Sawmill: A Logging File System for a High-Performance RAID Disk Array

by

Kenneth William Shirriff

B.Math. (University of Waterloo) 1986
M.S. (University of California at Berkeley) 1990

A dissertation submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor John Ousterhout, Chair
Professor Randy Katz
Professor Ronald Wolff

1995

Sawmill: A Logging File System for a High-Performance RAID Disk Array

Abstract

Sawmill: A Logging File System for a High-Performance RAID Disk Array

by

Kenneth William Shirriff

Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor John Ousterhout, Chair

The widening disparity between processor speeds and disk performance is causing an increasing I/O performance gap. One method of increasing disk bandwidth is through arrays of multiple disks (RAIDs). In addition, to prevent the file server from limiting disk performance, new controller architectures connect the disks directly to the network so that data movement bypasses the file server. These developments raise two questions for file systems: how to get the best performance from a RAID, and how to use such a controller architecture.

This thesis describes the implementation of a high-bandwidth log-structured file system called "Sawmill" that uses a RAID disk array. Sawmill runs on the RAID-II storage system; this architecture provides a fast data path that moves data rapidly among the disks, high-speed controller memory, and the network.

By using a log-structured file system, Sawmill avoids the high cost of small writes to a RAID. Small writes through Sawmill are a factor of three faster than writes to the underlying RAID. Sawmill also uses new techniques to obtain better bandwidth from a log-structured file system. By performing disk layout "on-the-fly," rather than through a block cache as in previous log-structured file systems, the CPU overhead of processing cache blocks is reduced and write transfers can take place in large, efficient units.

The thesis also examines how a file system can take advantage of the data path and controller memory of a storage system such as RAID-II. Sawmill uses a stream-based approach instead of a block cache to permit large, efficient transfers.

Sawmill can read at up to 21 MB/s and write at up to 15 MB/s while running on a fairly slow (9 SPECmarks) Sun-4 workstation. In comparison, existing file systems provide less than 1 MB/s on the RAID-II architecture because they perform inefficient small operations and don't take advantage of the data path of RAID-II. In many cases, Sawmill performance is limited by the relatively slow server CPU, suggesting that the system would be able to handle larger and faster disk arrays simply by using a faster processor.

Table of Contents

List of Figures

List of Tables

Acknowledgments

Many people have given me support, assistance, guidance, and friendship during my years in Berkeley. Foremost is my advisor, John Ousterhout, who challenged and inspired me with his enthusiasm and insight into systems research. I am lucky to have had such a brilliant and patient advisor.

I would also like to thank the other members of my committe, Randy Katz and Ronald Wolff, for their comments and suggestions. Professors John Stallings and David Patterson were on my qualifying exam and provided sound advice.

My parents and family provided me with constant love and encouragement during my thesis. My father has my surprised gratitude for reading my entire thesis.

I've had the good fortune to work with many members of the Sprite group: Mendel Rosenblum, Brent Welch, Fred Douglis, Mary Baker, Bob Bruce, John Hartman, Mike Kupfer, and Jim Mott-Smith. Brent Welch was very patient in starting me with Sprite and explaining the Sprite file system to me. Members of the Berkeley RAID group gave me assistance with the RAID-II hardware: Peter Chen, Ann Chervenak, Ethan Miller, Ed Lee, Srini Seshan, and especially Ken Lutz. Theresa Lessard-Smith, Bob Miller, and Kathryn Crabtree provided much-needed help with the administrative side of Berkeley.

My officemates were a constant source of information, answers, and support; John Hartman, Mary Baker, and Kim Keeton in particular made Evans and Soda Halls fun and interesting places to work.

Graduate school wouldn't be the same without the friends I had. The members of the Hillegass House: John Hartman, Steve Lucco, Ramon Caceres, Mike Hohmeyer, and Will Evans provided much of the fun of grad school, as did friends such as Randi Weinstein, Bob Boothe, Alison Kisch, and the members of the Haste Street Players. Renny Lucas deserves special mention for her friendship.

I was supported by an IBM Graduate Fellowship, as well as by the California MICRO program, ARPA grant N00600-93-C-2481, NASA/ARPA grant NAG-2591, and NSF grants IRI-9116860, CCR-8900029, and MIP 87/5235.

1 Introduction

One key problem in operating systems is how to provide fast and efficient access to files. To provide high-bandwidth access, file systems can take advantage of two recent developments in storage system technology. First, disk arrays have been introduced to provide high-bandwidth storage. Second, to move data efficiently, storage system controllers that move data directly to the network have been developed. These innovations create new challenges for file system design. This thesis examines how file systems can use these storage systems to provide inexpensive, high-bandwidth access to data.

The first change is the development of disk arrays to provide high-bandwidth disk storage. Disk arrays were motivated by the I/O gap between the high data demands of processors and the lower data rates that individual disks can supply [PGK88]. This gap is worsening as processor speeds continue to increase and as new applications such as multimedia and scientific visualization demand ever higher data rates. Disk arrays can solve this I/O bottleneck by using several disks in parallel to provide much higher performance than a single disk, while still remaining relatively inexpensive. By configuring the disk array as a RAID (Redundant Array of Inexpensive Disks), data storage can be made reliable despite disk failures.

The second change is the introduction of new high-bandwidth controller architectures to solve the problem of getting data to the clients rapidly. Even though disk arrays have the potential of supplying high-bandwidth data inexpensively, it can be difficult to make this data available to client machines across a network. For example, the RAID group at Berkeley built a prototype disk array storage system called RAID-I that used an array of 28 disks connected to a Sun-4 workstation [CK91]. Although the disks could provide over 20 MB/s, the bandwidth available through the system was very low, only 2.3 MB/s, mainly because the memory system of the Sun-4 file server was a bottleneck.

This file server bottleneck can be avoided by new architectures that move data directly between disks and clients through a high-bandwidth interface. As an example of such a system, the Berkeley RAID group designed a storage system called RAID-II [DSH+94]. RAID-II uses hardware support to move data directly between the disks and the network at high rates, avoiding the bottleneck of moving data through the file server's CPU and memory.

Thus, two important techniques that are likely to be a part of future storage systems are RAID disk arrays and controllers with high-bandwidth data paths. The disk array will provide the high raw bandwidth, and the controller will move the data quickly to the clients.

There are several reasons why a new file system is required to take advantage of these characteristics. First, the access patterns of current file systems don't work well with a RAID. Traditional file systems tend to perform small reads and writes, which are inefficient on a RAID. Because of parity update costs, small writes to a RAID are especially expensive. Write performance can be improved, however, by a log-structured file system (LFS) [Ros92], which writes everything into a sequential log so there are only efficient large sequential writes. As will be shown in this thesis, a log-structured file system can greatly improve performance of small random writes.

Second, a traditional file system will not take advantage of a high-bandwidth data path built into a controller. Because file systems normally move data through kernel memory, they must be redesigned to move data to the network through the controller rather than through the file server. They must also be able to use the controller memory to improve performance.

Finally, supporting very high data rates requires file systems to be much more efficient and avoid per-block overheads. Workstation file systems normally operate in blocks of 4 KB or 8 KB. While this is not a problem at low bandwidths, the overhead of moving this many blocks can cause a high CPU load that may limit performance on high-bandwidth storage systems. Thus, file systems should perform large, efficient transfers whenever possible, rather than breaking requests into small blocks.

To summarize, storage systems that use RAID disk arrays and new controller architectures are likely to increase in importance as processors continue to get faster and I/O demands increase. File systems must change to take advantage of these storage systems. In particular, they must avoid small writes in order to work well on a RAID, they must take advantage of the controller's fast data path, and they must perform large, efficient transfers.

1.1. Contributions

This thesis describes the Sawmill file system, which has been designed to provide high bandwidths by taking advantage of the RAID-II architecture. Sawmill is a log-structured file system that is able to read data at 21 MB/s and write at 15 MB/s, close to the raw disk bandwidth. In comparison, a standard file system running on RAID-II performs at less than 1 MB/s. Sawmill is designed to operate as a file server on a high-bandwidth network, providing file service to network clients. Performance measurements show that a log-structured file system performs very well when run on a RAID disk array. They also show that the high-bandwidth data path greatly improves performance.

The main contributions of this thesis are methods for a file system to work efficiently with a RAID and techniques to use a high-bandwidth data path efficiently. In order to get good write performance from the disk array, Sawmill uses a log-structured file system. In addition, Sawmill uses a new method, on-the-fly layout, for laying out write data before it goes to disk. This avoids the overheads of cache-based writes that previous log-structured file systems used. Finally, the file system batches requests together to improve read performance.

The implementation of Sawmill also uses several techniques to take advantage of the high-bandwidth data path in the RAID-II architecture. One technique is stream-based data movement, rather than cache-based data movement. This stream-based approach allows large, efficient data transfers with low overhead. A second technique is caching of metadata to reduce the overhead of copying it between the controller and the file server. Finally, the file system is designed to minimize per-block processing overheads in order to handle high data rates without creating a CPU bottleneck. While these techniques were developed for RAID-II, they are also relevant to other systems that have an I/O path that bypasses the file server memory.

1.2. Outline of the dissertation

Chapter 2 of this dissertation provides the motivation for the Sawmill file system and covers related work. The chapter first discusses disk arrays in detail. It then explains the storage system bottlenecks that motivated the RAID-II storage system. Finally, it describes methods that file systems can use to improve performance.

Chapter 3 describes the implementation and internal structure of the Sawmill file system. It explains how Sawmill attempts to preserve the potential bandwidth of the storage system and how client requests are handled with the new controller architecture. The chapter discusses the event-driven structure that Sawmill uses to minimize context switches in the kernel. Finally, it explains how Sawmill handles interactions between controller memory and a file server cache.

Chapter 4 explains how file systems can take advantage of controller memory. This chapter focuses on how reads are performed in Sawmill and the buffering mechanisms. Unlike most file systems, Sawmill design doesn't use a block cache. Instead, it moves data as streams rather than blocks. This chapter explains the motivation behind the decision to avoid a block cache and shows how Sawmill uses other methods to provide high performance.

Chapter 5 discusses the techniques that Sawmill uses to improve write performance. Sawmill uses a log-structured file system so small writes will be grouped together before going to disk. This chapter describes a new method called "on-the-fly layout" that Sawmill uses to arrange data in the log before it goes to disk. This technique avoids the use of a cache and greatly decreases the processing overhead of write layout.

Chapter 6 evaluates the performance of the file system. The chapter first shows the performance of the raw disk array and the array when configured as a RAID. It then presents the Sawmill file system performance and examines the factors that affect it. These measurements illustrate the benefits of a log-structured file system and also show the performance gain of using a hardware architecture with a fast data path.

Finally, Chapter 7 summarizes the thesis, describes future work, and presents conclusions about Sawmill and about the RAID-II architecture.

2 Background

Many computer systems require fast and inexpensive access to large amounts of data. This need has grown with the increase in applications such as scientific computing and file storage, as well as the growing importance of applications such as multimedia, which require fast access to large amounts of video data. These factors have led to heavy interest in high-bandwidth storage systems, both in academia and in industry.

The goal of a storage system is to fill this need for data by providing high-bandwidth access to data stored on mass-storage devices. Unfortunately, fast storage technologies are generally more expensive than slow technologies. Thus, it isn't cost-effective in most cases to store all the data with the fastest technology. The usual solution to this problem is to use a storage hierarchy, as shown in Figure 2-1. Most of the data is stored by a slow, inexpensive technology, while a small fraction of the data is stored by the most expensive and fastest technology. This strategy takes advantage of locality of references: most references to data will go to the subset of the data that is stored at a faster level, so most data accesses can take place at the higher speed. This thesis focuses on the disk storage level of the storage hierarchy, but also discusses the use of main memory for file caching and prefetching.

A typical disk storage system can be divided into three components: the disk subsystem, the I/O data path, and the file system, as shown in Figure 2-2. The disk subsystem provides the raw data storage. The I/O data path is the collection of buses, controllers, memories, and networks used to moved file data between disks and clients. Finally, the file system is the software that controls the layout of data on the disks, manages "metadata" such as directories and file attributes, and controls the movement of information in the I/O data path. These three components will be discussed in this thesis, with a focus on how the file system can take advantage of the disk subsystem and the I/O data path.

This chapter first discusses disk subsystem architectures, focussing on disk arrays. By using multiple disks in parallel, a disk array provides high performance and reliability. Section 2.1 describes RAID disk arrays and previous work on their use. Although disk arrays can provide high bandwidth and throughput, they are inefficient for small writes; Section 2.1 also discusses techniques to improve write performance.

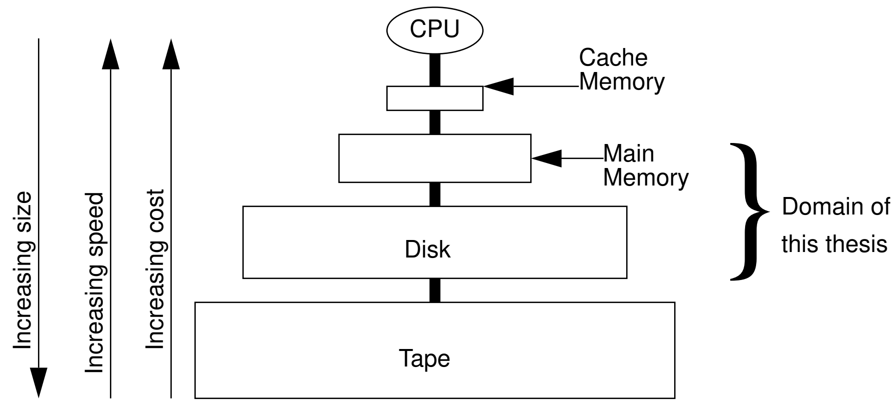


Figure 2-1. Tradeoffs in the storage hierarchy

This figure shows the elements of a typical storage hierarchy. Fast storage technologies are more expensive, leading to a hierarchy of storage systems. At the top of the hierarchy, a small fast cache provides high performance at an expensive price. At the bottom of the hierarchy, tape provides very inexpensive high-capacity storage, but performance is very poor. This thesis focuses on disk storage of data, with memory used for buffering and caching.

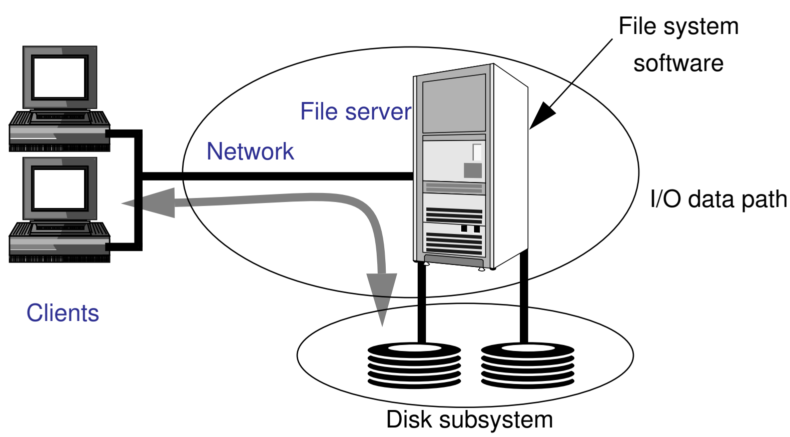


Figure 2-2. Division of the storage system into components

In this chapter, discussion is divided into three parts: the disk subsystem, which provides the raw data storage; the I/O data path, which is the hardware that moves data between the disks and the clients; and the file system, which is the software that controls data movement and organization and provides high-level abstractions to the clients.

The next sections of this chapter describe architectures for I/O data paths. The general goal of the architecture is to move data between the disks and the clients quickly and cost-effectively. Systems must provide both high bandwidth, moving a large amount of data per time interval, and low latency, handling the request with minimum delay. Section 2.2 describes various storage systems that attempt to provide efficient data movement. Section 2.3, describes the data path of the RAID-II storage system, which is used as a platform for the Sawmill file system.

Finally, this chapter discusses previous work in file systems. At the lowest level, the file system must lay out file blocks on the disk; Section 2.5 describes file layout techniques. To improve write performance, Sawmill lays out blocks using a log-structured file system; Section 2.6 explains log-structured file systems in detail. Section 2.7 discusses the use of file system techniques such as caching and prefetching to improve performance. The chapter ends with Section 2.8, which contains performance measurements, and Section 2.9, which concludes the chapter.

2.1. Disk arrays

High bandwidth storage systems put heavy demands on the raw disk storage components. As discussed in [PGK88], the increases in individual disk seek and transfer speeds have not kept up with the demands for data. Mechanical factors are the main limitation; because disks must rotate and the heads must physically move across the disk, the performance of disks is increasing only gradually. For example, positioning times are only improving by a factor of two every ten years [Sea91]. This is in sharp contrast to the fast pace of improvements in semiconductor technology, leading to a doubling in processor speeds every three years [HP90].

Disk storage system speed can be improved by using multiple disks in parallel, forming a disk array. Since the physical size of disk drives is dropping rapidly and the cost is decreasing about 20% per year [Sea91], a disk array of small disks can have lower cost and higher performance than a single expensive disk [Gib92]. Disk arrays provide two performance benefits. First, the total available bandwidth scales with the number of disks. This benefits applications that require high data rates. Second, the increased number of disks allows operations to be performed in parallel. This benefits “disk-arm limited” applications, such as transaction processing, that are limited by the time to seek to each record.

One important issue with a disk array is the layout of data. A straightforward layout technique would be to place a logically contiguous collection of data on each disk; for instance, each disk might hold a separate file system, each with a collection of directories and subdirectories. This type of layout leads to problems of space allocation and load balancing. First, because one subdirectory may grow faster than another, one disk may fill up while there is still space on the remaining disks. This requires some sort of reorganization or reallocation to use the free space. The second problem occurs when parts of the file system are accessed more heavily than others, resulting in “hot spots.” Performance will be limited by the “hot disk,” while other disks remain idle.

To avoid these problems, disk arrays usually use data striping. A logically contiguous data stream is *striped* across multiple disks. That is, the first part of the file is written to one disk, the next part is written to the second disk, and so on. The unit of interleaving is called the *stripe unit*. In fine-grain striping, the stripe unit is very small (several bytes, a single byte, or even a single bit), so every logical disk access goes to all the disks in parallel. In coarse-grain striping, the stripe unit is larger (a sector or multiple sectors), so small operations only use a single disk, while large operations use all the disks in parallel. Determining the best stripe unit size is an area of active research ([Lee93], [RB89]).

Because disk arrays increase the number of disks, the chance of a disk failure increases accordingly. To provide reliability in the event of disk failures, disk arrays are often configured with redundant storage of data, forming a Redundant Array of Inexpensive Disks, or *RAID*. There are many different ways of storing data redundantly, ranging from storing mirrored duplicates of every disk to computing parity of the stored data; the performance and tradeoffs of these techniques are discussed in [PGK88] and [Gib92].

