that lives is copied into a higher level (older generation) in the hierarchy. The principle behind generational collectors is to separate data by age so the new space will contain short-lived rapidly changing data while the older generations are more stable and need less collection. Similarly Sprite LFS groups together files of similar age during cleaning so that the files in a segment will be deleted at the same rate.

The segment selection policy of Sprite LFS is similar to the policies that generational garbage collectors use to move data between generations. Like the Sprite LFS cleaning mechanism, the garbage collectors typically base movement between generations on the age of the data and the base decisions on which generation to garbage collect on the amount of fragmentation.

In spite of the similarities between segment cleaning and generational garage collection schemes, there are several major points of difference. Generation collectors typically have a small number of spaces. For example, the SPUR lisp garage collector has four generations[57]. Sprite LFS file systems have a large number of segments. Each segment forms its own generation, and the segments as a whole form a continuum of age brackets. The cost-benefit selection policy differs from those of the main-memory based collectors because of the lack of efficient random access to disk-resident data structures. Unlike the memory-based collection that can do efficient random fetches, the Sprite LFS cost function must include reading the entire segment, not just the live blocks.

Another differences is that Sprite LFS can exploit the fact that blocks can belong to at most one file at a time. This permits the use of much simpler algorithms for identifying live data than those used in the systems for programming languages which must support multiple pointers to the same storage and recursive structures.

7.3. Database storage management

The use of write-ahead logging for crash recovery and high performance was pioneered in database systems[16, 58]. Most commercial database managers use some form of logging. The Sprite LFS design borrows heavily from this area.

The Sprite LFS crash recovery mechanism of checkpoints and roll forward is similar to ‘‘redo log’’ techniques used in database systems and object repositories[59]. The logs in these systems contain a list of operations that potentially must be redone after a crash. During recovery each log entry is examined and the operation redone if its effects are missing. Having the log as the final home of the data in Sprite LFS, simplifies the redo log mechanism. Rather than redoing the operation to the separate data copy as in traditional logging systems, Sprite LFS recovery only has to update the indexes to point at the newest copy of the data in the log.

Collecting data in the file cache and writing it to disk in large writes provides a performance benefit similar to group commit in database systems[60]. Using group commit, database managers collect many changes to the data together and write them to the log in a single disk transfer. Doing one larger transfer rather than many smaller transfers allows the the data base manager to use the disk and CPU more efficiently.

The chief differences between Sprite LFS and the almost all database systems systems that use write-ahead logging for crash recovery and high performance is how the log is used. Both Sprite LFS and the database systems view the log as the most up to date ‘‘truth’’ about the state of the data on disk. The difference is that database systems do not use the log as the final repository for data: a separate data area is reserved for this purpose. Read requests are not processed by the log data structure.

The separate data area of these database systems means that they do not need the segment cleaning mechanisms of the Sprite LFS to reclaim log space. The space occupied by the log in a database system can be reclaimed when the logged changes have been written to their final locations. Since all read requests are processed from the data area, the log can be greatly compacted without hurting read performance. Typically only the changed bytes are written to database logs rather than entire blocks as in Sprite LFS. Along with the benefits of separate data area comes the disadvantage of having to write the data twice: once to the log, and once to the final location. The overhead of this double write can be as high as a factor for two for workloads traditionally supported by the Unix file systems.

One database system that uses techniques similar to Sprite LFS is Elhardt and Bayer's Database Cache[61]. The Database Cache design sequentially writes all modified blocks to a disk that is treated as a circular buffer. In addition, a separate copy of the database is maintained using traditional update in place techniques. Log wrap in the database cache design is handling by writing the still live blocks to the separate copy. Like Sprite LFS, blocks that have been overwritten require no additional processing on log wrap. Unlike Sprite LFS, the having the separate copy means that copying data back into the log is not needed.

7.4. Main-memory database systems

Although most storage manager designs are optimized for read performance, a few designs exist that attempt to improve write performance. Main-memory database systems are the most obvious examples of this approach[43, 60, 62, 63]. They are disk storage mangers in which the entire storage fits in memory so disk reads are only needed during cold start. Because read performance is not important, the evaluation criteria for main-memory data base systems are write performance and crash recovery time. These are similar to the metrics used in the Sprite LFS design.

Because their data fits in memory and reads are not important, main-memory database designs focus on the speed of logging and checkpoint operations. Checkpoints are performed by transferring the entire memory image of the database to disk as fast as possible. Hagmann's system[43] writes the memory image into a circular buffer of *sections* that are similar to Sprite LFS segments. Cleaning of sections is not needed because the contents of a section are always overwritten by the next checkpoint.

Hagmann's system also supports *fuzzy dumps* in which the checkpoint can proceed while modifications occur to the database. Fuzzy dumps would be useful to limit the overhead of Sprite LFS checkpoints. Since the disk accesses of both checkpoints and the log are sequential, main-memory data bases use the disk very efficiently. Another main-memory database by Lehman and Carey[63] writes the checkpoint as a pure-threaded log. The fragmentation due to pure-threading is not presented.

Although the main-memory techniques have high write performance and fast crash recovery, they are also limited to databases that fit in memory. This limitation is too severe to be accepted in a general-purpose file system. Even an implementation restriction that a single file must fit in memory has been found to be unacceptable in the Unix environment. For example, the newest version of the Andrew file system removes a restriction that the entire file must fit on the client workstation disk.

7.5. Other write-optimized storage managers

Another write-optimization technique is to attempt to hide the cost of the writes by doing them during the disk overhead times for disk reads. This approach, used in a system called the Write-only Disk Cache[64], delays writing modifications until a read request forces the disk head to move near the location where the changes are to be written. By ''hiding'' the write requests in the overhead for the read requests the system can process writes with almost zero overhead.

The problem with the Write-only Disk Cache approach is that it needs the read requests to disk in order to process writes efficiently. If the file cache works well and absorbs the read requests then the technique fails.

7.6. Contemporary disk storage managers

Several disk storage managers for Unix workloads have been proposed and built concurrently with Sprite LFS. Two such systems are the Episode[2] and Echo[29] disk storage managers. The designs used in these storage managers build upon earlier research efforts in logging file systems including Cedar[3] and Alpine[27]. This section compares Sprite LFS to these systems. Each of these systems uses logging to increase performance and reduce crash recovery time yet they use the log differently.

Episode is like a traditional Unix file system except that it maintains a small log of changes to file system metadata such as directories and inodes. Changes to directories such as creation or deletion cause a log record to be written before the changed inodes or directories are written to the main storage area of the file system. Only file system metadata records are logged; application file data is processed as in a normal Unix file system.

By using write-ahead logging on the metadata, Episode avoids several of the problems of traditional Unix storage managers. Synchronous disk transfers on file creations and deletions are no longer needed in Episode. Directory operations write a log record and then can write-cache the metadata modifications to both the inode and directory block. The Episode log also supports fast crash recovery so the disk scavenger approach of Unix is not needed.

The disadvantage of the Episode approach as compared to Echo and Sprite LFS is that it does not improve the small-file bandwidth for data. Because file data is not logged, it must be written promptly from the file cache to its home on disk storage. As CPU's become faster, Episode may not be able to keep up with the modification bandwidth requirements of the applications.

The Echo design is like Episode except that all modifications including file application data are logged. By logging everything, Echo can get all the benefits of Episode and also the ability to safely write-cache application file data to improve application write bandwidth. Because application writes are logged, Echo can exploit write caching to the fullest for increased bandwidth.

Logging everything has additional costs, though. The large size of application data means the log in Echo occupies significantly more disk space than in Episode. Consider the example of random writes to a 100 megabyte file measured in Section 6.2.1.2. Almost all the traffic to disk will be application data so Episode's log needs little space. To fully exploit write caching, Echo's log should be a large fraction of the size of the file cache on the machine. Under current technology, the difference in log size would be as great as 100 megabytes, 10% to 30% of the file system's size. The large log sizes needed to log application data means that less disk space is available for the main storage area for files.

The Sprite LFS design compares favorably to both Echo and Episode. By using logging, it achieves the same fast crash recovery as the other systems. In addition it supports higher write bandwidth for small objects than the other systems can obtain. Because the Sprite LFS log is the only data structure on disk it can avoid the problems that Echo has with a large log area stealing disk space from application data.

The areas in which Sprite LFS does not compare favorably are when the file system must operate at close to maximum disk capacity utilization and under certain read access patterns. Sprite LFS's write cost gets very large when the file system is operating near 100% of disk capacity. Like traditional storage managers, Episode's and Echo's performance degrades less quickly than Sprite LFS when the disk fills.

Sprite LFS will also suffer on workloads where clustering or contiguous allocations are important for the read accesses yet the write access pattern does not generate the correct read locality. An example of this is the sequential read after random write example in Section 6.2.1.2.

CHAPTER 8

Conclusion

This thesis presents a new disk storage management technique called a log-structured file system. The basic principle is simple: 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. Not only does this technique use single disks efficiently, it is also the most efficient way to access disk arrays such as RAIDs. The large transfers of log-structured file systems minimize the cost of maintaining the redundancy that is needed for reliability in large RAIDs. This is particularly significant for workloads containing modifications to small objects. Such workloads suffer large overheads when the redundancy is updated under traditional storage managers.

Writing to disk in large transfers requires the storage manager to maintain large free areas on disk. The key to the log-structured file system is to make the overhead of maintaining the free space less than the overheads of traditional storage managers. The Sprite LFS prototype maintains large free areas on disk by using a simple mechanism that resembles garbage collection. The mechanism, called segment cleaning, reads fragmented segments from disk, compacts the data, and writes the result back to disk. Both simulation analysis and experience with the prototype suggest that the overhead for segment cleaning is considerably less than the overheads found in traditional storage managers.