In this thesis, coarse-grain striping is used, parity is computed across multiple stripe units, and the resulting *parity stripe units* are stored on multiple disks as shown in Figure 2-3. This technique, called RAID level 5, is used because it is efficient for file accesses [Lee93]. With N+1 disks, a *parity stripe* consists of N *data stripe units* (which hold the data) plus the associated *parity stripe unit* (which holds the parity), with one unit of the stripe stored on each disk. A parity stripe unit is computed by exclusive or-ing the corresponding data stripe units together. That is, the first byte of the parity stripe unit is the exclusive-or of the first bytes of the associated data stripe units, and so on.

For high performance, the stripe unit size should be large enough that seek time won't limit performance. That is, the time to seek to a stripe unit should be small compared to the time to read the stripe unit. For instance, a 16 KB stripe unit has a high seek overhead: reading it at 1.6 MB/s would take about 10 ms, so with a 20 ms seek time, only 1/3 of the time would be spent transferring data. Thus, systems should use a fairly large stripe unit size, such as 64 KB or larger, resulting in a parity stripe of be 1 MB or more.

The parity stripe units stored on disk are used to prevent data loss in the event of a single disk failure. To provide failure recovery a RAID disk array operates in one of three modes [MM92]: normal mode, degraded mode, or rebuild mode. If there is no disk failure, the disk array operates in normal mode, reading and writing blocks. After a disk failure, the array operates in degraded mode. To read a data stripe unit that was stored on the failed disk, the remaining data and parity stripe units in the parity stripe are read from the remaining disks. By combining these stripe units with exclusive-or operations, the desired data unit is obtained. For example, if Disk 1 in Figure 2-3 fails, Stripe unit 1 can be recovered by the exclusive-or of the remaining stripe units in the parity stripe: Stripe unit 0 $\oplus$ Stripe unit 2 $\oplus$ Parity = S0 $\oplus$ S2 $\oplus$ (S0$\oplus$S1$\oplus$S2) = S1. Thus, no data is lost and accesses to the data can continue uninterrupted. Accessing data that was stored on the failed disk is expensive, however, since N disk operations are required to read data from the N remaining disks. After the failed disk is replaced with a blank, working disk, the array operates in rebuild mode. In this mode, reconstruct reads are used to recover the lost data and write the data to the new disk. Rebuilding the disk can happen without interrupting service.

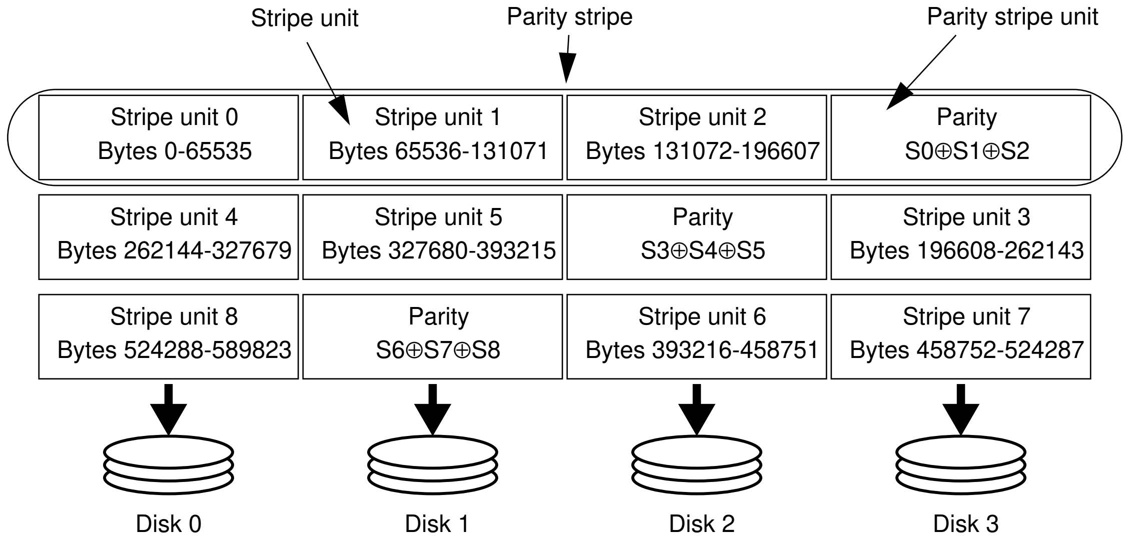


Figure 2-3. Data striping in a RAID-5.

This figure illustrates how data stripe units are striped in a configuration using 4 disks and a stripe unit of 64K. Successive parity stripe units are shifted one disk to the left; this avoids the potential “hot-spot” of a single parity disk. Data stripe units are assigned to disks in the above rotated pattern so sequential accesses will use all disks evenly.

2.1.1. Writes and parity update costs

Writes are more expensive with a RAID due to the additional overhead of computing parity. This overhead depends on the size of the operation and is worst for small writes. This section describes three different types of writes, shown in Figure 2-4, and discusses the overheads involved. For all three writes, parity must be computed as data goes to disk so the parity remains the exclusive-or of the data.

The most efficient way to write a RAID disk is the *full-stripe write*, which writes an entire parity stripe to disk at once. With this type of write, the parity is computed from the new data and written in parallel with the data. The disks are all used in parallel and the only overhead is the additional disk write for the parity stripe unit.

In a partial segment write, where less than a full parity stripe is written to disk, there is additional overhead to compute parity since it must be computed from data on disk, requiring additional disk reads. When less than half of a parity stripe is updated, the *read, modify, write* method is used. The old data in the modified stripe units is read in, along with the old parity stripe unit, and both are exclusive-or’d with the new data to yield the new parity. Due to the properties of exclusive-or, this process will yield the proper parity. The parity update is rather expensive, since each modified stripe unit results in two disk

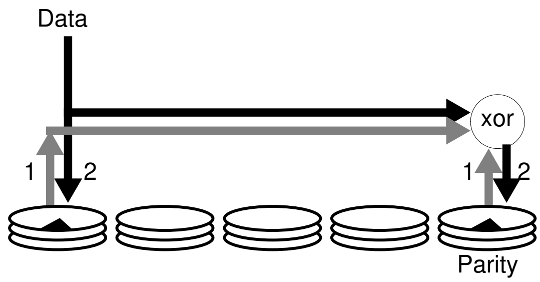
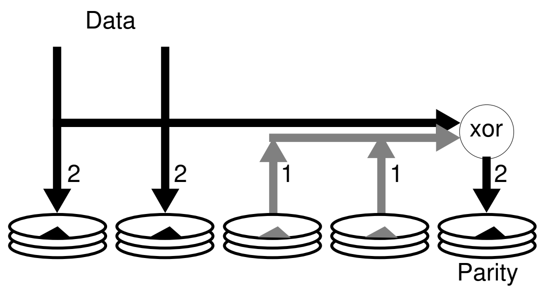
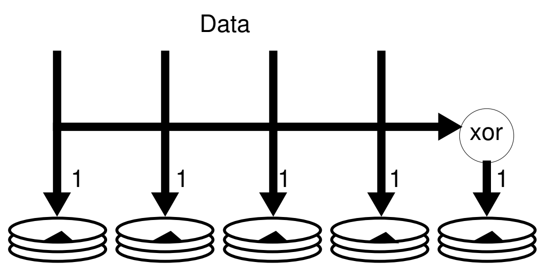
Number of stripe units (m)	Write type and data path	Disk operations	Time steps
m<N/2	Read, modify, write 	2m+2	2 (read, write)
N/2≤m<N	Reconstruct write 	N+1	2 (read, write)
m=N	Full-stripe write 	N+1	1 (write)

Figure 2-4. Three types of RAID writes

This figure illustrates the type of write used to write *m* data stripe units to *N+1* disks:
• The read-modify-write is used to update a small number of data units. First the old data and parity are read. These units are then used to compute the new parity.
• The reconstruct write is used to update many data units in a stripe. It first reads the unmodified data and computes parity from the new data and the data on disk.
• The full-stripe write is used to write an entire parity stripe at once. This write is the most efficient since data and parity can be written out in parallel. This has two benefits: first, the number of disk operations per data stripe unit written is minimized. Second, only one time step is required instead of two.

operations, plus there are two parity disk operations. In addition, the reads must take place before the writes, doubling the total time.

The third type of RAID write is the *reconstruct write*. In this write, the unmodified stripe units in the parity stripe are read in. Then the parity stripe unit is computed from the exclusive-or of the new stripe units and the unmodified stripe units. If more than half the stripe is modified, this type of write is more efficient than the read-modify-write because it doesn't access disks twice, so the total number of operations is lower. The unmodified data is read from the disk and is combined with the new data to compute the parity.

In summary, small writes are much less efficient than large writes. Figure 2-5 illustrates these costs graphically. Full-stripe writes are most efficient in two ways. First, the cost per stripe unit of the write is minimized. Second, the write can take place in one time step rather than two, since the write is not preceded by a read. Writing a single stripe unit is least efficient, requiring four disk operations, in comparison to the one operation it would require on a non-RAID disk.

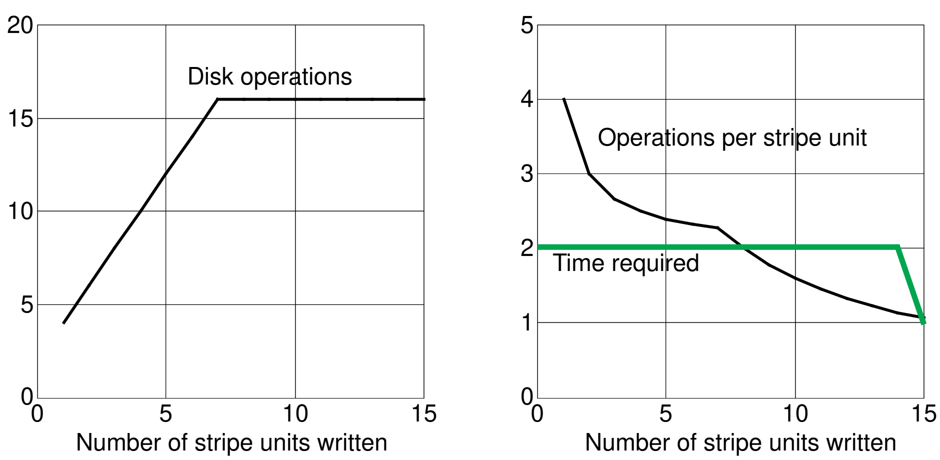


Figure 2-5. RAID write efficiency

This figure illustrates the time to write data to disk for a system with 16 disks. The left graph shows the number of disk operations required to write a specific number of stripe units. The right graph shows the number of disk operations per stripe unit and the total time required (1 time unit = the time to perform a disk operation).

2.1.2. Minimizing the parity update cost

The high cost of small writes due to parity updates is a major drawback to RAID disk arrays. This section discusses three solutions to the parity update problem. The first, floating parity, uses flexibility in parity placement to avoid the rotational cost of updating parity. The remaining techniques use various forms of logging to collect random updates into a log which can be written with large efficient writes. A fourth technique, the log-structured file system, will be discussed in Section 2.7.

Floating parity [MRK93] reduces the cost of writing parity to disk by combining the read and write into a single disk operation. The idea is that conventional parity updates are slow because the old parity is read and then rewritten to the same location, requiring the disk to wait an entire rotation. If the parity can be rewritten in a new rotationally-optimal position, the write can take place immediately after the read, with little more cost than a single read or write operation. In the "floating data and parity track" implementation of floating parity, cylinders on disk are assigned to either parity or data and each cylinder has one track left free. This free space allows data and parity updates to written to a rotationally optimal free block, rather than to a fixed position that causes a missed rotation. Thus, floating parity can perform small writes with a cost of two disk operations instead of four. Floating parity, however, results in reduced performance for larger sequential reads and writes since logically sequential blocks will now be spread out on the disk.

Parity logging [SGH93], the second technique, eliminates most of the parity update cost by delaying parity updates and then performing multiple updates as a group, allowing the use of efficient large disk operations rather than expensive random operations. Parity logging first buffers parity updates in fault tolerant memory and then writes this parity update log in large, sequential chunks. At regular intervals, large regions of the old parity are read into memory, the log is scanned, the parity updates are applied in memory, and then the new parity is written back. Thus, instead of updating parity after every write with expensive random disk operations, the parity updates only occur infrequently with inexpensive sequential operations. Parity logging has the drawback that additional disk storage is required to hold the logged parity. In addition, processing time and memory are required to apply the logged parity.

The Logical Disk [dJKH93] approach separates file management and disk management to allow efficient block placement. That is, the file system performs operations on logical disk blocks, which the disk driver maps to physical blocks through an indirection table. The driver writes all blocks sequentially to disk and uses the mapping to hide this from the file system. That is, write updates are collected into a log, which is written sequentially to disk. By performing large, sequential writes instead of random writes, parity updating can be done efficiently with full stripe writes. The Logical Disk can be considered a log-structured file system (see Section 2.5.2) implemented at the disk level rather than the file system level. The main disadvantage of the Logical Disk is that it requires a large amount of main memory for the indirection table, about 1.5 MB of memory for each gigabyte of disk. Also, like a log-structured file system, garbage collection is required to clean up free space on the disk. Finally, because the Logical Disk organizes the disk layout at the device driver level, it cannot take advantage of semantic information known to the file system that could help improve the layout, for example, by placing blocks from the same file together.

2.1.3. RAID summary

The characteristics of a RAID disk array motivate a new look at how a file system writes data to disk. If less than a parity stripe is written to disk, old data will have to be read from disk to perform the parity computation, making the write much more expensive. A parity stripe may be large, 512 KB or more, to prevent seek time from dominating performance. Thus, very large writes are required for efficiency. File systems typically process data in much smaller units: 4 KB or 8 KB file blocks are typical for a system such as NFS, and about 60% of files accessed are smaller than 4 KB [BHK+91]. Thus, there is a serious mismatch between the size of writes a file system will typically perform and the size of writes a RAID supports efficiently. Although solutions have been developed to mitigate the parity cost, they do not solve the basic problem that the file system makes small, inefficient writes.

2.2. High-performance storage systems

Given a disk system with high bandwidth, such as a RAID, the next issue is how to move data efficiently between the disks and the client applications that read or write it. This is the task of the I/O data path of the storage system. A high-performance I/O system must move data rapidly without being limited by the file server's CPU, memory, and I/O bandwidths. This task is becoming more difficult because increases in workstation processor speed are not being matched by improvements in memory bandwidth and I/O performance. For example, while older CISC machines could copy memory regions at 2.5 to 4 bytes per instruction, newer RISC machines can only copy 0.4 to 1 bytes per instruction; as a result, memory bandwidth isn't keeping up with faster processors [Ous90]. Likewise, file system performance is not scaling with processor performance [Ous90].

Several methods have been used to solve the problem of supporting higher I/O bandwidths. Mass storage systems can use an expensive mainframe, designed to support high bandwidth, as the file server. This provides a single, fast data path. Other systems use multiprocessors or multiple servers to provide high performance through multiple slower data paths. This section discusses these storage systems.

2.2.1. Mainframe mass storage systems

One way to provide high-bandwidth data access and solve the problem of the file server performance bottleneck is by using a mainframe as the file server. A mainframe server is designed to have the I/O bandwidth, memory bandwidth, and CPU power necessary to provide very high data rates to clients over a fast network.

Several very high performance mass storage systems are in use at supercomputing centers. One example is the LINCS storage system at Lawrence Livermore Laboratories [HCF+90]; this storage system has 200 GB of disk connected to a Amdahl 5868, serving Cray supercomputers across a Network Systems Corporation HYPERchannel. Another example is the Los Alamos system [CMS90], which has an IBM 3090 running the Los Alamos Common File System connected to clients by a HIPPI network and a crossbar switch. Other examples are the mass storage system at the National Center for Atmospheric Research (NCAR) [MT90] and NASA's MSS-II mass storage system [Twe90].

The main disadvantage of these mass storage systems is their cost. Because they are custom-designed and use an expensive mainframe, they are used at very few sites. Thus, the mainframe server is not a general-purpose solution.

2.2.2. Multiprocessor file servers

Another approach to increasing I/O performance is to use a multiprocessor as the file server. Such a system avoids the problem of the server being a performance bottleneck by using multiple processors, with the associated gain in server CPU power and memory bandwidth.

One example of this is the Auspex NFS server [Nel90]. The Auspex system uses asymmetric functional multiprocessing, in which separate processors deal with the components of the system: the Ethernet, files, disk, and management. The necessary disk bandwidth is provided by parallel SCSI disks. However, the performance of the Auspex is limited by its use of NFS and Ethernet; while it can saturate eight Ethernet networks with 1 MB/s each [HN90], it can supply only about 400 KB/s to a single client [Wil90].

The DataMesh project proposed a different approach to multiprocessing[Wil91]. The proposed system would consist of a large array of disk nodes, where each node had a fast CPU (20 MIPS) and 8 to 32 MB of memory. These nodes, as well as host-connection and memory nodes, would be connected by a high-performance interconnection network. The goals of DataMesh were to provide high I/O performance through parallelism, while allowing scalability and high availability. One application of the DataMesh system is the TickerTAIP [CLVW93] RAID storage system.

By providing multiple processors, these systems avoid bottlenecks from limited server CPU and memory bandwidths. However, parallelism tends to be an expensive solution, since it requires buying enough processors to provide the necessary memory bandwidth.

2.2.3. Replicated file servers

There are many file systems that replicate files across multiple servers, both for added availability and to improve performance. Replicating servers increases the total available read bandwidth, since there are more servers providing access to the data. Some systems with replication are the Andrew File System [HKM+88], Deceit [BEMS91], Echo[HBM+89] and Locus [Wal83].

Replication does not solve the problem of providing high-bandwidth I/O to a particular application, however, since a single I/O stream will still go to a single server. That is, although the aggregate read bandwidth is multiplied by the number of replicas, any particular application does not get higher bandwidth. Replication also does not solve the problem of write bandwidth; since writes are replicated, each replica of the data must absorb the entire write bandwidth.

2.2.4. Striping across servers

High bandwidth access to very large data objects can also be provided by striping files across multiple servers, so that overall bandwidth isn't limited by the performance of a single server. That is, just as a disk array stores blocks of data on different disks, data can also be stored across different servers, as shown in Figure 2-6. Then, the bandwidth for reads or writes is multiplied by the number of servers because an operation can use multiple servers in parallel.