The simulation study presented in Chapter 5 found that the overheads of segment cleaning could be further reduced by proper choice of the policies controlling the cleaning mechanism. Policies that segregate the data by age and include age in the selection of segments to clean significantly improved performance for workloads with locality of reference. The Sprite LFS prototype selects segments to clean based on a cost-benefit analysis that includes a space-time product of the fraction alive in the segment and the age of the data in the segment. The data written back to disk during cleaning is sorted by age to further reduce overheads.

The high write performance of log-structured file system is not at the expense of low read performance. Using data structures like the Sprite LFS inode map to locate and retrieve files, the read performance of a log-structured file system can match or exceed that of traditional storage managers. Although there are workloads that have worse read performance under Sprite LFS, there are also workloads that Sprite LFS provides better read performance.

Although developed to support workloads with many small files, a log-structured file system also works well for large-file accesses. In particular, there is essentially no free-space management overhead for very large files that are created and deleted in their entirety. Of course, the performance advantages of a log-structured file system is less with large files because existing storage manager techniques already handle large sequential accesses efficiently.

The log-structure of Sprite LFS not only improves performance; it also allows Sprite LFS to implement fast crash recovery. Crash recovery in Sprite LFS means restoring the on-disk data to a consistent state. Sprite LFS uses a technique based on checkpoint and roll-forward that allows recovery time to be controlled based on the setting of the checkpoint interval. This means the recovery time of the file system scales with the amount of modifications since the last checkpoint and not the size of the disk storage. Sprite LFS recovers much more quickly on large capacity disks than traditional Unix file systems where crash recovery time depends on the disk size.

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.

8.1. Lessons learned about research

Besides insight into disk storage management techniques, my involvement with Sprite LFS has also give me more understanding of techniques for doing research in computer systems. The Sprite LFS project differs from most other storage management technique because the evaluation stage included long-term measurements of real workloads. An interesting question is whether the large amount of work necessary to build and maintain a full-scale implementation was a productive use of research time. I think the answer to this question is yes. The justification for this answer is made in the following paragraphs.

One of the most interesting evaluation criteria for a disk storage manager is the overhead due to fragmentation over the periods of many months or years. Use of Sprite LFS as a production file system for periods of up to a year has captured this effect for log-structured file systems. Alternative evaluation techniques such as trace-driven simulations seem to be limited to time periods measured in days or weeks. For example, Baker[15] collected 192 hours of traces for a file caching study. Traces of this length are sufficient for file cache analysis but insufficient to examine the important issues in disk storage management. Without an implementation in production use the real overheads of fragmentation can not be known.

The production use of Sprite LFS also greatly enhances the credibility of the research results. There is no question that the Sprite LFS is implementable and works correctly. There does not exist some insurmountable problem awaiting the first implementor of the technique.

One disadvantage of the Sprite LFS implementation was that transfer of the implementation to systems other than Sprite was difficult. The implementation was closely tied to the internal Sprite file system design which is sufficiently different from standard Unix file systems internals such as vnodes[44] that it would be easier to rewrite it rather than port the Sprite LFS code. This meant that distribution of Sprite LFS prototype to other sites for evaluation and use was not possible.

The biggest impact of not being able to distribute Sprite LFS to other sites is that it requires much more effort on the part of companies to get a running prototype on which experiments could be run. Not being portable to a standard platform such as Unix makes the transfer of technology harder.

8.2. Future directions

The success of the log-structured file system ideas presented in this dissertation opens many possible directions for future research. This future research can be classified into three areas. The first is research that continues to investigate the characteristics of the current log-structured file system. The second is research that extends the log-structured file system to support new features or different workloads. Finally, the different disk layout of the log-structured file system means that much of the analysis done on traditional storage managers needs to be repeated using the new data layout.

8.2.1. Further examination of Sprite LFS

Although this dissertation contains much insight into the workings of log-structured file systems, there are still many areas that need to be examined further. One such further study is on the read performance of log-structured file systems. The work presented in this dissertation focussed primarily on the write throughput performance of file systems. Although this is an important metric it is important to characterize the read performance with the same level of detail. It would be interesting to extend the study to include performance metrics such as the latencies suffered by read requests that cause misses in the file cache.

The work presented in this dissertation suggested that the read performance of log-structured file systems would be good for the office-engineering environment. It would be interesting to evaluate the read-performance in other environments as well. This would mean evaluating the effects of the logical versus temporal locality on the read performance of different workloads. Analysis would include the amount of variance in response time introduced by the log writing and cleaning mechanism of Sprite LFS.

Another interesting study would be of the index structure used to locate files and blocks of files. Since an index is queried on every read and updated during every write and clean operation, the structure used is important for both read and write performance. Rather than use the same index structure as Unix, different index structures such as extents could improve performance. A study is needed to determine the best index structure to use in a log-structured file system.

Finally, the read performance of a log-structured file system can be controlled to some extent by the segment layout code of the file system. A study needs to be done to determine the effects of different segment layout policies on read performance. For example, should the inodes of files be placed with the file or in a block by themselves?