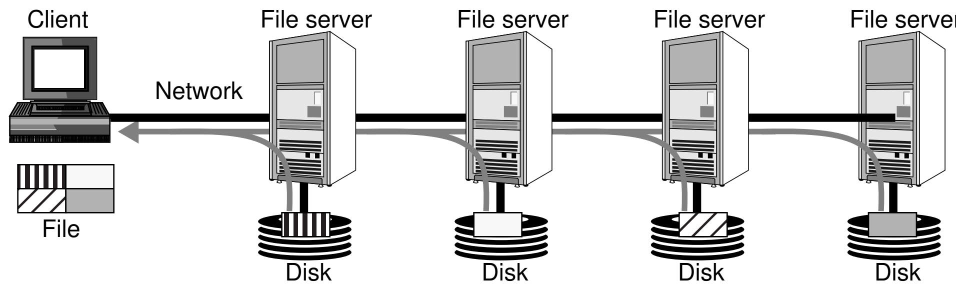


Figure 2-6. Striping across servers

This figure illustrates how a file can be striped across multiple servers. The total bandwidth is potentially much higher than a single server can provide, since the servers can operate in parallel to service a request. In this figure, the four servers are each providing part of the file to service a read.

An example is the Swift system [CL91]. In this system, data is striped across multiple file servers and networks to provide more bandwidth than a single server could provide. Swift was designed to provide high-bandwidth access to large data objects (such as multimedia video data) over a fast network, with the data stored on slow secondary storage. Swift uses a distributed architecture, with the components connected by one or more networks. This allows it to scale beyond the bandwidth of a single array controller or a single I/O channel. Simulations of Swift found that performance scaled almost linearly with the number of disks, but seek time was a limiting factor in performance. In addition, Swift doesn't provide the parallelism benefits for small objects, since they will be stored on a single server.

A second system that stripes data across multiple servers is Zebra [HO93]. In Zebra, each client writes its data to a sequential log. This log is then striped across multiple servers, each with a disk. Like Swift, Zebra increases the read and write performance by using multiple servers in parallel. By storing parity, Zebra can operate without data loss during a system failure. In addition, by combining writes into a log, Zebra provides high performance for small writes unlike Swift, which provided the parallelism benefits only for large objects.

Zebra and the Sawmill file system of this thesis both combine a log-structured file system with the parity and striping of a RAID. The key difference is that Zebra uses multiple servers each with a single disk and Sawmill uses a single server with multiple disks. Zebra avoids the file server bottleneck by using multiple servers, while Sawmill avoids the bottleneck by using a fast data path in hardware. There is a cost trade-off: Sawmill requires a special controller, while Swift and Zebra require multiple fast servers.

2.3. The RAID-II storage architecture

The Sawmill file system uses the RAID-II storage architecture [CLD+94], [DSH+94] which was designed to provide a high-bandwidth "bypass" data path. This path moves data directly between a RAID and the network, bypassing the file server's memory, so performance is not limited by the file server's data movement ability. Unlike the systems in the previous section, which increased the capabilities of the file server to prevent it from becoming a bottleneck, RAID-II provides a new hardware-assisted data path that doesn't require the file server to move data, so an inexpensive workstation can operate as the file server. Figure 2-7 is a photograph of the current RAID-II hardware setup at Berkeley. This section describes RAID-II in more detail.

The design of RAID-II was motivated by the performance of the first Berkeley RAID (RAID-I), which illustrated that a typical workstation file server could not handle the bandwidth of a disk array [CK91]. The RAID-I prototype used an array of 28 disks connected to a Sun-4/280 workstation serving as a network file server, as shown in Figure 2-8. Although the system could perform small, random I/Os quickly, the peak bandwidth available through the system was very low, only 2.3 MB/s. Although the disks could provide over 20 MB/s, the system delivered only a small fraction of this. Contention in the memory system was the main limitation on bandwidth, although the VME backplane, disk subsystem, and SCSI protocol overheads were also problems. The experience with RAID-I showed that a powerful file server CPU and many disks were not sufficient to provide high bandwidth.

To avoid the file server bottleneck, the follow-on RAID-II project moved the data path from the file server to an optimized controller that provides a high-bandwidth data path between the disks and the network, bypassing the file server. Control and data paths are separated: the file server handles requests and provides low-bandwidth control commands, while the controller board provides high-bandwidth data movement. The controller also has fast memory that can be used for buffering and prefetching.

Figure 2-9 is a block diagram of RAID-II. The controller provides the fast bypass data path between the disks and the network. It is built around a high-bandwidth crossbar switch that connects memory buffers to various functional units. These units include a fast HIPPI network interface, an XOR engine for computing parity rapidly, and the disk interfaces. Data in the system moves between the network and the memory, and between the disks and the memory. Since data blocks are moved over the high-bandwidth data path and never pass through the server, the file server does not limit data movement speed.

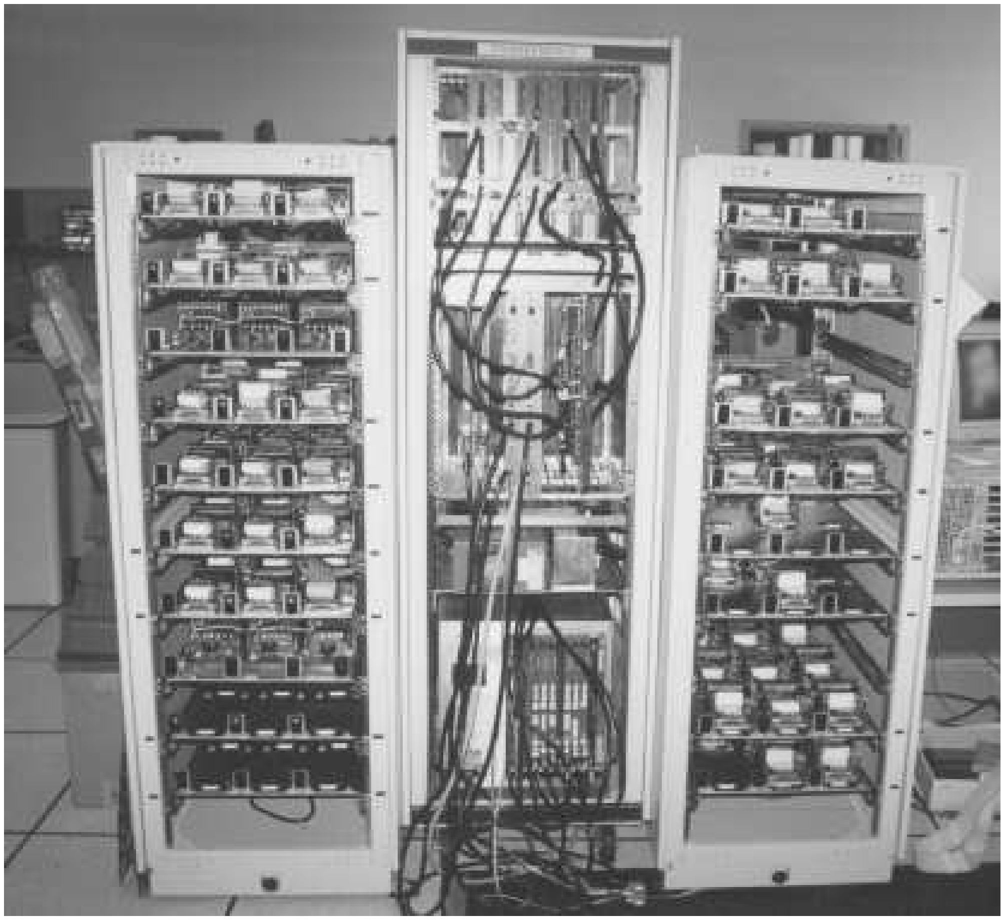


Figure 2-7. The RAID-II storage system.

This picture shows the RAID-II storage system. The left and right cabinets hold the disk drives and the disk power supplies. Each shelf holds nine disk drives, for a total of 144 drives; most are unused. The top of the center rack holds the four Interphase Cougar disk controller boards. Below them is a card cage holding the VME link boards, the HIPPI network boards, and the RAID-II board. The power supplies are below this. At the bottom is the Sun-4/280 that controls the system. This photograph is courtesy of the Berkeley RAID Group.

Because of the fast bypass data path through the interface board, RAID-II has a maximum bandwidth an order of magnitude better than RAID-I. With 30 disks, RAID-II could read at a maximum of 31 MB/s and write at 23 MB/s. Read performance was better because reads could benefit from the disks' track buffers and because writes had parity computation overhead. Further details of the design and performance of RAID-II are given in [DSH+94] and [CLD+94].

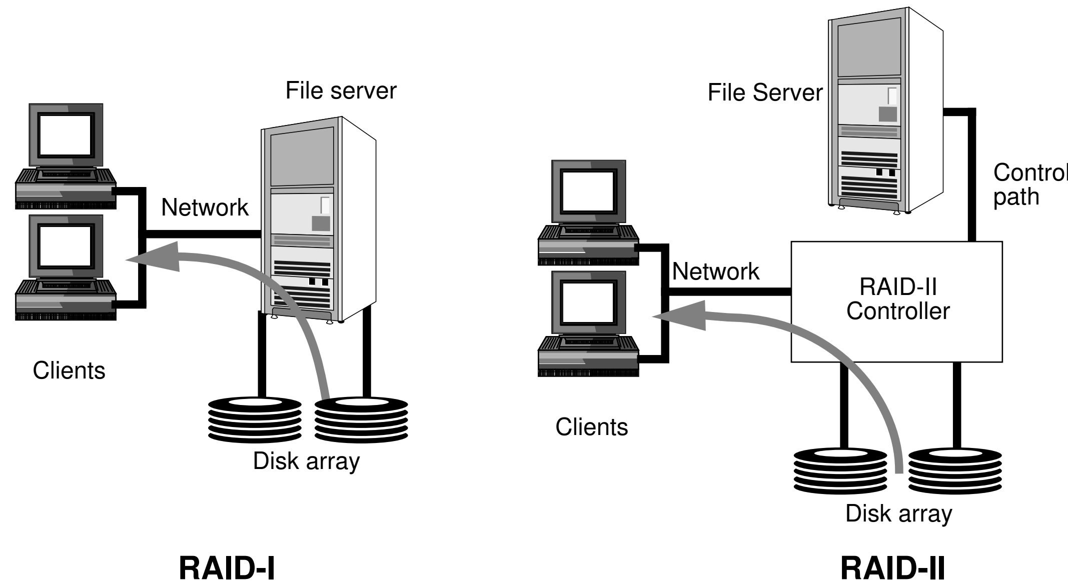


Figure 2-8. Comparison of the RAID-I and RAID-II architectures

This figure illustrates the difference between the RAID-I architecture in which data moves through the server, and the RAID-II architecture in which data moves directly through the interface without passing through the server. Performance of RAID-I was limited to 2.3 MB/s by the file server's bus bandwidth. In contrast, RAID-II can exceed 20 MB/s.

The architecture and performance of RAID-II has several influences on the design of a file system that uses it. First, the file system must take advantage of the bypass data path to the controllers. Traditional workstation file systems transfer data through the file server memory, which negates the performance benefit of RAID-II. Second, the file system can use the memory on the RAID-II controller to improve performance through techniques such as prefetching and caching. Third, the file system must access metadata over the separate control path, rather than being able to access it directly. Fourth, to make best use of the RAID, operations should be large and writes should take place as entire parity stripes. Finally, the high performance of RAID-II greatly increases the operation rate that the file system must handle, requiring the file system to be designed to minimize processing overheads to prevent a CPU bottleneck from developing.

2.4. The file system

This section discusses the file system, the third component of a storage system. The file system is the software that manages the mass-storage system and provides useful abstractions to applications. The file system converts the raw bytes stored on disk into higher-level objects such as files and directories, provides operations that can be performed on

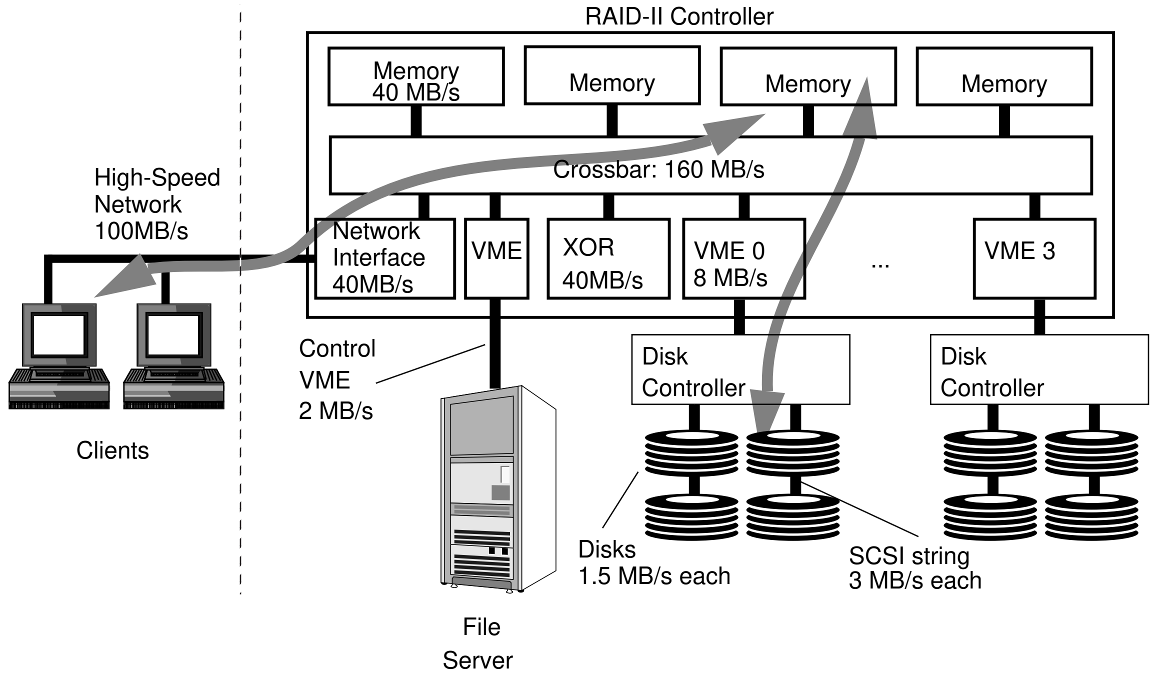


Figure 2-9. The RAID-II storage architecture.

The RAID-II storage system (on the right) provides high-bandwidth file access to the network clients (on the left). The RAID-II controller is built around a fast crossbar that connects the disks, the controller memory, and the network. The file server, running the Sawmill file system, controls the storage system over a VME bus but stays out of the data path. The current configuration uses 16 disks on four controllers. These disks are treated as a single RAID stripe group.

---

these objects, and provides access protection on these objects. Without a file system, clients can only use a storage system at a low level, reading and writing bytes on the raw disks.

There are several ways the file system can provide good performance for a storage system. One of the key issues is to prevent latencies in the file system from wasting the potential bandwidth of the storage system. That is, the file system should deliver as much performance as possible to the clients. The file system can also have positive effects on performance. At the high level, the file system can make data movement more efficient and avoid some disk accesses through techniques such as buffering, caching, and prefetching. At the low level, the file system performs file layout and disk allocation to determine where file blocks go on the disk. By performing the layout wisely, the file system can reduce seeks. The file system can also use techniques such as logging to make disk writes more efficient.

In a distributed file system, files are shared among multiple machines with access across a network. Distributed file systems typically use a client-server model, where files are stored on a file server and can then be accessed across the network by client machines. Several issues arise in distributed file systems that don't occur in a local file system. The first is distributed caching. To improve performance, files may be cached in the memory or local disk of clients; this is called client caching. In addition, files may be cached in server memory, forming a server cache. Second, because multiple copies of file data and metadata may exist on different machines, these copies must be kept consistent. That is, modifications to data on one machine should be reflected in any copies on other machines through some consistency mechanism. Finally, distributed systems have additional complexity for crash recovery, since the clients and servers may fail independently. The system should gracefully recover from crashes without losing data.

This section discusses the abstractions a file system provides and the issues involved in a distributed file system. Section 2.4.1 describes the Unix file system, which influenced much of the development of Sawmill. Sections 2.4.2 and 2.4.3 describe two distributed file systems, NFS and Sprite, and illustrates how these file systems handle consistency, caching, and crash recovery. NFS is described because of its widespread use. Sprite is discussed because it is the base for the Sawmill file system.

The following three sections also discuss file system issues. Section 2.5 discusses file layout techniques and the difference between read-optimized file systems and write-optimized file systems. Section 2.6 discusses log-structured file systems. Section 2.7 discusses methods for optimizing performance, such as caching and prefetching.

2.4.1. The Unix file system model

The Unix file system is in very wide use. Many file systems are based on the Unix model of file storage [LMKQ89], [MJLF84]. This model illustrates the types of operations that a file system must provide and also shows how the file system operates internally and on disk.

The key objects in the Unix file system are files and directories. A Unix file is a sequence of bytes with a symbolic file name. This is in contrast to file systems that give files more intrinsic structure such as internal records, multiple versions, or explicit file types. Files are organized by means of directories, which hold lists of files. Directories are stored internally as a special type of file that contains a list of file names and the associated file identifiers. Files are stored in a tree-based hierarchy of directories. The *path* of a file is a symbolic list of the directories and subdirectories traversed to reach the file. Files have a fixed collection of *attributes*, such as the creation date, the last time the file was read or modified, the owner of the file, and the access permissions on the file. Each file has an associated *inode*, which contains the attributes and the information that maps logical file blocks to physical disk blocks.

The Unix file model provides several types of operations on files. The first group of operations acts on the contents of a file. In order to be accessed, a file must first be opened. Operations can be performed on an open file such as reading bytes from the file, writing bytes into the file, appending bytes to the file, or truncating the file to a shorter length. A file is closed when no more operations are to be performed. The second group acts on the file as a whole, modifying its directory entry. Files can be created, removed from the file system, given a new name, or moved to a new directory. Finally, operations can be performed on the attributes of a file to read or modify the attributes (for instance to change the permissions).

A file system must have a mechanism to determine where a file's blocks are stored on disks. Unix accesses small files through the inode and uses multiple levels of indirection to find data blocks of large files. The inode holds pointers to the first few blocks of the file (typically 12), plus pointers to an indirect block, a doubly indirect block, and perhaps a triply indirect block. Each indirect block points to a large number of file blocks (typically 1024 or 2048), and each doubly indirect block points to a large number of indirect blocks. This scheme is illustrated in Figure 2-10. The advantage of this scheme is that it allows very large files with many blocks, but is efficient for small files since they do not require extra storage or levels of indirection.

The information that the file system stores on disk, other than file data, is called *metadata*. Directories, inodes, and indirect blocks are all types of metadata. The file system must be able to read metadata in order to perform file system operations such as looking up file names in directories and finding blocks on disk. When files are modified, the metadata must reflect these changes. In practice, metadata updates are a large part of the cost of write operations.

Many Unix implementations provide additional abstractions beyond the basic operations discussed above. Examples are file locking, which allows access to a file to be limited to a single application at a time; transaction-based file systems, which allow operations to be completed atomically with appropriate failure semantics; file mapping, which maps files into the address space of processes; and automatic compression of stored files.