The study of segment cleaning presented in this dissertation also leaves room for follow on research. Several of the cleaning policies were not fully investigated. Policies for starting and stopping segment cleaning are two examples. Another interesting study would be the use of segment cleaning to reorganize data for higher read performance. This could effect both the segment selection policy as well as the policy controlling the recombinations of the data being cleaned.

Other studies are needed on the segment cleaning mechanism itself. Possibilities such as variable sized segments and using a bitmap to track live blocks need to be investigated. Both of these possibilities could lead to lower cleaning cost by reducing the amount of data read or written during segment cleaning.

The recovery system for log-structured file system needs to be further studied. The tradeoff between recovery time and writing of indexes to the log should be examined in detail. For example, the writing of the file index could be delayed to reduce the amount index writes in some environments.

A final possible direction of study for log-structured file systems is a comparison of Sprite LFS with more traditional logging techniques such as found in Echo and Episode. It would be interesting to characterize the performance of these systems under different workloads. The amount of log space and bandwidth needed and the total performance of the systems could be compared.

It would also be interesting to do a thorough examination of LFS techniques applied to other application areas that have small objects being modified. One particular area of interest is transaction processing data base systems. Workloads from these systems tend to have small non-sequential modifications that are well suited for the log-structured file system sequential format. Preliminary studies of transaction processing on log-structured file system show promise.

8.2.2. Extensions to the log-structured file system

There are several open research areas covering extensions to the log-structured file system design presented in this thesis. The non-update-in-place feature of the system makes in possible to integrate several desirable features. Among these features are time travel, transactions, compression, and read-erase-write storage.

Time travel is when the file system allows the user to observe the state of the file system some time in the past. For example, the user could request to see the contents of a directory as it was on the same day the year before. This could be implemented in a log-structured file system by adding a time index and not cleaning segments until the data exceeds a certain age.

The time travel ideas also suggests a convenient way to integrate a log-structured file system with a storage archive. During segment cleaning the old data could be copied to the archive and kept around forever. The time travel feature allows the user to find and retrieve data from the archive.

Not updating the file system in place simplies the implementation of a transaction processing system. Since the previous version of the data is still around after the new version is written to the log, a transaction processing system can rollback aborted transactions by simply updating the index. The segment cleaner would need to be modified to retain data blocks that are parts of uncommitted transactions.

It is also possible to compress the log of the log-structured file system as it is being written to disk[65]. Compression can raise both the transfer bandwidth and the space efficiency of the file system. The append-only structure of the log-structured file system allows large blocks of data to be compressed together. This increases the benefits achieved using compression.

Furthermore, the log-structured file system can easily handle some of the problems caused by compression. For example, one problem with compression on traditional storage managers is that files can grow when updated because the effectiveness of the compression can change with updates. Since the log-structured file system can allocate space in the log in small units this changing of size does not cause problems.

Finally, the log-structured file system maps well to storage devices that require erasing before writing. Read-write optical drives and Flash RAM are examples of such devices. A log-structured file system

modified to perform the erasing during segment cleaning can use these devices efficiently. This is not true for traditional storage managers.

8.2.3. Other research

The on-disk format of the log-structured file system is sufficiently different from existing storage managers that some of the mechanisms and policies of other parts of the file system need to be re-examined. One example of this is the effectiveness of read-ahead on the disk. Much is known about the benefits of read-ahead on traditional storage managers. It is an open question as to how much read ahead should be done on log-structured file systems.

Another interesting issue to reexamine is the block size of the file system. A log-structured file system with a large block size would have less problems with the sequential re-read after a random write problem described in Section 6.2.1.2. The tradeoff would be that the system might have to write much more data to disk under workloads with small updates to files. This study would be different than the studies of block size in traditional storage managers. Traditionally studies of block size involve the tradeoff between transfer rate and fragmentation.

The write access pattern of log-structured file systems is different enough from traditional storage managers that the algorithms used for disk scheduling need to be reexamined. Since all write requests are sequential in a log-structured file system, normal disk scheduling is not needed. Furthermore, the designs of disk controllers should be updated to support the large accesses of the log-structured file systems.

8.2.4. Summary

This dissertation generated many more questions than it answered, however the questions that were answered are of significant importance. Log-structured file systems were shown to be a viable technology that will enable RAIDs to be efficiently used by workloads with small modifications. Also presented was the cost-benefit segment selection policy that enable the log-structured file system to exploit locality to further improve performance. Finally, it provides a software technology to help magnetic disk storage track the improvements in CPU technology so future systems can achieve higher performance.