Unix file systems typically store data on disk using layout techniques developed for the Fast File System (FFS) [MJLF84], which was designed to provide fast file access while reducing the amount of wasted disk space. The Unix FFS stores data on disk in units of file blocks. To avoid wasted disk storage for small files, a file block can be broken into multiple fragments. Thus, small files or the partial block at the end of a file can be stored with fragments, rather than using an entire block.

Normally, Unix interleaves blocks on disk. That is, logically sequential blocks will not be stored sequentially on disk, but will be separated by one or more other blocks. The motivation behind this is that successive reads or writes will have some processing delay between them. Since the disk will be rotating during this time, by the time the second I/O request arrives, the disk will have rotated past the start of the next block. Thus, if blocks were stored sequentially on disk, the file system would have to wait an entire revolution before it could access the next block. By skipping this block and using the next block, the file system avoids the large penalty of a rotational delay. In exchange, the file system can

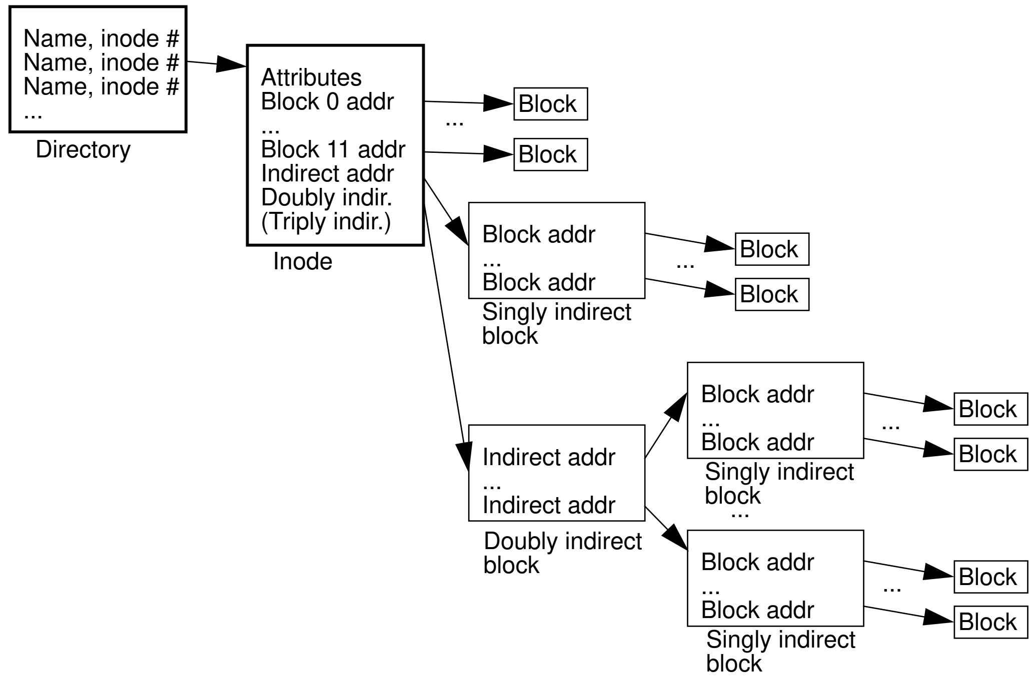


Figure 2-10. File block mapping under Unix

This figure illustrates how the disk location of a file block is determined in Unix. Every file has an inode, which holds the main information on the file. The inode is found by looking up a symbolic file name in a directory. Beside the attributes of the file, the inode holds the disk addresses of the first 12 blocks. For larger files, the inode points to a singly indirect block and a doubly indirect block. Each indirect block points to (typically) 2048 file blocks. Each doubly indirect block points to 2048 indirect blocks. With this system, very large files can be accessed and small files do not require indirection or large index tables.

---

only get a maximum bandwidth of half the raw performance the disks can provide, because it only uses half the blocks on any rotation.

To summarize, there are several key data structures that the Unix file system must manage. The file system must manage an inode and the associated attributes for each file. The inode and any indirect blocks must reference the position of file blocks on disk. The file system must lay out these blocks on disk efficiently to provide good read and write performance.

2.4.2. NFS

The Network File System (NFS) [SGK+85] [Sun89] is probably the most widely used distributed file system for Unix systems. NFS is a protocol for sharing file systems across a network, and it is designed to be portable across a variety of platforms and operating systems. With NFS, clients send requests across the network to access files that are stored on the server.

One key design decision in NFS is that it is designed to make crash recovery very easy. To accomplish this, it is *stateless*. That is, servers do not hold state information (such as which clients have files open) in their memory. For example, clients issue reads and writes without opening and closing files. A related design decision is that almost all operations are idempotent; repeating an operation more than once does not have any effect. Because NFS is stateless and idempotent, no recovery is required between clients and servers in the event of a server crash or network failure; the client can simply repeat an operation until it successfully completes.

One problem with most implementations of NFS is that performance is limited because writes of data and metadata to the server are synchronous; that is, the write goes through to disk before the reply is returned to the client. Synchronous writes are used to guarantee to clients that writes are permanent when the call returns. However, this is a performance bottleneck, as the client will block until the disk operation completes. Client caching can reduce this problem, since writes can take place to the client cache rather than going through to the server. However, to improve consistency, most NFS caching systems use write-through-on-close, which forces all data to disk when a file is closed. Performance improves somewhat since clients won't block on every write, but they will block on every close; since many files are open only a short time [BHK+91], write-through-on-close doesn't provide a large improvement. One solution to the synchronous write problem is for the server to cache write data in nonvolatile RAM, as in the Prestoserve product [MSC+90]; since writes are stored stably in nonvolatile RAM, the server can return immediately. NFS version 3 [PJS+94] eliminated the synchronous write bottleneck by providing asynchronous writes: clients write data with a WRITE command that returns immediately. Clients then do a COMMIT command to force the data to disk. This, however, requires clients to perform recovery operations in the event of a server crash to ensure the page is stored on disk.

A second problem with NFS is that file data in client caches cannot efficiently be kept consistent with the data on disk. Because the server does not have any state information on the cached copies, the server cannot inform clients if their cached data becomes stale. Instead, clients poll the server every three to 60 seconds to check if the data has changed. During this window, a client could see inconsistent data if another client has modified the data. Thus, NFS does not provide full consistency of data.

2.4.3. Sprite

Sprite is a distributed operating system designed to run on a network of workstations and provide high performance [OCD+88]. The Sprite file system provides high performance, fully consistent access to a shared file system image. The model of a Sprite cluster is a department-sized group of 10 to 50 diskless workstations connected to one or more file servers over a local area network such as Ethernet or FDDI.

The Sprite file system uses file block caching on the clients and on the servers to provide high performance. The client caches allow many file accesses to take place locally to memory without going to the disk or network. The server cache provides additional caching benefits by providing data without going to the disk. Measurements [Wel91] found that the client cache was able to serve about 60% of reads. Because many files exist for a very short time [BHK+91], the cache was also effective for eliminating write traffic to the server. By using a delayed write strategy, where files age 30 seconds before leaving the cache, the client cache was able to filter out 40 to 50% of write data; this data died in the cache and was never written through to the server.

Because of its client and server caching, Sprite has considerable distributed state, in contrast to the stateless nature of NFS. For instance, the server must keep track of which machines have cached blocks. Distributed state makes recovery after a client or server crash much more difficult since the server and clients must ensure that they have the correct values of the distributed state. For instance, after a server crash, the server must poll the clients to find out if they have cached blocks. The Sprite crash recovery mechanism is discussed in more detail in [Bak94].

Unlike NFS, Sprite maintains full file consistency; it does this through server-based cache consistency algorithms. In Sprite, all open and close requests go to the server, in contrast to NFS, which doesn’t use opens and closes. Thus, unlike NFS, the Sprite server can detect any potential sharing of files. When a file is opened for writing, any cached blocks of the file on other clients are flushed from the cache. If a file is being concurrently write shared (that is, one client has the file open for writing while another client has it open for reading and writing), then client caching is disabled and all requests are handled through the server cache. If a file is being sequentially write shared (that is, one client had the file open for writing and now another client wishes to access the file), the file server flushes any dirty cache blocks of the file back to the server so the client will obtain the most recent data. Thus, in all cases file accesses are kept totally consistent.

Figure 2-11 illustrates the structure of the Sprite file system. The Sprite file system can be split into two components: the name server and the I/O server. The name server provides operations on symbolic pathnames, such as creating and modifying directories, and creating, renaming, or removing files. The I/O server provides operations on data streams, such as reading, writing, or waiting on a stream.

The I/O server performs reads and writes through a file system block cache [Nel88]. File system requests to the cache consist of requests on cache blocks: get a block, return a block to the cache, or free a block. The cache holds individual file blocks and replaces them according to a least-recently used (LRU) discipline. The cache module calls backend modules to perform disk operations such as writing dirty blocks to disk, reading blocks from disk, or allocating space on disk.

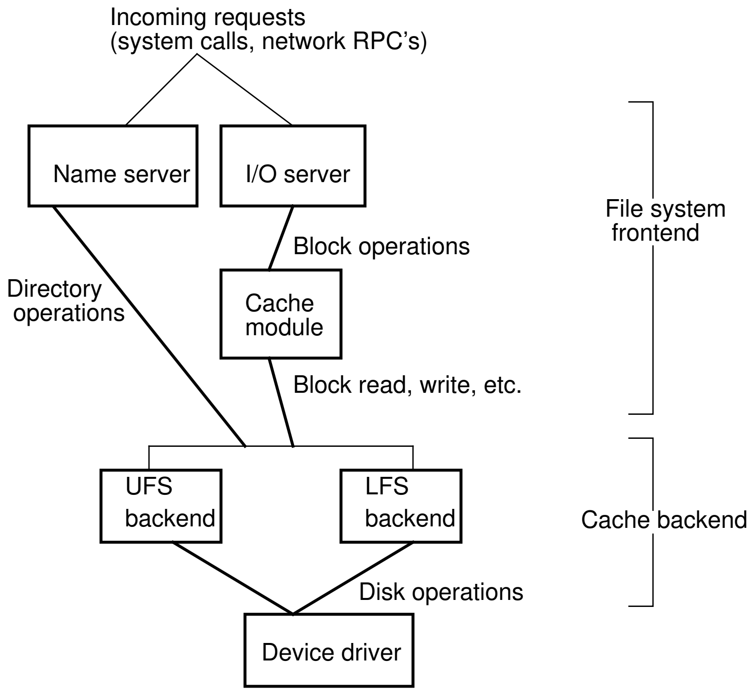


Figure 2-11. The Sprite file system

This figure shows how the Sprite operating system performs file system operations. In Sprite, name server and I/O server operations are separated. The name server provides operations on file names, such as looking up names and maintaining directory structure. The I/O server handles data operations such as reads and writes. The cache backend has different modules to provide different types of on-disk storage. Sprite stores data using a Unix-like File System (UFS) and a Log-structured File System (LFS).

The backends provide the implementations of particular disk layout methods. Each backend converts the file block operations it receives from the cache into low-level disk operations and keeps track of where files are stored on disk. Currently, Sprite has a Unix-like file system backend and a log-structured file system backend. These backends have two interfaces. The cache interface provides operations on cache blocks, such as reading blocks from disk, writing blocks to disk, and allocating blocks. The second interface provides metadata operations for the name server such as creating a file inode, modifying a file inode, and performing directory operations. These backends then call the appropriate disk device driver to perform the low-level data reads and writes.

To summarize, the Sprite file system uses a client-server architecture with caching on the clients and the servers to improve performance. It maintains consistency of shared files. The server handles file operations by passing them through a block cache, and then to a backend that performs the disk layout.

2.5. File layout techniques

A file system determines the position of data on disk through a file layout policy. That is, the file system must decide where on disk to store data to provide the best performance. File systems can arrange data on disk to favor fast reads or fast writes. A read-optimized file system attempts to store files with their blocks positioned contiguously and sequentially on disk so large sequential reads can take place quickly. On the other hand, write-optimized file systems organize data on disk to improve write efficiency, even if the layout may harm read performance. This involves techniques such as logging, where data is written to a sequential log.

Write-optimized file systems may grow in importance for several reasons. First, the increasing popularity of file system caches, both on clients and on servers, is increasing the fraction of disk traffic that is writes. A cache can filter out most of the reads, but most writes must still go to disk for reliability. Thus, disk traffic is more heavily weighted towards writes than the application traffic. One study [Wel91] found traffic to the server was 22% writes, but because of the server cache, traffic to the disk was 40% writes. Writes are likely to become even more important as cache sizes increase. A second reason for the increasing importance of write-optimized file systems is the use of RAID disk arrays, which perform much worse for small writes than for reads as explained in Section 2.1.1. Write-optimized file systems can reduce the write cost on a RAID by performing writes in large, efficient units, helping overall performance.

This section first discusses read-optimized file systems, and then write-optimized file systems. Log-structured file systems, a particular type of write-optimized file system, are described in Section 2.6.

2.5.1. Read-optimized file systems

Read-optimized file system allocation policies are designed to support fast, sequential reads through efficient file allocation. Read performance is best if files are allocated in sequential, contiguous blocks on disk to minimize seek time. Read-optimized storage allocation can be broken down into two models [Koc87]. In block-based systems, files are allocated from fixed-sized blocks. In extent-based systems, files are allocated from one or a small number of large contiguous regions of various sizes, called *extents*. This section discusses disk allocation, block-based systems, and extent-based systems. Read-optimized file systems are discussed in more detail in [SS91].

A key problem with the allocation of disk space is fragmentation. Internal fragmentation occurs when space is allocated but not used because space must be allocated in larger units than needed. For instance, if blocks are allocated in units of 8 KB, then a 1 KB file will waste 7 KB of space. External fragmentation happens when free space on disk cannot be used to satisfy a request because the free space is broken apart. For instance, if 100 KB of contiguous disk space is required but the largest contiguous regions are 90 KB, then external fragmentation prevents the request from being satisfied. Allocation with large blocks increases the problems of internal fragmentation, since blocks may have a lot of unused space. Block allocation with a uniform block size avoids external fragmentation, though, since the free space will always be in usable blocks. On the other hand, allocation as contiguous extents of varying sizes causes external fragmentation problems since the free space may not be contiguous. Extent-based allocation minimizes internal fragmentation, though, since the extent size can be selected to fit the file.

A typical block-based file system is the BSD fast file system (FFS) [MJLF84], which was discussed in Section 2.4.1. To summarize, files in BSD-FFS are allocated from fixed-sized blocks. To reduce internal fragmentation, blocks can be broken into smaller fragments. Various techniques are used to allocate related blocks close together to minimize seek distances. Sequential reads are efficient with BSD-FFS, since file blocks are stored close together.

One variant of the Unix fast file system is the extent-like Sun file system, which attempts to improve performance by grouping operations together into clusters before they go to disk [MK91]. The motivation of this design was to decrease the CPU load for high-bandwidth I/O transfers and to improve disk performance by storing blocks on disk non-interleaved (block interleaving was discussed in Section 2.4.1). With this file system, I/Os to blocks stored sequentially on disk will result in a single cluster I/O, rather than separate I/Os for each block. Typically 15 to 30 I/Os could be grouped together. The Unix file allocator was used; it attempts to allocate blocks sequentially. This system was able to get performance similar to extent-based systems for sequential reads and writes, but the modifications did not help performance of random accesses.

In extent-based systems, files are allocated from a few large contiguous extents on disk, rather than from many small blocks. Extent-based allocation has the advantage that the number of seeks required is minimized because the file will not be broken into blocks that may be scattered across the disk. A second advantage is that random allocation of blocks is easier, since the addresses of the small number of extents can all be stored in the file descriptor [Koc87]. This eliminates the need to traverse a chain of blocks or to use a Unix-style indirection table. Extents make disk allocation more difficult, however; since large, contiguous regions of free space must be found, external fragmentation can be a problem. Extents also make it difficult to grow a file, since either the file will have internal fragmentation due to space reserved for growth, or additional extents will have to be created as the file grows.

There are several examples of extent-based file systems. The FTD extent-based file system [SAS89] was designed to run on a collection of RAID disk arrays and handle transaction-processing, database, and large-object workloads. It allocates files as a collection of extents, attempting to minimize the number of extents in a file and to maximize the sizes of free extents. Small files are allocated from a separate pool of disk blocks. The Cedar file system [Hag87] allocated file pages in extents, but this resulted in external fragmentation. A newer implementation partitions free space into a “big file area” and a “small file area” to attempt to reduce fragmentation. IBM’s extent-based MVS system [IBM91] reduces the problem of growing a file by initially allocating a primary extent and then growing the file by secondary extents. The Dartmouth Time Sharing System [Koc87] allocates files from multiple extents formed with a binary buddy system to reduce fragmentation.

2.5.2. Write-optimized file systems

Write-optimized file systems are designed to improve write performance. This is typically done by performing sequential writes rather than the random writes that arise with a read-optimized disk layout. The problem is that writes often require random disk operations, which hurt performance because of disk seeks. These operations can result from updates to random file blocks. In addition, random disk I/Os may be required for metadata modifications; this overhead can be especially important for small files. One technique to avoid random disk writes is to store updates in a structure called a log, which is updated by appending data to the end of the log. Thus, the log can be updated through sequential writes, rather than random writes. The log can either be temporary storage or the permanent copy of data. This section discusses several write-optimized file systems. The write-optimized log-structured file system is described in Section 2.6. Write-optimized file systems are discussed in more detail in [Sel93].

One example of a file system that stores data into a log is the Clio logging file system in V [FC87], which provides a mechanism for efficiently writing small log entries to a write-only medium. Logging is especially suited for write-only media because logging doesn't rewrite old data. The two main issues addressed in Clio were how to provide efficient access to records in the log and how to ensure fault-tolerance of the log. The logging techniques in Clio, however, won't work for a read/write file system because they are designed for write-only media.

A large part of the cost of a write is the need to update metadata such as directories and inodes. Whenever a file is created or deleted, the directory structure must be updated to reflect this. When new file blocks are created, the inode must be updated to reflect new file block positions. These updates are expensive for two reasons. First, the changes must be forced to disk to ensure that they will be reflected after a crash. Second, metadata updates usually require a disk seek because the metadata won't be in the same place as the file data. This update will be even more expensive in systems that avoid inconsistency following a crash by making metadata updates atomic. Providing atomicity increases the cost but ensures that a failure won't leave a modification half-done. The Unix Fast File System (FFS) uses an intermediate approach; it writes metadata immediately to ensure information is written in the right order but, since it does not guarantee atomicity, expensive crash recovery checking is required.

One example of a system that uses logging to reduce metadata update costs is Cedar [Hag87]. In Cedar, the in-memory copy of metadata is updated, but is not immediately written to disk. Instead a record of the metadata update operation is written to a sequential log to provide the stable record for reliability. Since the log can be written sequentially, the update cost is much lower. In the event of a crash, the metadata updates from the log are applied to the file system to restore it to a stable state.