CHAPTER 9

Bibliography

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 Systems 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 Mason, Shu-Tsui Tu, and Edward R. Zayas, ‘‘DEcorum File System Architectural Overview,’’ *Proceedings of the USENIX 1990 Summer Conference*, pp. 151-164 (June 1990).
3. 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).
4. 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).
5. Marshall K. McKusick, ‘‘A Fast File System for Unix,’’ *Transactions on Computer Systems* **2**(3) pp. 181-197 ACM, (1984).
6. Dennis Ritchie and Ken Thompson, ‘‘UNIX time-sharing system,’’ *Bell System Technical Journal* **57**(6) pp. 1905-1929 (July 1978).
7. David A. Patterson, Garth Gibson, and Randy H. Katz, ‘‘A Case for Redundant Arrays of Inexpensive Disks (RAID),’’ *ACM SIGMOD 88*, pp. 109-116 (June 1988).
8. John K. Ousterhout and Fred Douglis, ‘‘Beating the I/O Bottleneck: A Case for Log-structured File Systems,’’ UCB/CSD 88/467, Computer Science Division (EECS), University of California, Berkeley, Berkeley, CA (Oct 1988).
9. R. Daley and P. Neumann, ‘‘A General Purpose File System for Secondary Storage,’’ *Proceedings of the AFIPS Fall Joint Computer Conference*, pp. 213-229 (1965).
10. Lawrence J. Kenah and Simon F. Bate, *VAX/VMS internals and data structures*, Digital Press, Bedford, Mass (1984).
11. Marshall Kirk McKusick, William 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).
12. Alan Jay Smith, ‘‘Disk Cache-Miss Ratio Analysis and Design Considerations,’’ *ACM Transactions on Computer Systems* **3**(3) pp. 161-203 (1985).
13. Michael N. Nelson, ‘‘Physical Memory Management in a Network Operating System,’’ UCB/CSD 88/471, Computer Science Division (EECS), University of California, Berkeley, Berkeley, CA (Nov 1988).
14. James G. Thompson, ‘‘Efficient Analysis of Caching Systems,’’ UCB/CSD 87/374, Computer Science Division (EECS), University of California, Berkeley, Berkeley, CA (1987).
15. Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout, ‘‘Measurements of a Distributed File System,’’ *Proceedings of the 13th Symposium on Operating Systems Principles*, pp. 198-212 ACM, (Oct 1991).
16. Jim Gray, ‘‘Notes on Data Base Operating Systems,’’ in *Operating Systems, An Advanced Course*, Springer-Verlag (1979).

17. Robbert van Renesse, Andrew S. Tanenbaum, and Annita Wilschut, ‘‘The Design of a High-Performance File Server,’’ IR-178, Vrije Universiteit Amsterdam (Nov 1988).

18. Philip D. L. Koch, ‘‘Disk File Allocation Based on the Buddy System,’’ *Transactions on Computer Systems* **5**(4) pp. 352-370 (1987).

19. Mike Powell, ‘‘The DEMOS File System,’’ *Proceedings of the 6th Symposium on Operating Systems Principles*, pp. 33-42 ACM, (Nov 1977).

20. Larry McVoy and Steve Kleiman, ‘‘Extent-like Performance from a UNIX File System,’’ *Proceedings of the USENIX 1991 Winter Conference*, (Jan 1991).

21. Margo I. Seltzer, Peter M. Chen, and John K. Ousterhout, ‘‘Disk Scheduling Revisited,’’ *Proceedings of the Winter 1990 USENIX Technical Conference*, (January 1990).

22. Carl Staelin, ‘‘High Performance File System Design,’’ CS-TR-347-91, Princeton Technical Report, Princeton, NJ (October 1991).

23. Jim Gray, ‘‘A Census of Tandem System Availability Between 1985 and 1990,’’ TR 90.1 (Part Number 33579), Tandem Computers Inc., Cupertino, CA (January 1990).

24. William H. Paxton, ‘‘A Client-Based Transaction System to Maintain Data Integrity,’’ *Proceedings of the 7th Symposium on Operating Systems Principles*, pp. 18-23 ACM,, (1979).

25. M. Fridrich and W. Older, ‘‘The Felix File Server,’’ *Proceedings of the 8th Symposium on Operating Systems Principles*, pp. 37-46 ACM, (1981).

26. M. Schroeder, D. Gifford, and R. Needham, ‘‘A Caching File System for a Programmer’s Workstation,’’ *Proceedings of the 10th Symposium on Operating Systems Principles*, pp. 25-34 ACM, (1985).

27. Mark R. Brown, Karen N. Kolling, and Edward A. Taft, ‘‘The Alpine File System,’’ *Transactions on Computer Systems* **3**(4) pp. 261-293 (1985).

28. A. Chang, M. F. Mergen, R. K. Rader, J. A. Roberts, and S. L. Porter, ‘‘Evolution of storage facilities in AIX Version 3 for RISC System/6000 processors,’’ *IBM Journal of Research and Development* **34**(1) pp. 105-109 (Jan 1990).

29. 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).

30. G. Amdahl, ‘‘Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities,’’ *Proc. AFIPS Spring Joint Computer Conference*, (Apr 1967).

31. Peter M. Chen, ‘‘An Evaluation of Redundant Arrays of Inexpensive Disks using an Amdahl 5890,’’ UCB/CSD 89/506, Computer Science Division (EECS), University of California, Berkeley, Berkeley, CA (May 1989).

32. M. Y. Kim, ‘‘Synchronized Disk Interleaving,’’ *IEEE Transactions on Computers* **C-35**(11) pp. 978-988 (Nov 1986).

33. Control Data Corporation, *Product Specification for Wren IV SCSI Model 94171-344*, Control Data OEM Product Sales, Minneapolis MN (Jan 1988).

34. Ethan Miller, ‘‘Input/Output Behavior of Supercomputing Applications,’’ UCB/CSD 91/616, Computer Science Division (EECS), University of California, Berkeley, Berkeley, CA (January 1991).

35. O. G. Johnson, ‘‘Three-Dimensional Wave Equation Computations on Vector Computers,’’ *Proceedings of the IEEE* **72**(1)(January 1984).

36. M. Satyanarayanan, ‘‘A Study of File Sizes and Functional Lifetimes,’’ *Proceedings of the 8th Symposium on Operating Systems Principles*, pp. 96-108 ACM, (1981).

37. 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).

38. Garth A. Gibson, ‘‘Redundant Disk Arrays: Reliable, Parallel Secondary Storage,’’ UCB/CSD 91/613, Computer Science Division (EECS), University of California, Berkeley, Berkeley, CA

(December 1990).

39. Keith Muller and Joseph Pasquale, ‘‘A High Performance Multi-Structured File System Design,’’ *Proceedings of the 13th Symposium on Operating Systems Principles*, pp. 56-67 (Oct 1991).

40. Venkat Rangan and Harrick Vin, ‘‘Designing File Systems Digital for Video and Audio,’’ *Proceedings of the 13th Symposium on Operating Systems Principles*, pp. 81-95 (Oct 1991).

41. Ramesh Govindan and David Anderson, ‘‘Scheduling and IPC Mechanisms for Continuous Media,’’ *Proceedings of the 13th Symposium on Operating Systems Principles*, pp. 68-80 (Oct 1991).

42. ANSI, ‘‘Common Command Set (CCS) of the Small Computer System Interface (SCSI),’’ X3T9.2/85-52, ANSI Standard Committe of X3 (1989).

43. Robert B. Hagmann, ‘‘A Crash Recovery Scheme for a Memory-Resident Database System,’’ *IEEE Transactions on Computers* **C-35**(9)(Sep 1986).

44. Steve Kleiman, ‘‘Vnodes: An Architecture for Multiple File System Types in Sun UNIX,’’ *Proceedings of the Winter 1986 USENIX Technical Conference*, pp. 238-247 (January 1986).

45. John K. Ousterhout, ‘‘Why Aren’t Operating Systems Getting Faster As Fast as Hardware?,’’ *Proceedings of the USENIX 1990 Summer Conference*, pp. 247-256 (June 1990).

46. Carl Staelin and Hector Garcia-Molina, ‘‘Clustering Active Disk Data To Improve Disk Performance,’’ CS-TR-283-90, Princeton Technical Report, Princeton, NJ (September 1990).

47. Brent Ballinger Welch, ‘‘Naming, State Management, and User-Level Extensions in the Sprite Distributed File System,’’ UCB/CSD 90/567, Computer Science Division (EECS), University of California, Berkeley, Berkeley, CA (April 1990).

48. Liba Svobodova, *Management of Object Histories in the Swallow Repository*, MIT Laboratory for Computer Science, Cambridge, MA (1980).

49. Liba Svobodova, ‘‘A Reliable Object-Oriented Data Repository for a Distributed Computer System,’’ *Proceedings of the 8th Symposium on Operating Systems Principles*, pp. 47-58 ACM, (1981).

50. 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).

51. Ross S. Finlayson and David R. Cheriton, ‘‘Log Files: An Extended File Service Exploiting Write-Once Storage,’’ *Proceedings of the 11th Symposium on Operating Systems Principles*, pp. 129-148 ACM, (Nov 1987).

52. Jason Gait, ‘‘The Optical File Cabinet: A Random Access File System for Write-Once Optical Disks,’’ *IEEE Computer* **21**(6) pp. 11-22 (1988).

53. Terry Laskodi, Bob Eifrig, and Jason Gait, ‘‘A UNIX File System for a Write-Once Optical Disk,’’ *Proceedings of the USENIX 1988 Summer Conference*, pp. 51-60 (1988).

54. H. G. Baker, ‘‘List Processing in Real Time on a Serial Computer,’’ A.I. Working Paper 139, MIT-AI Lab, Boston, MA (April 1977).

55. 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).

56. Henry Lieberman and Carl Hewitt, ‘‘A Real-Time Garbage Collector Based on the Lifetimes of Objects,’’ *Communications of the ACM* **26**(6) pp. 419-429 (1983).