Logging techniques also make crash recovery faster. Disk operations may fail, for example during a system crash, leaving the file system in an inconsistent state. For example, if the system failed during file creation, the file could exist on disk but not be entered into a directory, resulting in a lost file. Solutions that involve scanning the entire file system to fix it, such as the Unix `fsck` file system consistency checker, may be very slow for large disks. Scanning the file system becomes even more of a problem with a RAID, which may have hundreds of disks. With a logging system, however, only the end of the log must be scanned after a crash to make the file system consistent. This can reduce crash recovery time from hours to seconds.

The Episode file system, part of the DECorum Distributed Computing Environment (DCE), uses logging to improve crash recovery [KLA+90]. Metadata changes are grouped into atomic transactions and written to a log. The log entries are collected and written to disk periodically (typically every 30 seconds). To recover from a crash, the active part of the log is scanned; any transactions marked as committed are completed, while uncommitted transactions are undone. Logging also improved Episode write performance because metadata writes were grouped together.

To summarize, several file systems use logging to improve write performance, to reduce metadata update cost, and to reduce crash recovery costs. By grouping together writes into large sequential units, disks spend more time writing and less time seeking. Since only the active portion of the log must be scanned after a crash, recovery is much faster than file systems that require scanning the whole disk.

2.6. The Log-structured File System (LFS)

In the Sprite log-structured file system (Sprite-LFS), developed by Mendel Rosenblum [Ros92], logging is used both for the data and for the metadata and the log is the permanent copy of the data. Like other logging file systems, the log-structured file system is designed to improve write performance by buffering all file system modifications and then writing these modifications sequentially with large transfers. The key difference between LFS and most other file systems based on logging is that in LFS, the log is the only data structure on disk, instead of being separate from the actual data storage. This eliminates all random writes, while other logging file systems only reduce the number of random writes.

Because the Sawmill file system is based on Sprite-LFS, this section describes log-structured file systems in detail. Section 2.6.1 describes how blocks are located in the log and how metadata is managed. Section 2.6.2 describes the management of free space in LFS. Section 2.6.3 discusses crash recovery in LFS.

2.6.1. Reading and writing the log

One issue in the log-structured file system is how to read data efficiently. Normally, a log is scanned sequentially; this would be unacceptable for random file system accesses. To locate file blocks, LFS uses indexing structures similar to the inodes and indirect blocks in the Unix file system. Each file has an inode that contains the file's attributes and the addresses of the first 10 blocks; additional blocks are referenced through singly and doubly indirect blocks as explained in Section 2.4.1. However, unlike Unix, where inodes are assigned fixed positions, the inodes in LFS are written to the log, so their position changes every time an inode is modified. To find the inode, a data structure called the *inode map* is used. The inode map is an array holding the mapping between *i-numbers* (the number of the inode) and the position of the inode on disk. The inode map is divided into blocks and is also written to the log. Blocks of the inode map are cached in memory to avoid performing an additional disk access to read the map. To summarize, LFS introduces an inode map, which points to the latest copy of each inode. The inode then points to disk blocks and indirect blocks, as in Unix. Thus, reads from the log can be performed efficiently, without requiring the log to be scanned.

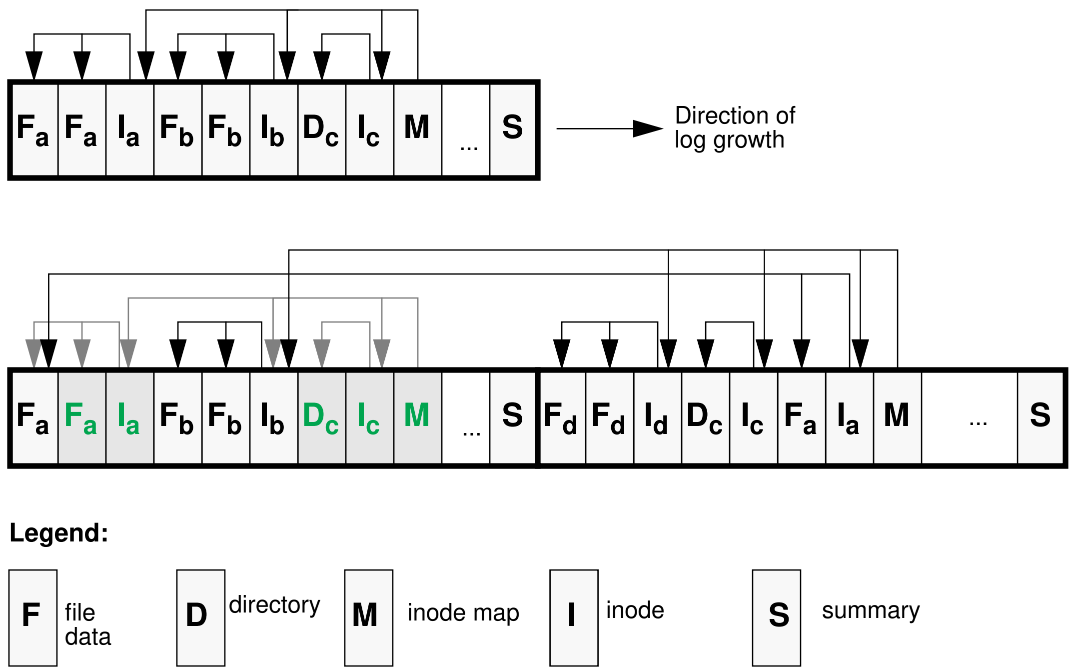


Figure 2-12. The log in a log-structured file system.

This figure illustrates the growth of the log as files are created and modified. The top diagram shows the data structures for two 2-block files (a and b) and the associated directory (c). Note that the data blocks, the updated directory, the associated inodes, and the inode map are all written sequentially to the log. New inodes are written to point to the new file and directory blocks. Then, new inode map blocks are written to point to the new inodes. To assist with cleaning, a summary block at the end of each segment of the log contains information on the log contents. The bottom diagram shows the data structures after a new file (d) has been created and a block has been modified in the first file. Note that the new inode map points to some newly-written inodes and some old inodes; it always points to the most recent copy of any particular inode. Also note that the log may contain a mixture of obsolete (dead) and in-use (live) data.

Figure 2-12 illustrates how a log is written to disk. Data, inodes, directories, and other metadata are all written sequentially to the log. As file blocks are overwritten, the new blocks are appended to the log, rather than overwriting the old blocks on disk. As a result, the log will contain “dead” blocks that are no longer in use. Thus, the LFS performs writes sequentially, even if they are to random file blocks. In addition, the log can be buffered in memory and written in large units. Thus, the log-structured file system improves write performance because it uses large sequential writes, rather than the small random writes of file systems such as BSD-FFS. The performance benefit of LFS is even larger on a RAID because the writes can take place as full parity stripe writes, avoiding the parity computation overhead of small writes.

The log also contains three special types of information. The first consists of directory log entries, which keep track of modifications to the directory structure (creations, deletions, etc.) to assist in crash recovery. For each directory modification, a record is stored in the directory log. The second is an abstraction called stable memory, which is used to maintain file system data structures in memory while ensuring they are reliably stored on disk. That is, the file system backs up some of its in-memory data structures by storing them into the disk log so they can be recovered in the event of a crash. Stable memory is used for two purposes: to keep track of segment usage information and to store file descriptors (inodes). The final type of information is the segment summary. This summary contains information describing the structure of the data in the log. The summary is used in cleaning (which will be described in the next section) to indicate what information is stored in the log.

2.6.2. Managing free space and cleaning the log

A key issue in a log-structured file system is how to manage free space. As the log grows, it will eventually fill up the disk. Unlike a database log, where only the tail of the log is active, the LFS log may contain live and dead data scattered throughout the log. Thus, there must be some mechanism to discard dead blocks and free up space so the log can continue to grow.

As discussed in [Ros92], there are several alternatives for managing free space. One alternative is threading, where the log is treated as a circular list of blocks and new blocks are written to the first free space. The disadvantage of threading is that free space will become fragmented and writes will no longer take place in large, efficient units. A second alternative is copying, where free space is obtained in front of the head of the log by moving live blocks out of the way. The disadvantage of copying is that long-lived blocks would be recopied many times during the life of the disk, wasting disk bandwidth.

LFS uses a segment-based approach to free-space management, where the log is broken into large fixed-size extents called *segments*. This can be considered a hybrid of threading and copying. Segments are threaded together to form the log; the tail of the log is written to a free segment. Individual segments are made free by copying any live data into a different segment, compacting the free space. The process of reading in segments, copying the live data back, and freeing up the segment is called *cleaning*, and is illustrated in Figure 2-13.

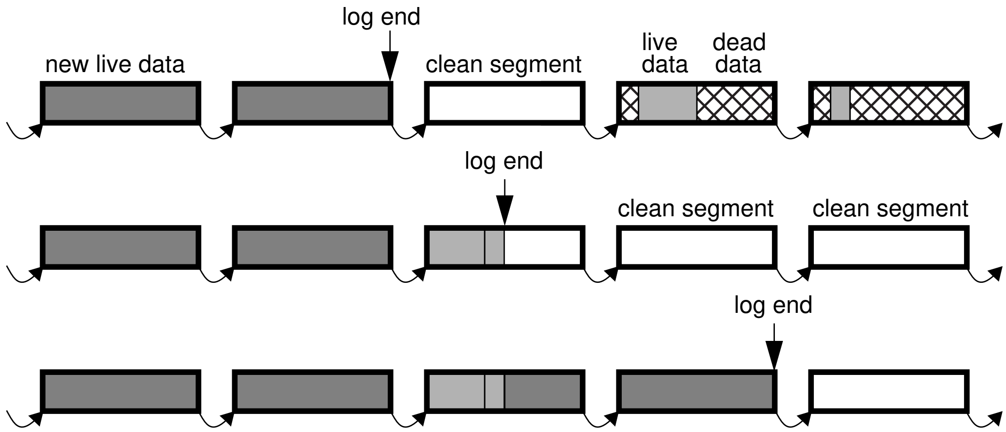


Figure 2-13. Cleaning the log

This figure shows the process of log cleaning. Each large rectangle represents a segment of the log. Dark regions indicate new blocks, light regions indicate old live blocks, cross-hatched regions indicate dead blocks, and white regions indicate free space. The top diagram shows a portion of the log before cleaning, as the log is about to fill and wrap around. The middle diagram shows the log after two segments have been cleaned. The bottom diagram shows new parts of the log written into the cleaned segments. Note that the segments in this figure are threaded together; they are typically not contiguous on disk.

The choice of cleaning policy is very important for good performance. The goal is to minimize the amount of disk traffic used for cleaning in order to maximize write performance. Several cleaning algorithms are examined in [Ros92]. An algorithm that selects segments based on their age and the amount of free space was found to perform well.

2.6.3. Crash recovery

The log-structured file system can recover very quickly from crashes because of its use of a log. It uses a checkpoint and roll-forward approach for recovery [Ros92]. At periodic intervals during use, the log-structured file system writes out a checkpoint; this is a consistent copy of the file system data structures, written to a known position on disk. To recover from a crash, the information in the checkpoint is read to restore the file system to its state when the checkpoint was written. Then the log is rolled forward from the checkpoint and the file system changes indicated in the log are applied to the file system. The result is the file system is in a consistent state that includes the changes since the checkpoint.

With a log-structured file system, the recovery time is proportional to the time between checkpoints; there is a trade-off between the time taken to write frequent checkpoints, compared to the added time to recover. The recovery time is generally very small, on the order of seconds. As explained earlier, this fast crash recovery is very important with a high-capacity RAID storage system, since a Unix-style recovery that scanned the entire file system could take hours.

2.6.4. BSD-LFS

A second log-structured file system was implemented under BSD Unix by Seltzer et al [SBMS93]; it is called BSD-LFS. It uses a cleaner process at user level, rather than in the kernel as in Sprite-LFS. This allows more flexibility in cleaning policies. BSD-LFS also uses different layout techniques; in particular, the inode map and segment usage tables were kept in a file, rather than in separate data structures. With these special structures eliminated, the disk layout and cleaning code can be simplified. BSD-LFS also changed the processing of directory operations to avoid the need for a directory log to be written to disk. To summarize, BSD-LFS uses the same basic logging techniques as Sprite-LFS, but contains several performance improvements.

2.6.5. LFS summary

The key advantage to a log-structured file system is that the log is written to disk sequentially, and it can be written in large units. This improves performance even if the file system uses a single disk, because data can be written without seeks. The performance improvement is even more dramatic on a RAID, however, because small RAID writes are comparatively much more expensive. In addition, a log-structured file system dramatically reduces the crash recovery time of the file system. For these reasons, the Sawmill file system was based on Sprite-LFS.

2.7. File system optimization techniques

This section describes techniques that can be used to improve file system performance. It first discusses caching and prefetching, which are motivated by the large gap between memory speed and disk speed. Section 2.7.1 explains caching, which holds data blocks in memory so that later accesses can use these blocks without going to disk. Section 2.7.2 discusses how prefetching starts I/O operations before the data is required to reduce the latency of disk operations. Section 2.7.3 describes other optimization techniques and discusses how they benefit different types of requests.

2.7.1. File caching

The idea behind caching is to hold file blocks in memory so file operations can take place from memory, rather than the much slower disks. Because of the relative costs of memory and disks, the memory available for a cache is much smaller than the disks, so only some of the data can be held in the cache. The result is the formation of a storage hierarchy, as discussed in the introduction to this chapter; a memory layer is added between the disks and the rest of the system [Smi85].

For reads, data blocks are stored in memory so later accesses to the same data can get the data from fast memory instead of the slow I/O system. Caching takes advantage of the locality of requests. One study found that a cache of several megabytes could service 80% to 90% of read requests [Smi85]. However, a more recent study [BHK+91] found that a large cache (typically 7 MB) only provided a 60% read hit rate, due to the increased use of large files. Thus, it is probable that cache sizes will need to continue to grow as systems use larger files.

In general, caches help more for reads than for writes because write data cached in memory is vulnerable to crashes until a copy is stored on disk. There are several alternatives for write caching, depending on when data is forced to disk. A write-through cache writes data immediately to disk, ensuring the data is reliably stored. The cache does not improve write performance but provides a benefit if the data is later read. A copy-back cache writes the data to the disk at a later time, improving performance in two ways. First, the write operation completes at memory speed, rather than disk speed, since write speed is limited by the cache. Second, the cache will eliminate a disk write if a cached block is overwritten or deleted before it goes to disk. Another caching alternative is write-back-on-close, as in NFS [Sun89]. This technique writes data to disk when the file is closed for reliability and consistency reasons, while providing some of the benefits of delaying the writeback. Write-back-on-close hurts performance compared to a copy-back cache though [SM89], since many files are open a brief time and many are deleted shortly after the file is closed.

The system must decide what data to hold in the cache in order to maximize the *hit rate*, which is the probability of a request being serviced from the cache. The hit rate is very important for performance, especially when there is a large gap between the performances of the cache and of the lower storage, as is the case for disks. If the hit rate is h, the time for a memory access is T_m, and the time for a disk access is T_d, then the average time for an access is hT_m+(1-h)T_d. Thus, as the hit rate increases, performance approaches the memory speed.

Caches typically use a least-recently-used page replacement policy (LRU) to manage which pages stay in memory. In this system, pages are kept in a list according to time of last access. Whenever a new page is to be loaded, the oldest page is flushed from memory and replaced with the new page. This scheme operates under the assumption that a recently used page is more likely to be reused than a page that hasn't been used recently. For some applications, different replacement policies will perform better; [CFL94] discusses some of these policies.

In a network file system, caching can take place on the clients as well as on the server. Client caches have the advantage of reducing network traffic and server load, since requests that are serviced by the cache do not go to the server. Server caches can perform better than client caches when files are shared among machines, since the server cache will be able to supply a shared file to all the clients while client caches would each miss on the file.

Client caching introduces consistency problems, since one client may have cached data that another client has modified, resulting in stale data. A system may use a protocol that maintains strict consistency [NWO88], or it may relax the requirement that each client see consistent data if files are write-shared. A second problem with many client caching schemes is that they result in distributed state that must be recovered after a crash [Bak94] There are several alternatives for client caching architectures. Client caches can hold file blocks, as in Sprite [NWO88], Locus [Wal83], and NFS, or whole files, as in Andrew [HKM+88], the NFSAutoCacher [Min93] and Cedar [Hag87]. This cache can also be in memory or on a local disk. The tradeoffs of client caching are explored in detail in [Nel88].

2.7.2. Prefetching

Prefetching data refers to fetching data blocks from disk into memory before they are requested. This reduces the latency of disk accesses since the disk access will be started and may be completed by the time the application requests the data. By prefetching data, an application can overlap computation and I/O, rather than performing these two tasks in sequence. In the best case, the I/O will take the same amount of time as the computation and will overlap exactly with the introduction of prefetching, resulting in a 50% speedup.

Several approaches can be used for prefetching. One simple technique of prefetching is one-block lookahead [Smi85]. This technique prefetches block $i+1$ when block i is accessed. More advanced techniques prefetch a variable number of blocks ahead, depending on the access patterns. These techniques work with sequential access patterns.

More complex techniques are possible when the references have a non-sequential access pattern. Kotz [Kot91] examined prefetching techniques for use with scientific computation workloads on parallel processors. One technique of prefetching attempts to learn working sets of files for a particular application [TD91], and found cache hit rates were substantially improved. A related technique [PZ91] uses associative memory to learn access patterns. Complex prediction algorithms can be used to predict which blocks will be accessed next, for example [GA94].

A major problem with techniques such as caching and prefetching is that the file system cannot always correctly predict the access patterns to data. A cache must predict which blocks should be discarded and which should be kept. Prefetching must predict which blocks to read. If the file system guesses wrong, it loses out on the benefits of caching or prefetching. In addition, incorrect prefetching causes unnecessary I/O to load the prefetched data, which will slow down the system. Finally, prefetching unused data can flush needed data out of the cache, harming cache performance. In the worst case, prefetching could actually reduce overall system throughput.

The technique of "hints" has been proposed to solve this problem. With hints, a mechanism is provided for the client application to inform the file system of the future access patterns, so the file system can act appropriately. The main drawback of hints is that they require changes to the application programs to notify the operating system of access patterns. One system that uses hints is called Transparent Informed Prefetching [GPS93]. In this system, the file system gains two types of knowledge from hints: information on I/O concurrency, which allows the system to process requests in parallel, and knowledge of future resource demands, which permits the system to schedule resources efficiently.

Asynchronous I/O allows a form of application-controlled prefetching. With asynchronous I/O, a client can start an operation before it requires the data. Rather than blocking until the operation completes, as with normal synchronous I/O, application execution continues after the asynchronous I/O is issued and the application is signalled when the operation completes. Thus, asynchronous I/O allows client-controlled prefetching.