57. Benjamin Zorn, Paul Hilfinger, Kinson Ho, and James Laarus, ‘‘SPUR Lisp: Deisgn and Implementation,’’ UCB/CSD 87/373, Computer Science Division (EECS), University of California, Berkeley, Berkeley, CA (September 1987).

58. R. Peterson and J. Strickland, ‘‘LOG Write-Ahead Protocols and IMS/VS Logging,’’ *Proceedings of the Second ACM SIACT-SIGMOD Symposium on Principles of Database Systems*, pp. 216-243 (March 1893).

59. Brian M. Oki, Barbara H. Liskov, and Robert W. Scheifler, ‘‘Reliable Object Storage to Support Atomic Actions,’’ *Proceedings of the 10th Symposium on Operating Systems Principles*, pp. 147-159 ACM, (1985).

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

61. Klaus Elhardt and Rudolf Bayer, ‘‘A Database Cache for High Performance and Fast Restart in Database Systems,’’ *ACM Transactions on Database Systems* **9**(4) pp. 503-525 (December 1984).

62. Kenneth Salem and Hector Garcia-Molina, ‘‘Crash Recovery Mechanisms for Main Storage Database Systems,’’ CS-TR-034-86, Princeton University, Princeton, NJ (1986).

63. Tobin J. Lehman and Michael J. Carey, ‘‘A Recovery Algorithm of A High-Performance Memory-Resident Database System,’’ *SIGMOD 1987*, ().

64. Jon Solworth and Cyril Orji, ‘‘Write-Only Disk Caches,’’ *1990 ACM SIGMOD International Conference on Management of Data*, pp. 123-132 (1990).

65. Michael Burrows, Charles Jerian, Butler Lampson, and Timothy Mann, ‘‘On-line Data Compression in a Log-structured File System,’’ Research Report 85, DEC SRC, Palo Alto, CA (April 1992).

The End

Table of Contents

CHAPTER 1: Introduction 1
1.1 Log-structured file systems 1
1.2 Focus of research: Unix office/engineering environment 2
1.3 Thesis Contributions 3
1.4 Relationship to Sprite and RAID 3
1.5 Outline of Dissertation 3

CHAPTER 2: Disk Storage Manager Design 4
2.1 Unix file system abstraction 4
2.2 Unix file system metrics 5
2.3 Unix file system implementation 6
2.3.1 Berkeley Unix Fast File System 8
2.4 Design techniques for high performance 9
2.4.1 Main memory techniques 9
2.4.2 Disk optimization techniques 11
2.4.2.1 Magnetic disk characteristics 11
2.4.2.2 Contiguous allocation 12
2.4.2.3 Clustering 14
2.4.3 Disk Scheduling 14
2.5 Fast Crash Recovery 15
2.5.1 Careful updates 15
2.5.2 Scavenger program recovery 17
2.5.3 Logging techniques 17
2.6 Summary 19

CHAPTER 3: Motivation 20
3.1 Technology for Disk Storage Managers 20
3.1.1 Processor technology 20
3.1.2 Disk technology 21
3.1.2.1 Disk array technology 21
3.1.3 Memory technology 22
3.2 Workloads for disk storage managers 24
3.2.1 Storage request classifications 24
3.2.2 Application areas 25
3.3 Future technology and workloads 26
3.3.1 Future technology 26
3.3.2 Future workloads 26
3.4 Research focus for the 1990s 27

CHAPTER 4: Log-structured file systems 28
4.1 Overview of log-structured file systems 28
4.2 Data location and reading 29

4.2.1 Inode map 30
4.3 Free Space Management 32
4.3.1 Threading 32
4.3.2 Copying 33
4.4 Sprite LFS - Threading and Copying 35
4.4.1 Segments 35
4.4.2 Segment cleaning 36
4.4.3 Segment selection 38
4.4.4 Segment layout 39
4.4.5 Log regions and segment summary blocks 39
4.4.6 Log region layout 40
4.5 Crash recovery 42
4.5.1 Roll forward 42
4.5.1.1 Writing policy 42
4.5.1.2 Directory operation log 43
4.5.1.3 Roll forward algorithm 44
4.5.1.4 End-of-log detection 46
4.5.2 Checkpoints 48
4.5.2.1 Synchronization 48
4.5.2.2 Checkpoint region update 49
4.6 Implementation 50
4.7 Summary 50

CHAPTER 5: Sprite LFS cleaning policies 51
5.1 Write cost 52
5.2 Methodology 54
5.3 Simulation study 54
5.3.1 Simulator implementation 55
5.3.2 Workload generator 55
5.3.3 Cleaning policies 56
5.4 Simulation results 56
5.5 Other cleaning policies 61
5.5.1 File reorganization during cleaning 61
5.5.2 Cleaning timing and amount 65
5.5.3 Segment size 67
5.6 Summary 67