2.7.3. Summary of optimization techniques

File systems can use different techniques to optimize various types of accesses. Table 2-1 summarizes these techniques, dividing requests along three axes: reads versus writes, sequential acceses versus random, and small accesses versus large. This section describes these optimization techniques.

Access type	Optimization techniques
Small random reads	Client caching, server caching
Small sequential reads	Prefetching, caching
Large reads	Prefetching, large transfers, read-optimized file system
Small random writes	Log-structured file system, delayed writeback
Small sequential writes	Log-structured file system, delayed writeback
Large writes	Delayed writeback, large transfers

Table 2-1. Optimization techniques

This table lists many of the techniques that a file system can use to speed up various types of reads and writes.

The first division of requests is into sequential or random acceses. Sequential accesses are successive references to consecutive logical blocks of a file. Sequential access patterns are very common; one study [BHK$^+$91] found that 84% of bytes were accessed sequentially. Many files are read sequentially from start to finish; this accounted for 76% of file accesses. By taking advantage of sequential access patterns, file systems can operate more efficiently. The opposite of sequential access is random access; this refers to file accesses that are not to consecutive blocks. Usually these accesses are not truly random, and some systems attempt to take advantage of structure in nonsequential access patterns (see Section 2.7.2). However, it is much harder for a file system to perform efficiently with random accesses.

Reference patterns can also be divided into small and large accesses. In this thesis, small accesses are those on the order of a single file block or several blocks. For these accesses, disk seek time dominates performance. Large accesses are those for which transfer time dominates performance. Because a RAID has very high bandwidth compared to the seek time, requests on the order of megabytes are considered large transfers. Division into sequential and random requests is not as meaningful for large transfers because a large transfer accesses a sequential stream of bytes.

For fast performance of small random reads, caching on the client or the server is the best approach. If blocks are accessed more than once, the cache will provide the blocks quickly. If the access goes to disk, there isn't much the file system can do since a disk seek will probably be required for each access. The file system can cache file metadata so it can determine the disk block access and fetch the data without a second access to fetch metadata. Prefetching will reduce performance in the small random case unless the prefetching algorithm can determine the access pattern because sequentially prefetched data will be the wrong data.

Sequential reads and large reads will benefit from the read-optimized disk layout techniques of Section 2.5.1. By placing blocks together on disk, the reads can take place without many seeks. Prefetching will also improve performance, especially if large prefetch reads take place. Large reads are an opportunity to obtain the maximum read bandwidth of the system since overheads can be amortized over many bytes.

Small random writes are a difficult problem for file systems. In a file system that assigns fixed disk addresses to data blocks, a seek will be required for each write. If less than a block is written, the old block must be read in and modified. Even worse, on a RAID, the old stripe unit and parity unit must be read in to recompute the parity. These factors make random writes very expensive. By using a log-structured file system, however, to perform the layout, small random writes will become sequential writes, avoiding the seek and the extra parity overhead. A read-modify-write problem remains, though, for writes to less than a file system block; since the entire block must be rewritten, the old data must first be read in. Small sequential writes are less of a problem than random writes, since they do not require seeks, but they will still benefit from a log-structured file system.

Large writes are fairly easy to perform efficiently, since disk seek overhead is minimized. A key factor is avoiding high per-block or per-byte overheads such as copying data in and out of the cache. In addition, writes should go to the disk in large units; if writes occur on a per-block basis, the disks won't be able to transfer data at full speed because they must skip blocks to avoid rotational latency, as discussed in Section 2.4.1.

Caching with delayed writeback will improve small write performance, since writes can be delayed before going to disk. In many cases the file will be overwritten or deleted, and the disk write won't have to take place at all.

To conclude, different techniques work best depending on the type of access. By combining these methods, file systems can work efficiently for a variety of access patterns.

2.8. Measuring performance

Several metrics are commonly used for I/O system performance. Different types of applications try to optimize different metrics. For instance, a system may wish to increase the total amount of data moved, increase the speed of a single applications, or increase the total number of operations performed. Thus, different measurements are needed to describe different aspects of the system.

This thesis will focus on three measurements: latency, bandwidth, and operation rate. Latency is a measure of the fixed overhead time associated with a request. Latency is closely related to the response time to complete a request. For small operations, latency is the most important factor. Factors affecting latency include disk seek and rotation time, queueing delays due to disk contention, and software overheads to process requests.

Bandwidth is a measure of the speed of data movement, typically measured in megabytes per second. For large requests, bandwidth is usually more important than latency because the fixed latency cost is spread over many bytes. Bandwidth is bounded by the peak ability of the system to move bytes of data quickly. (Bandwidth is sometimes called throughput [HP90].)

The third measure is the operation rate, in terms of operations per second. This measure is most important for transaction processing applications, since it is closely related to the number of transactions that can be processed per second. (The operation rate is also known as the I/O rate [HP90].) The operation rate is improved by the ability to perform multiple operations in parallel.

2.9. Conclusions

I/O performance has traditionally been a neglected area of research compared to CPU and memory system performance. However, as processor and memory speeds continue to increase, I/O is increasingly becoming a performance bottleneck. As a consequence, interest in I/O systems is steadily growing, and many techniques have been proposed for improving I/O performance through improvements in the disk storage system, the file server architecture, and the file system.

At the storage level, disk arrays can be used to provide higher bandwidths and operation rates than a single disk can provide. However, RAID disk arrays have the disadvantage that small writes are costly due to parity updates. A second problem is that seeks are very expensive compared to the transfer speed, resulting in latency being very high compared to bandwidth.

Several high-performance I/O systems have been implemented. The traditional solution is to use a mainframe with high-capacity I/O channels, but this is a very expensive solution and does not scale well. Other solutions are to provide faster I/O accesses through parallelism and special architectures that provide high bandwidth between the disks and the clients. By using an architecture such as RAID-II, the data path can be optimized without requiring the entire system to support high data rates.

Finally there are many ways the file system software can use the system efficiently to improve performance. Software techniques include efficient disk layout techniques, methods to improve the speed of disk accesses, caching to handle I/O at main memory speed, and prefetching to lower the latency of I/O requests. In particular, logging is a good match for a RAID, because logging avoids small writes.

Based on these developments, future storage systems are likely to combine several components. They will use a RAID disk array to provide high raw disk bandwidth and reliable storage. In order to move this data quickly, the controller architecture is likely to be similar to RAID-II and will provide a bypass data path to move data to the network without passing through the server. The goal of the file system is to use these components efficiently. The remainder of this thesis examines how the implementation of the Sawmill file system approaches this goal and provides high-bandwidth file data relatively inexpensively.

3 Sawmill File System Overview

"That thou doest, do quickly." – John 13:27

3.1. The Sawmill file system

This chapter gives an overview of a high-bandwidth log-structured file system that is designed to take advantage of a disk array and high-bandwidth controller. This logging file system, called Sawmill, is designed to operate as a file server on a high-speed network, providing file service to network clients. Sawmill was implemented under the Sprite operating system and uses the RAID-II storage system.

There were several goals for the Sawmill file system. The first was to implement a file system that could provide I/O rates close to that of the raw storage system, to see if a storage system architecture with a data path that bypasses the file server would perform well for file storage. Second, the implementation attempted to make use of a hardware architecture that provides a fast "bypass" data path between the disks and the network. The final goal was to combine a log-structured file system with a disk array.

The Sawmill file system demonstrates several key ideas:

- It uses a new technique, called "on-the-fly layout", to lay out the log onto disk efficiently.
- It uses new data buffering techniques to take advantage of the controller memory and it uses a stream-based approach rather than a cache-based approach.
- Sawmill reduces the cost of accessing metadata by using metadata caching since the data-path memory is separate from controller memory.
- It integrates the direct-to-network fast "bypass" data path with a slow data path through the file server, providing two separate data paths.
- Sawmill uses an event-based control flow mechanism, rather than a procedure-based model.

Most of techniques used in the Sawmill file system are not specific to RAID-II, but can be applied to related types of storage systems. The data movement and buffering of Sawmill are applicable to storage systems that provide a direct data path through the controller, bypassing the file server. Second, the log layout techniques, which will be described in Chapter 5, are applicable to log-structured file systems regardless of the data path used. They are especially appropriate for log-structured file systems running on a RAID.

Performance measurements show that Sawmill fulfills its goals of providing high performance. Sawmill reads data at up to 21 MB/s and writes at 15 MB/s, close to the raw disk bandwidth of RAID-II. Performance of Sawmill will be discussed in more detail in Chapter 6.

This chapter provides an overview and some details of the implementation of the Sawmill file system. Section 3.2 describes in more detail the motivation for the Sawmill design and the factors that affected the implementation. Section 3.3 looks at the techniques Sawmill uses to preserve bandwidth. Section 3.4 gives an overview of how Sawmill processes client requests. Section 3.5 covers how clients communicate with the Sawmill file system. Section 3.6 describes how Sawmill fits into the rest of the operating system. Section 3.7 discusses the internal structure of the Sawmill file system. Section 3.8 explains how Sawmill provides two data paths, the fast path and the slower path. Section 3.9 concludes the chapter.

3.2. Motivation behind the design of Sawmill

Several design factors influenced the Sawmill file system. First, Sawmill was intended to run on a storage system that provides a direct data path to the network. Second, Sawmill was designed to run on a RAID disk array. Finally, since the hardware available for Sawmill was the RAID-II controller, some of the design decisions in Sawmill were motivated by characteristics of this hardware. This section examines these factors, their effects on the Sawmill file system, and why a new file system was needed.

The most important factor behind the design of Sawmill was the intent to examine storage systems that have a high-bandwidth data path and explore how they interact with the file system. In particular, Sawmill was intended to run on a storage system that provides a data path between the disks and the network, through high-speed buffer memory. This data path, which will be called the *bypass data path*, provides a means of moving data to clients rapidly without moving the data through the file server's memory.

The bypass data path influenced the design of Sawmill in two main ways: data does not normally go through the server and data rates are much higher. Since most data does not pass through the file server, data that the server must access, such as file metadata, must be treated specially. Instead of accessing metadata from kernel memory, as in a traditional file system, Sawmill must explicitly copy metadata between controller memory and kernel memory.

The bypass data path also affected the file system design by providing much higher data rates. As a result, CPU load became more important than in typical file systems. For example, if a file system is based on 4-KB blocks and is transferring at 20 MB/s, a block is transferred every 200 μsec. Thus the total CPU overhead per block must be significantly less than 200 μsec to keep the CPU from being a bottleneck. As a result, the Sawmill file system was designed to minimize CPU overheads, especially per-block overheads. This resulted in the use of large block sizes in Sawmill and the implementation of faster layout techniques direct to memory. Although CPU load was especially important in Sawmill because Sawmill runs on a relatively slow Sun-4, CPU load is a general problem as it can limit the ability of the system to scale to a larger number of disks.

The presence of memory in the controller also affected the design by enabling the implementation of several features that would not be possible in an architecture that did not have buffer memory in the data path. Without buffer memory, a log-structured file system wouldn't be possible unless the clients built up log segments, since data would have to go directly to the disk without an opportunity to build up the log on the server. In addition, without the controller memory, it wouldn't be possible to improve read performance through techniques such as prefetching or caching. Thus, controller memory permits file system designs that wouldn't be possible otherwise.

To summarize, the storage system architecture with a bypass data path had many effects on the design of Sawmill. This path led to a new metadata path, new layout techniques, prefetching and buffering of reads, and avoidance of per-block overheads.

A second factor that heavily influenced the design of Sawmill was the decision to use a RAID disk array. As discussed in Section 2.1.3, disk arrays have several performance characteristics that contrast with performance of individual disks. The main properties of the disk array are the large discrepancy between read performance and write performance for small operations, the large bandwidth penalty due to seeks, and the necessity of keeping the disks busy for maximum performance. These characteristics affected the design of Sawmill in several ways.

Small writes are slow on a RAID compared to reads or large writes. This effect is due to the overhead of performing a read-modify-write cycle to update the parity after a small write. To minimize this problem, Sawmill uses a log-structured file system. A log-structured file system only performs large sequential writes, avoiding the expensive small, random writes, as discussed in Section 2.6. In addition, to take advantage of the controller architecture, the log-structured file system in Sawmill was implemented with efficient new layout techniques.

Due to the very high bandwidth of a disk array, seeks in a disk array are much more expensive in terms of lost potential bandwidth than seeks on individual disks. For example, during the 12 ms taken by a seek, an individual disk of the type in RAID-II could have transferred 19 KB; with 16 disks, the disk array could have transferred 300 KB during this time. Another way of looking at this is that 19 KB transfers will use 50% of the bandwidth of a single disk, while 300 KB transfers are required to use 50% of the disk array bandwidth. Thus, it is very important that the file system perform large transfers in order to use the disk array's potential bandwidth. As a result, the Sawmill file system attempts to group related data together in LFS segments. It also uses prefetching so multiple blocks can be read at once, rather than separating them by seeks or missed rotations.

Finally, Sawmill was designed to keep the disks as busy as possible. Because a RAID has many disks in parallel, it is much harder to keep all the disks in use. An individual disk can be kept active with a single request, while keeping the disks in a RAID busy requires either a large request striped across all the disks or multiple parallel requests. In addition, any file system dead time with a RAID will result in much more wasted bandwidth than for a single disk. To keep the disks busy, Sawmill prefetches data during otherwise idle times. It also uses pipelining for reads, so network transfers and disk operations can be overlapped, rather than having a gap between disk and network operations.

Besides the major characteristics of its server architecture, several less important characteristics of RAID-II also influenced the design of Sawmill. These factors are not as generally applicable as the previous ones, but they may still be relevant to some systems.

First, the link between the RAID-II controller and the file server CPU is fairly slow compared to the rest of this system. This motivated metadata caching in kernel memory to reduce the amount of data that gets copied between the controller and the server. With a fast link, it might be simpler to keep data in controller memory and copy it as required.

The second characteristic is the high CPU load that RAID-II puts on the server. Because the controller does not have sufficient intelligence to handle low-level disk operations, the server must issue a separate command to each disk and handle the interrupts from each disk. Coupled with the low-performance CPU in the controller, this necessitated special attention in the file system design to minimizing the CPU processing required.

Finally, the file system was designed to be integrated with the current Sprite file system, and to allow access to data through the server using standard Sprite remote procedure calls (RPCs), as well as directly over the high-speed network. Figure 3-1 illustrates these three request paths. Servicing multiple request paths introduces several problems into the design, stemming from the need to move data through kernel memory to support this. This design decision led to the implementation of a *split cache*, which will be discussed later.

To summarize, it is likely that many storage systems in the future will use architectures similar to RAID-II in order to eliminate the file server as a bottleneck. Using a RAID disk array for storage is likely to provide high data bandwidths and storage capacity. As a result, many of the ideas in the Sawmill file system are likely to be applicable to future storage systems.

There are several reasons why existing file systems are not suitable for RAID-II. These problems center around current file systems generally being designed for individual disks attached to a file server, rather than a disk array attached to a high-performance controller with a bypass data path.

First, the disk access patterns of current file systems do not work well with a RAID. Workstation file systems typically operate in small blocks, on the order of 4K. While these access patterns may be reasonably efficient for single disks, RAIDs work best with large

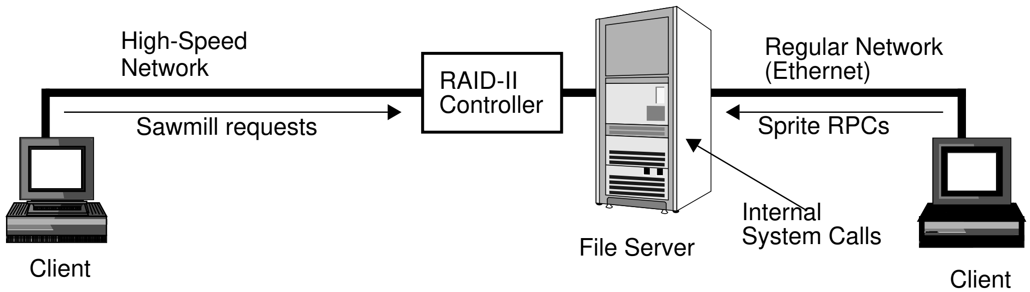


Figure 3-1. Request paths in Sawmill

The Sawmill file system handles requests over three paths. Besides the main request over the high-speed network attached to the controller, Sawmill also handles the existing request paths: the local-area network and system calls from programs running on the file server.

---

requests that can use all disks in parallel. In particular, RAIDs are very inefficient for small writes.

Second, current file systems move data through kernel memory, rather than along a data path that bypasses the kernel. The separate bypass data path in an architecture such as RAID-II architecture requires major modifications to a file system in order to move data along this path, rather than the kernel path. In addition, current file systems expect to find data in kernel memory in order to access metadata, and this won't work with the RAID-II architecture.

Finally, existing file systems don't take advantage of controller memory to improve performance. This memory can provide significant performance improvements, so a file system that ignores it will not achieve the potential of the storage system.

3.3. Bandwidth preservation

The purpose of a high-bandwidth storage architecture such as RAID-II is to avoid hardware bottlenecks. As discussed in Section 2.3, the RAID-II storage system uses a disk array to provide high raw bandwidth. The controller architecture provides a bypass data path that bypasses the file server, to avoid the file server's memory system becoming a bottleneck. A fast network, such as HIPPI, connects the storage system to the clients, preventing the network from limiting performance. Thus, the raw hardware provides very high potential raw bandwidth, about 25 MB/s for RAID-II.

The goal of the Sawmill file system is to deliver as much of this raw bandwidth as possible to the clients. There are many points in the system where the software can "waste" bandwidth. First the file system can fail to take advantage of the bypass data path; for efficiency, data movement must use the different components of the system (disks, network, memory) in parallel. In addition, poor disk layout can reduce the potential raw bandwidth. Next, CPU processing overhead can become a limiting factor. Finally, the network protocols can reduce performance.

Effective use of the bypass data path is vital to ensure that the file system doesn't waste the potential system performance. In Section 2.3, RAID-I illustrated how bandwidth can be lost due to data movement through the server. Because RAID-I moved all data through the server, the server became a performance bottleneck. By taking advantage of the RAID-II architecture, Sawmill avoids this problem.

Disk layout can waste a large amount of potential bandwidth. For instance, as discussed in Section 2.4.1, disk layout in the BSD Fast File System (BSD-FFS) typically only allocates every other block on disk for any particular file. Thus, the maximum potential read or write bandwidth is only 50% of the raw disk bandwidth. In addition, poor disk layout can result in the disks spending much of their time seeking rather than doing useful data movement.

Sawmill uses a log-structured file system to provide efficient disk layout. As explained earlier, a log-structured file system groups writes together on disk to improve write performance. This is especially important on a RAID disk array in order to minimize parity computation costs.

CPU performance can also limit performance, if the file server spends too much time processing data or performing file system operations. Sawmill minimizes performance loss due to CPU processing by trying to reduce the per-block computation required to handle a request. It does this for writes by using "on-the-fly layout" to lay out a sequence of blocks as a single operation, as will be discussed in Chapter 5. For reads, various techniques to minimize processing time to determine addresses on disk will be discussed in Chapter 4.

Once data reaches the network, bandwidth can be lost both by the network hardware and by the network protocols. Since RAID-II uses a high-bandwidth HIPPI network, network bandwidth isn't a limitation as it would be with an Ethernet. Network protocols such as NFS, however, can severely limit performance. To prevent the network protocol from being a performance bottleneck due to small block sizes and a request-response protocol, Sawmill uses a simple socket-based protocol that will be described in Section 3.5.1.

The performance chapter (Chapter 6) and the conclusions (Chapter 7) evaluate the success of these techniques at preserving bandwidth. To summarize, the Sawmill file system is very successful for large requests and moderately successful for small requests. About 80 to 90% of the raw disk bandwidth is available to the network for large requests, but for small requests the CPU load and seek times become bottlenecks.

3.4. Processing of requests in Sawmill

This section gives a high-level description of how client requests are handled in the Sawmill file system. This description steps through the operations performed by the system to handle requests, focusing on how data is moved through the RAID-II controller and how the file server interacts with the controller. The internal details of the Sawmill file system will be discussed later. Although the exact sequence of events described below is specific to the RAID-II architecture, other controller architectures with similar data paths will behave similarly.

File system responsibilities are split between the file server and the RAID-II controller, as explained in Section 2.3. The file server is responsible for control tasks, while the controller handles data movement. The file server processes incoming requests, determines where blocks are stored on disk, issues commands to the controller, and handles interrupts. The RAID-II controller moves data between controller memory and disk, moves data between controller memory and the network, and computes parity for writes.

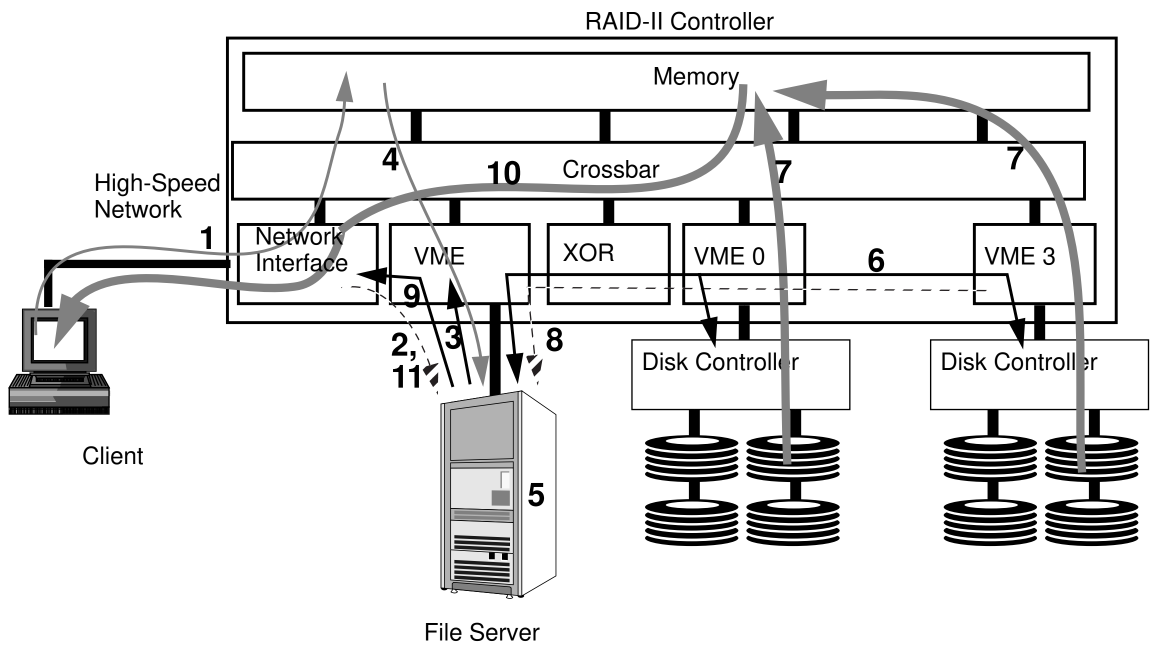


Figure 3-2. Processing of a client read request.

This figure illustrates the operations that take place to handle a read request from a client. The thick gray lines indicate data movement. The thin gray lines show request header movement. The dotted lines indicate interrupts. The thin solid lines indicate control messages. The operations are discussed in the text. Note that while the file server manages all the data movement, the actual data movement goes only through the bypass data path of the RAID-II controller.

Figure 3-2 illustrates how the Sawmill file system handles a typical read request. The read operation starts when the client sends a read request (containing the offset into the file and the length of the request) over the network and this data goes into RAID-II controller memory (1). The file server receives an interrupt informing it of the arrived data (2). The file server issues a command to copy the incoming request into file server memory over the VME link (3). After receiving the request (4), the Sawmill file system has the incoming request in kernel memory and can process it.

The file system next processes the request and sends the data to the client. After determining the positions of the requested blocks (5), the file server issues disk read commands (6). The disk controllers copy data into RAID-II memory (7) and interrupt the file server when done (8). If the file system requires any metadata from disk in order to handle the request, the metadata is read in a similar manner to (7) and (8) and is copied over the VME link as in (3). The file server issues a network send command (9) and the data is sent to the client (10). The file server receives an interrupt when the send is done (11). At this point, it can free the memory buffers if they are no longer required.

For large requests, the operation sequence is similar, but more complicated. When the first block of data has arrived from the disks, the file server issues commands to send the data across the network and to read the next blocks of data. Pipelining the transfers in this way reduces latency. In addition, disk requests may be initiated to perform prefetching.

Writes to Sawmill are handled in an analogous fashion to reads. To summarize, the client sends the write parameters over the network, followed by the data. Because Sawmill uses a log-structured file system, write data is collected into a log in controller memory. After a full segment of the log has been collected in controller memory, the file server commands the controller to write the data to disk.

Figure 3-3 illustrates in detail how a typical write request is handled. As for the read operation, the client first sends the write request over the network, and the request is received into RAID-II controller memory (1). The file server receives an interrupt (2) notifying it of the request. It issues a copy operation (3) and the request is copied to kernel memory (4).

Once the file server has the request, it can process the write by receiving the incoming data and sending it to disk. The file server first processes the incoming request and determines where the blocks should be positioned in the file log (5). The file server issues network receive operations informing the network interface where the incoming write data should be stored in memory (6). After the data is received into memory (7), the file server receives an interrupt (8). Depending on the size of the request, this receive operation may be repeated. If enough data has been received to complete a segment of the log, the file server issues a disk write operation (9). The log is written from memory to disk (10) after parity is computed in the XOR unit. Finally, the file server receives an interrupt (11), informing it that the disk write has been completed.

This high-level description of the file system operations illustrates several characteristics of how the file system must process client operations. Note that the file server only processes requests and handles control messages, while the controller does the actual data movement. The only data that must be processed by the server is the request header and any necessary file system metadata. By minimizing the data handling that the file server

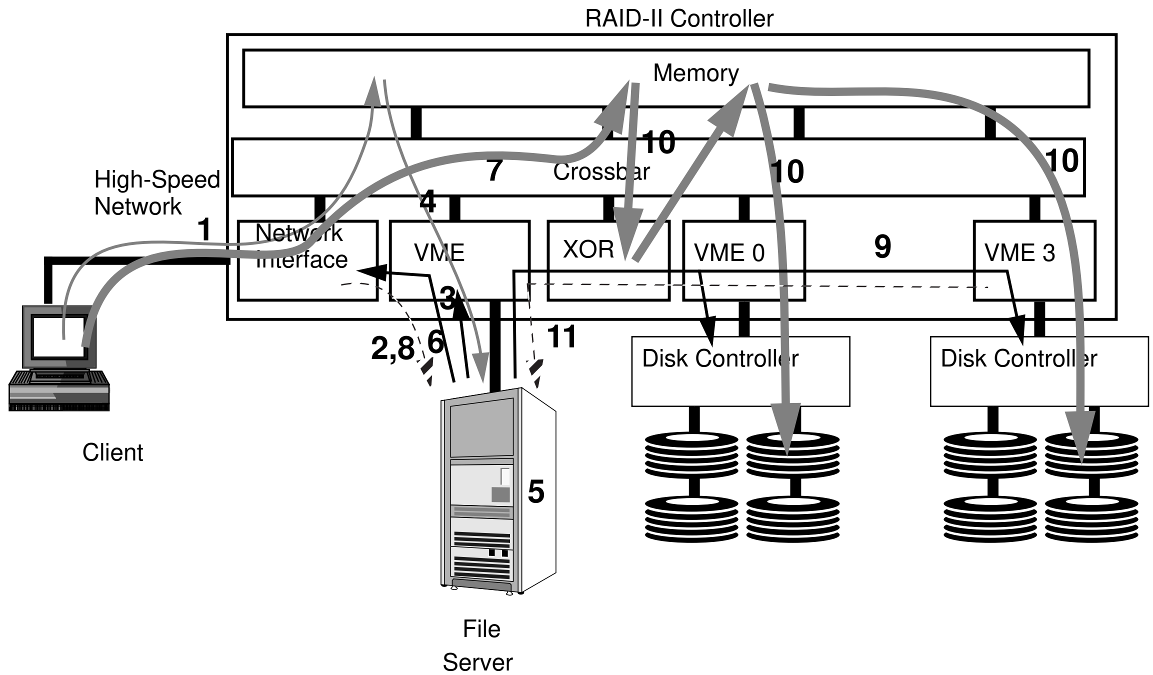


Figure 3-3. Processing of a client write request.

This figure illustrates the operations that take place to handle a write request from a client. The operations are described in the text.

---

must do, the controller architecture prevents the file server's data path from becoming a performance bottleneck. On the other hand, the file server must manage a large number of events and must ensure that the disks and network are kept in use.

The previous descriptions of reads and writes showed the events that take place to handle a single request. In actual use, the file system may handle multiple requests concurrently. In this case, it must schedule the disks and network to avoid contention and maximize performance. It must also allocate memory among the multiple operations.

3.5. Communication with the client

This section discusses how clients interact with the Sawmill file system. Clients communicate with Sawmill over a high-speed network and issue requests to the file system. The two main issues on the client side are how applications interact with the Sawmill file system and how the network is used to communicate between the clients and the server.

Since this thesis concentrates on the storage system, the goals of the client-side implementation were to provide simple, efficient access to the data stored on Sawmill, without implementing a complex client-side interface. There are several features, such as client caching of data or access through NFS, that were not implemented as they are not directly relevant for a prototype. Thus, the client side of Sawmill currently uses a very simple protocol designed for performance and ease of implementation.

Section 3.5.1, considers the network protocols used in Sawmill. This is followed by Section 3.5.2, which examines the application interface to Sawmill. It first discusses design alternatives for the interface and then briefly describes the application interface currently implemented with Sawmill.

3.5.1. Network communication with the client

The network protocols used to communicate between the file server and the clients can have a major impact on the system performance. The two important factors are minimizing latency to improve performance of operations that don't transfer large amounts of data, and maximizing bandwidth so large reads and writes can take place rapidly.

This thesis does not look at the low-level network protocols, but assumes that the network hardware and drivers provide efficient and reliable means of moving data between the clients and the server. This section, instead, examines the file-system level protocols running on top of the low-level network. Some of the high-level decisions, however, are dependent on the basic services provided by the network. In some cases, the file system can be simplified by taking advantage of features of the network.

Two approaches to network communication are connection-based protocols and connectionless protocols. In a connection-based protocol, a link is established between the client and the server. This link can then be used to provide a stream of communication, and the link is removed when it is no longer needed. A connectionless protocol, however, treats each message separately, without establishing a more permanent link between the client and server. These messages are often called *datagrams*. A connection-based protocol is usually faster than a connectionless one, once the connection has been established, since subsequent messages can make use of the established path. In addition, connection-based protocols can include flow control, which can improve performance and simplify the design. The downside of connections is the overhead of establishing and destroying the connection.

Two examples of file system protocols built on top of a connectionless and connection-based protocol are NFS and FTP [PR85] respectively. In NFS, each message between the client and server is sent independently and the server doesn't preserve state about the connection. With FTP (File Transfer Protocol), on the other hand, the FTP connection results in state both at the application level and the transport layer. At the application level, the FTP connection handles client authentication, directory selection, and transfer type at the start, so transfers don't have later overhead due to these factors. At the transport layer, FTP results in a low-level TCP [Pos81] connection; TCP improves performance by keeping track of network state such as the most efficient window size for network transfers, sequence numbers, and packet acknowledgment status. As a result, the connection-based nature of FTP and TCP helps them provide much higher data transfer bandwidth than NFS.

The high-level communication model for performing operations also affects performance. Both Sprite and NFS use a *remote procedure call* (RPC) model for file system operations. The client issues a request to the server and waits for a response. This is somewhat inefficient for operations where the response is not the goal of the operation, since the system will be delayed while an acknowledgment response is sent across the network. For instance, to write multiple file blocks, the client will send each block and wait for an acknowledgment from the server. This results in a lack of pipelining in the RPC model; each transfer must wait for the previous one to complete. In some cases excess acknowledgments can be optimized out [BN84], but a RPC model still limits performance.

One possibility for the Sawmill file system would be for the file server to export the Sawmill file system through an existing protocol, such as NFS [SGK+85]. A significant advantage of this is that clients could mount the file system without any modifications on the client side. One disadvantage of this is that performance would be significantly limited by NFS protocol overhead. As explained in Section 2.4.2, NFS does not perform well as a high bandwidth protocol. Clients are delayed for writes since NFS performs writes synchronously. Also, NFS limits the size of packets transmitted over the network, reducing the potential peak bandwidth. This approach wasn't used in Sawmill because of the performance penalty of a protocol such as NFS and the added difficulty to implement such a protocol in Sawmill.

To maximize performance while minimizing implementation effort for the protocol, the Sawmill file system uses a special-purpose, connection-based network protocol. In this protocol, a socket is opened between the file server and the client application, and they communicate over this socket. For example, an open operation on a Sawmill file results in opening a socket connection to the server. The open parameters are then sent across the socket. A close operation result in the closing of this socket connection. Operations such as reads and writes result in the parameters for the operation being sent across the socket connection, followed by the data, if any. By using a socket protocol, the implementation can take advantage of existing network drivers, which provide a socket interface and flow control. Originally, RAID-II was connected by the HIPPI to an Ultranet network and Sawmill used the Ultranet socket library [Ult91]. Now, RAID-II is connected to clients directly by the HIPPI and uses the HIPPI network protocols.

The current client protocol is not a fundamental property of the Sawmill file system, but was implemented for high performance and ease of testing. Several alternatives are possible, and would be desirable in a production implementation. For instance, rather than using one socket per open file, there could be a single socket per client, and all operations could be transmitted over this single socket. This would reduce the total number of sockets required, and would be especially beneficial if socket creation time is large. However, it would require more complex code on the clients, since the client code would have to multiplex different files onto the single socket. In particular, the client library approach would no longer work.

A final issue is the possibility of using separate networks for requests and for data transfer. This would typically be used if the high-speed network had high latency, for instance due to setting up a connection. It is common for mass-storage systems to use two separate networks, as discussed in [BCGI93]. Sawmill currently uses the high-speed network both for requests and for data. It would be straightforward, however, to use a separate network (such as the Ethernet) for requests. Because the high-speed network available for the Sawmill implementation has sufficiently low latency, there would be little performance advantage in this case from separating the requests across two networks.

3.5.2. Client application interface

The client application interface provides a way for applications running on clients to make use of the Sawmill file system. There are several alternatives for how the file system is presented to the applications, as well as different ways of implementing the interface.

From the user's perspective, the easiest way to access a Sawmill file system would be if the file system could be used identically to standard file systems. That is, normal Unix system calls would function unchanged. This approach would be most convenient to the user, since programs and utilities could run unchanged. As discussed earlier, one approach to providing transparent access to Sawmill files would be for the Sawmill file server to export the file system using a standard network protocol, such as NFS. This approach would have limited performance, though.

If the file server uses a nonstandard network protocol, then there must be code on the client to convert file system requests to this protocol. This code can provide different degrees of transparent access, from running any applications unchanged, to requiring rewriting of the applications. For maximum transparency, the file system code can be placed in the kernel, so existing system calls can be used unchanged. That is, applications use functions such as `read` and `write`, but these functions are modified to use the new file system. These new functions would then determine if the file being used is stored on a regular file system or a Sawmill file system, and perform the appropriate operations. One way this could be implemented is by putting the interface to Sawmill in the kernel and then the Sawmill file system could be exported to the rest of the system through file system hooks such as the vnode interface [Kle86]. This technique is more complex to implement than some alternatives, as it requires kernel modifications.

A less transparent approach allows the user to use standard file system operations, but inserts new code at user level to make use of the new file system. This can be implemented in several ways. One approach is to write a new interface library that provides new versions of the procedures used to manipulate files, such as `open` and `read`; the new code will provide Sawmill access. This allows applications to run without source code changes, but requires them to be relinked. If the operating system uses dynamically-linked libraries, then the new library can be substituted for the standard one without relinking the applications. This provides full transparency as long as applications are dynamically linked. There are some disadvantages, however, to providing the implementation as a user-level library. First, non-Sawmill operations may be slowed down slightly by the additional code, since every file operation must be checked to see if it is a Sawmill or a standard operation. Second, there are several features that cannot be implemented well at user level. One example is file system caching; a file system cache is much easier to implement in the kernel, as it can be integrated with an existing cache. In addition, there may be performance penalties to user-level code, if it requires additional copying of data between the kernel and user levels.

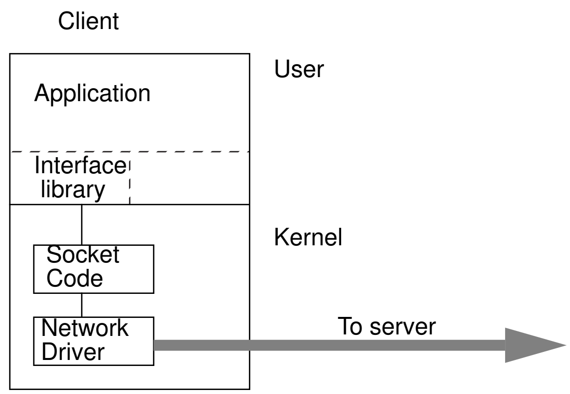


Figure 3-4. Client application interface implementation

This figure illustrates the current implementation of the client interface to the Sawmill file system. The application is linked with a small interface library that converts Sawmill file operations into socket operations. These operations use the kernel's drivers to communicate over the network with sockets.