CHAPTER 6: Experience with Sprite LFS 68
6.1 History of Sprite LFS 68
6.2 Synthetic workload analysis 68
6.2.1 Micro-benchmarks 69
6.2.1.1 Small file performance 69
6.2.1.2 Large file performance 72
6.2.1.3 Crash recovery 74
6.2.2 Macro-benchmarks 75

6.3 Measurements from the Sprite LFS system 76
6.4 Sprite environment 76
6.5 Cleaning overheads 77
6.6 Other overheads in Sprite LFS 79
6.7 Summary 79

CHAPTER 7: Related work 81
7.1 Log-structured storage systems 81
7.2 Main-memory storage management 81
7.3 Database storage management 82
7.4 Main-memory database systems 83
7.5 Other write-optimized storage managers 83
7.6 Contemporary disk storage managers 83

CHAPTER 8: Conclusion 85
8.1 Lessons learned about research 86
8.2 Future directions 86
8.2.1 Further examination of Sprite LFS 86
8.2.2 Extensions to the log-structured file system 87
8.2.3 Other research 88
8.2.4 Summary 88

CHAPTER 9: Bibliography 89

List of Figures

2-1. File block index format of the Unix file system 7
2-2. Disk layout of a Unix file system 8
2-3. Magnetic disk organization 11
2-4. Consistent updates using shadow pages. 16

3-1. Disk organization of RAID level 4 23

4-1. A comparison between Sprite LFS and Unix FFS 30
4-2. A threaded log-structured file system 32
4-3. A copying log-structured file system 34
4-4. Segment cleaning in Sprite LFS 36
4-5. Layout of a Sprite LFS log region 40
4-6. Multiple log writes to same segment 47

5-1. Write cost as a function of *u* 53
5-2. Initial simulation results 57
5-3. Effect of locality on write cost 59
5-4. Segment utilization distribution with greedy cleaner 60
5-5. Segment utilization distribution with cost-benefit policy 62
5-6. Write cost with cost-benefit policy 63
5-7. Benefits of age sorting 64
5-8. Effect of the amount of data cleaned 65
5-9. Effect of segment size on write cost 66

6-1. Small-file performance 70
6-2. Resource utilization during file creation 71
6-3. Prediction of future small-file performance 72
6-4. Large-file performance 73
6-5. Andrew file system benchmark results 75
6-6. Segment utilization in the /user6 file system 78

List of Tables

3-1. Technology trends effecting storage managers 21

4-1. Achievable bandwidth for different segment sizes 35

6-1. Recovery time for various crash configurations 74
6-2. Sprite LFS production file systems 77
6-3. Segment cleaning statistics and write costs for production file systems 77
6-4. Disk space and log bandwidth usage 79

Acknowledgements

Many people deserve special acknowledgements for making this dissertation possible. The foremost of these people is my advisor John Ousterhout. John provided me with the basic ideas for the log-structured file system and guided me in my research. His advising was an excellent balance between insistence on his way so as to avoid time-consuming pitfalls, and the freedom to investigate the areas that I considered interesting. I can only hope that my graduate students will be as happy with me as I was with John.

Another key person for this dissertation is my friend and soon to be wife, Diane Greene. Besides being a technical consultant for my ideas, Diane provide much emotional support. I feel that I could not have excelled in graduate school without the happiness and self-confidence that Diane instilled in me.

My research would not have been possible without the help of the other Sprite and RAID project members. I benefited greatly from the interactions with the faculty involved in these projects. Randy Katz served on my qualifying exam and reader committees. He provided insight that aided both the direction of the research and the quality of the presentation in this dissertation. Mike Stonebraker served on my qualifying exam committee and provided the challenges and criticisms that led to the simulations presented in Chapter 5. Finally, I am very grateful for David Patterson's influence on both my research direction and presentation.

I owe a great deal of thanks to my Sprite project officemates Mary Baker, Fred Douglis, and John Hartman. Besides putting up with me as an officemate, each of them contributed significantly to my research. Mary contributed technical advise and friendship. When I had a problem, technical or otherwise, talking with her usually made the solution obvious. Fred Douglis was active in the initial discussion that shaped the log-structured file system. John Hartman deserves a special thanks for showing great confidence in the log-structured file system and converting most of the production file systems to use it.

I also owe thanks to the Spriters that graduated before me, Mike Nelson and Brent Welch. They designed and built the Sprite kernel that made it both possible and relatively easy to demonstrate the log-structured file system ideas with an actual implementation. I would also like to thank the rest of the Sprite project: Mike Kupfer, Jim Mott-Smith, and Ken Shirriff.

Furthermore, I must thank all the members of the RAID project who both provided input into my research and were courageous enough to store their files on a log-structured file system. Without these users the measurements in Chapter 6 would not have been possible. Special thanks go to Garth Gibson, Edward K. Lee, Ann L. Chervenak Drapeau, Peter M. Chen, Ken Lutz, Ethan Miller, and Srinivasan Seshan.

Finally, I owe unmeasurable debt to my father and mother. They instilled in me the desire to get a PhD as well as the confidence that made it possible.

My work was funded 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.