The least transparent method would be to use special functions to access Sawmill files. These functions could be analogous to normal Unix system calls, but would be written to use the special file system rather than the standard Unix file system. This technique is used by some other systems that provide special file systems, such as the Connection Machine parallel file system [Thi93]. The main disadvantage of this approach is that applications and utilities must be modified to make use of the new file system operations. These operations can be implemented in the kernel or as a user-level library, as discussed earlier. One advantage of using new operations is that they can provide functionality beyond the standard file system operations. For example, the Connection Machine parallel file system includes new operations for reading and writing matrices efficiently.

This last approach is used in the current Sawmill application interface because of its simplicity and efficiency. New function calls, analogous to the Unix buffered file operations, are used to access Sawmill files. These function calls are summarized in Table 3-1. Client applications are linked with a small library to use the Sawmill file system; this structure is illustrated in Figure 3-4. This library converts file system operations into Unix socket operations. This approach minimizes client-side code development, since existing network drivers for the high-speed network can be used, and no new kernel code is required on the clients.

`RFILE *raidOpen(char *filename, char *type)`	Opens a file.
`int raidRead(char *ptr, int len, RFILE *stream)`	Reads from the file.
`int raidWrite(char *ptr, int len, RFILE *stream)`	Writes to the file.
`int raidClose(RFILE *stream)`	Closes the file.
`int raidSeek(RFILE *stream, int offset, int ptrname)`	Changes the position in the file.

Table 3-1. The client interface to Sawmill.

This table lists the function calls that clients use to access the Sawmill file system. These calls are analogous to the Unix buffered I/O calls `fopen`, `fread`, `fwrite`, `fclose`, and `fseek`.

3.6. Adding Sawmill to the operating system

This section discusses how the Sprite operating system was modified to operate as a file server using a file system such as Sawmill with new control and data paths. The modifications fall into three categories: providing the new operation request path from the clients, using the new data movement path, and supporting the new log-structured file system. Because the changes did not fall into a simple group, it was difficult to structure the changes to minimize file system modifications, and the implementation of Sawmill required in changes throughout the file system.

3.6.1. Data movement

The primary problem with data movement is how to modify the file system to take advantage of the bypass data path through the controller. Because file systems typically move all data through the file server's memory, the file system must be changed to move data instead over the new bypass data path. The key problems are how to integrate the bypass path with the existing path and how to make the bypass path work efficiently.

One alternative was to treat RAID-II as a new device and just write a new device driver. This provides a simple path to using RAID-II, since the file system can remain unmodified. The problem with this approach, however, is that all data will be read in and out of the cache in the server's memory. This negates the performance benefit of the RAID-II fast path, since transfer speeds will again be limited by the server's memory and bus. Because of its simplicity, this approach was used for initial testing of RAID-II. Performance, however was very poor, under 1 MB/s, as will be shown in Section 6.4. This illustrates that this technique is not suitable for a high-bandwidth implementation.

A second alternative was to implement the file system to operate on high-level data path abstractions. These abstractions would then be mapped onto the different physical data paths by lower-level code. That is, the file system would perform generic operations of moving blocks between the disks and memory and between memory and the network, without worrying about, for example, if the memory were controller memory or kernel memory. The different data paths for disks, network, and memory would be hidden at a lower level. Figure 3-5 shows how the abstract file system model could map onto an underlying physical system.

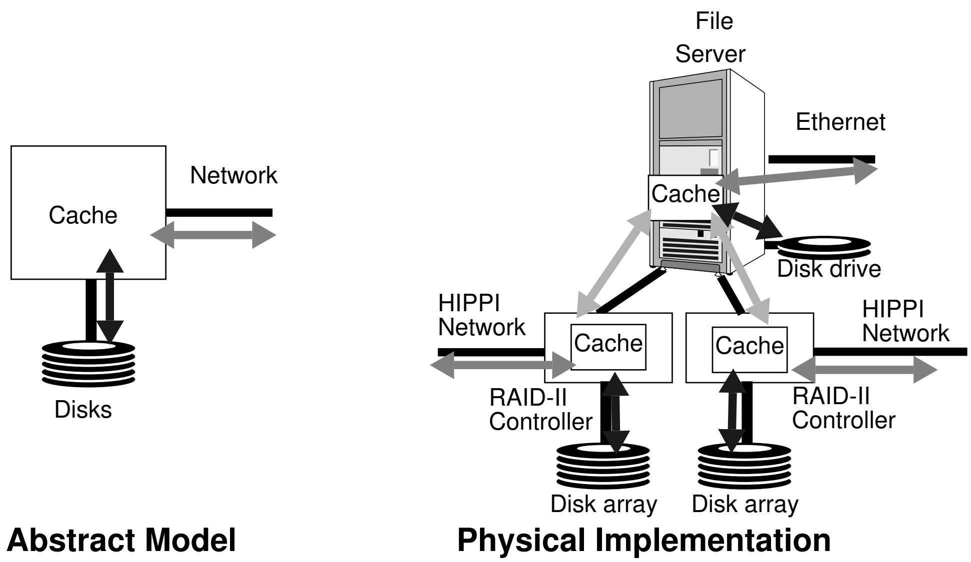


Figure 3-5. Abstract data path model of the storage system

This figure compares the abstract view of the file system data movement with a possible physical implementation. At the abstract level, the file system moves data between disks and the cache, and between the cache and the network. What this view hides is the physical details. The cache may reside in controller memory or file server memory. Each type of memory may service only one of the disks. The networks may be attached to a controller or to the server.

Table 3-2 summarizes four options for data movement under the abstract data path model. For example, consider a read request over a network attached to the server, for data on the controller's disks. At the abstract level, the file system reads blocks from the disks into memory and then sends blocks over the network. At the low level, the disk code determines that the disks are attached to the controller, so the data blocks are read into controller memory. The network send code determines that the data must be sent from kernel memory but is in controller memory, so it copies the data automatically. Thus, with the data abstraction model, the high-level file system can be implemented without worrying about the details of the controller datapath.

Device path	Network path	
	Server network	Controller network
Server disk	Read data into kernel memory. Send over server network.	Read data into kernel memory. Copy data to controller memory. Send over controller network.
Controller disk	Read data into controller memory. Copy data to kernel memory. Send over server network.	Read data into controller memory. Send over controller network.

Table 3-2. Data paths for reads under the data abstraction model

A storage system with a bypass data path has multiple sources and destinations for data. Data movement can be considered a choice of network and a choice of disk storage, where each choice can either use controller memory or the file server kernel memory. This yields four paths for data movement.

The primary advantage of the data path abstraction model is that it simplifies the file system design by separating data movement issues from file system design issues. The file system does not need to worry about which data path is being used and only the low-level data movement routines need to know where data blocks are stored. In addition, the model provides flexibility, since the system could be extended with multiple controllers or multiple networks without changes to the high-level code.

The primary disadvantage to this approach is that it assumes the high-bandwidth file system will use a block cache just like the old file system. This doesn't allow the data movement patterns to be optimized for the new controller architecture. As will be explained later, performing small block-oriented transfers is an inefficient way of using the RAID-II controller. It is much more efficient to implement new transfer models through the controller, so data can be moved as large streams. Thus, while allowing the file system to treat all data movement uniformly through these abstractions simplifies the file system, it decreases performance since the file system cannot take advantage of the characteristics of a particular data path.

A second disadvantage is that it pushes a lot of complexity down to the low level of the system, since the correct data path must be determined based on where data blocks are stored and blocks must be moved back and forth as appropriate. This requires the low level data movement routines to be closely integrated with the cache code. In addition, since all data movement must go through these general low-level routines, performance may suffer, since every transfer must be examined to determine the proper data path, rather than allowing this to be decided once at a high level.

A third alternative for the file system, besides moving everything through the server and hiding data movement at a low level, was to do everything through the controller. That is, Sawmill would be a standard file system, except that it would use controller memory instead of file server memory and run only using the controller data path. That is, the file system is simply pushed down into the controller, yielding an entirely separate file system, parallel to the existing one. This can be implemented by using a block cache that holds cache blocks in controller memory instead of server memory. In this approach, transfers can take place through the controller data path. This is a simpler approach than the data abstraction model, since there is only a single data path. This approach still has the disadvantage that the data movement won't be optimized for a high-bandwidth RAID.

A final alternative for handling the new data path is to implement a new file system that uses the bypass data path. The disadvantage of this, of course, is the effort in implementation. The key advantage is that the data path can be optimized for the new architecture, instead of using old data movement patterns on the new data path. Because of the large difference in bandwidth between standard disks and the storage system RAID, as well as the performance differences from a RAID, a standard file system won't perform as well as one optimized for the high-bandwidth storage system.

The actual implementation of Sawmill primarily uses the final approach of rewriting the file system, but the other approaches also have a role. The main Sawmill data path was rewritten to avoid a block cache and move data as pipelined streams. This avoids the excessive overhead of using a standard block cache, which will be discussed in Section 4.2. Sawmill also uses a new device driver that moves data over the old path through the server. This provides file system access to the RAID-II system without using the new Sawmill path; this is useful for testing and for bootstrapping the system. Finally, the data abstraction approach is used to provide access to Sawmill from the slow network; this is discussed in more detail in Section 3.8.

3.6.2. Log-structured file system

The second area of changes to the file system to support the new controller architecture was the log-structured file system backend module. This file system module writes blocks to disk and reads blocks from disk. Because this module controls the organization of data on disk, it is at this level that the log-structured file system is implemented. Sprite already contained Rosenblum's original LFS implementation [Ros92], which uses a single disk. Much of the LFS implementation was rewritten for Sawmill, however, to take advantage of the RAID-II architecture.

Changes to the LFS module fell into three categories. The first category was changes to the layout to make it run more efficiently on a RAID. This will be discussed in detail in Chapter 5. The second set of changes concerns how to use the controller memory and the new data path. This will be covered in Chapter 4. The final group of modifications to the LFS module changed its interface to interact better with the control flow of the Sawmill file system, which will be discussed in Section 3.7.2.

3.6.3. Client interface

The file server implementation also required new code to support the new client interface. As described in Section 3.5, using an existing remote procedure call file system interface would result in significant performance penalties. For the reasons described there, Sawmill uses a socket-based protocol to communicate with clients, and uses simple stream-based requests for file system operations. To provide this interface, Sawmill required a new set of interface routines.

3.6.4. Other changes

Several other parts of the file system also needed to be changed to support the new controller. Two important changes were to the block cache and the name server.

The file system block cache module required changes to handle data movement between controller memory and the file server cache. The modifications to the block cache are explained in more detail in Section 3.8.

The name service module provides the translation between symbolic file names and internal file descriptors. The name service in Sawmill is based on the standard Sprite implementation. Since name lookups make heavy use of metadata to perform the name translation, the name service module had to be modified to move metadata over the link between the controller and the file server. This metadata path is described in more detail in Section 4.5.

3.6.5. Summary

The implementation of the Sawmill file system required changes throughout the existing file system. Although restricting the changes to a single file system module would have simplified the design, this proved unworkable. The key reason for the widespread changes was the new data path, which required changes to all parts of the file system involved with data and memory storage. Some of the design decisions, such as the new layout techniques, the avoidance of the block cache, and the implementation of two data paths, also resulted in more changes to the file system.

3.7. Structure of the Sawmill file system.

Internally, the Sawmill file system consists of three functional modules. The key module is the *request manager*; it provides the high-level file system operations and manages the storage system resources. The *LFS module* implements the on-disk storage of data; it manages the details of where blocks are stored on disk and Finally, the *event scheduler module* ties these modules together and handles the control flow; the request manager consists of routines to handle disk and network events; the event scheduler executes these routines as required. This section discusses these modules and their interconnections.

Figure 3-6 shows the interactions among the Sawmill modules. The Sawmill request manager is at the center of file system operation processing. It issues commands to the network module to send and receive data and requests, to the log-structured file system to manage disk layout, and to the striping driver to perform reads and writes. The request manager receives interrupts from the network driver and disk driver when operations complete or new network data arrives. The event scheduler queues up actions that the request manager needs to perform, and issues them when the request manager is ready.

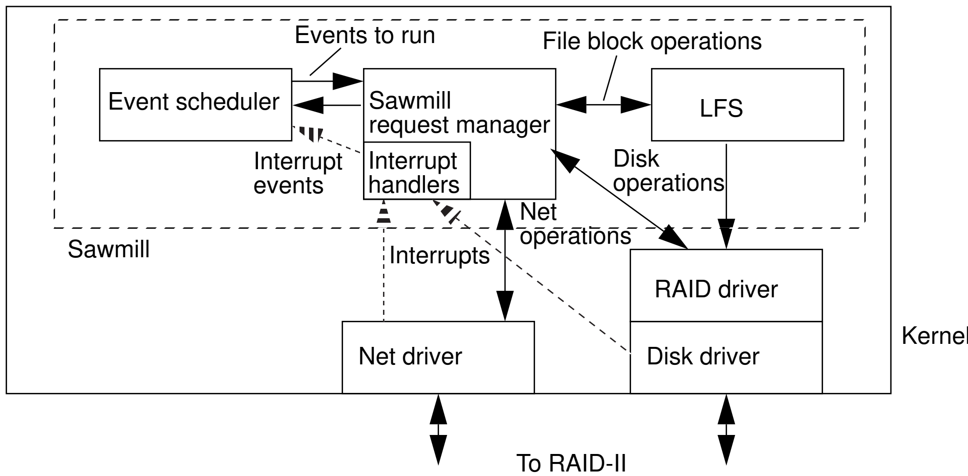


Figure 3-6. The software structure of Sawmill.

This figure illustrates the components of the Sawmill file system and how they interact with the rest of the operating system. Dotted lines indicate interrupts. Solid lines indicate data movement.

There are two reasons that Sawmill is structured in this way. The first reason was the decision to use an event-based system, as will be discussed in Section 3.7.2. For convenience, the event management code was separated from the rest of the code to form the event scheduling module. The second reason for this modular organization is that the log-structured file system in Sawmill is based on Sprite-LFS. Since the LFS module in Sprite is separate from the rest of the Sprite file system, it was natural to keep the LFS as a separate module. This led to the structure of one module to handle the high-level file system operations and a second module to handle the disk storage details of LFS.

As explained in the previous section, the implementation of Sawmill required changes throughout the file system. Figure 3-7 illustrates how the Sawmill modules fit into the Sprite file system, how the new modules interact with the rest of the file system, and where the necessary changes occurred throughout the file system. Sawmill uses the Sprite name server to provide name operations such as filename lookups. Many parts of Sawmill-LFS use the original Sprite-LFS. Finally, Sawmill interacts with the cache code through a “split cache” to provide a second data path; this will be discussed in Section 3.8.2.

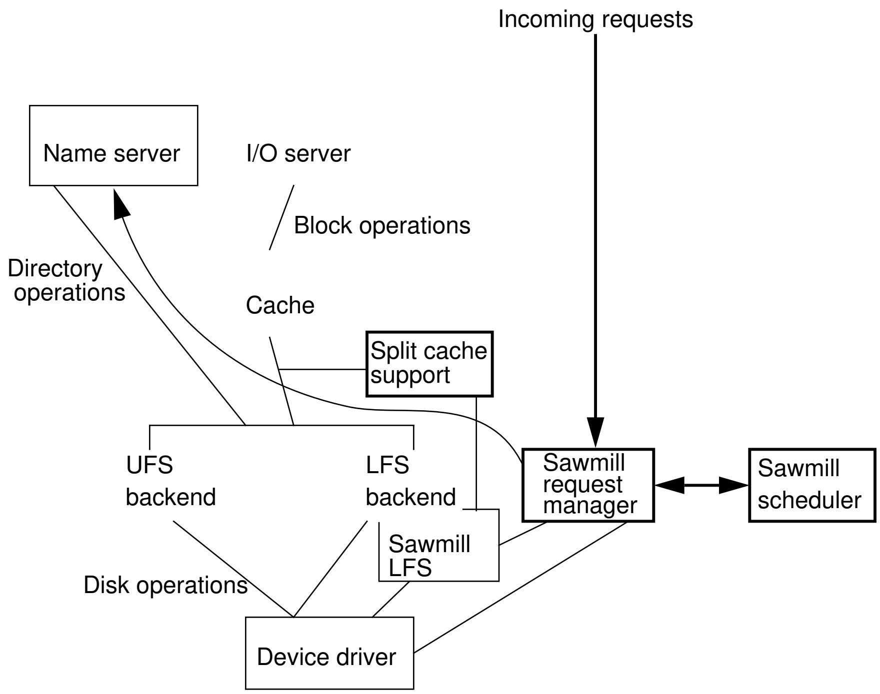


Figure 3-7. Sawmill in the Sprite file system

This figure shows how the Sprite operating system is modified to support Sawmill. The Sawmill request manager handles the incoming requests. It uses the Sawmill LFS module to determine where blocks are positioned on the disk. Scheduling of events is done through the Sawmill event scheduler. Sawmill uses the existing Sprite name server to perform name lookups; this module was modified to support the new metadata path of Sawmill. This figure can be contrasted with Figure 2-11, the original Sprite file system.

The remainder of this section describes the three modules of the Sawmill file system. Section 3.7.1 discusses the request manager, Section 3.7.2 covers the event scheduler, and Section 3.7.3 summarizes the LFS module.

3.7.1. The Sawmill request manager

The request manager is the heart of the Sawmill file system. It provides the interface with the clients. It also provides the connections between the log-structured file system and the rest of the system, issues and handles disk operations, and looks after controller memory management.

The first task of the request manager is to provide the file system interface to the clients. That is, the request manager performs all network communication with the clients. It establishes connections with the clients, opens files for them, receives requests, and sends and receives data from the clients. In some file systems, such as Sprite, the client interface is provided as a separate layer above the file system implementation to provide more flexibility in the interface design. Since Sawmill is designed to operate with high-speed data streams, the Sawmill client interface is tied much more closely to the rest of the file system. For instance, the original Sprite file system handles a read request by fetching the data into the cache and then passing these blocks up to the client interface routines, which send the blocks to the clients. In Sawmill, on the other hand, the disk read and network send operations are closely integrated in the request manager. This allows Sawmill to reduce latency by sending data to the client as soon as part of it arrives from disk, rather than waiting for all the data to arrive.

The second role of the request manager is to communicate with the LFS module to carry out file system operations. The request manager deals with abstract file operations; it uses the LFS module to determine the actual position of blocks on disk.

The third task of the request manager is to perform the necessary disk reads and writes. Most disk operations are not performed directly by the LFS module; instead, the LFS module only informs the request manager of the appropriate disk addresses. This allows the request manager to schedule disk operations and perform prefetching.

Finally, the request manager manages controller memory. The main use of memory is to perform prefetching for reads. Chapter 4 discusses controller memory in more detail.

Thus, the request manager is the core of the Sawmill file system. It ties together the clients, the LFS disk layout module, and the RAID-II device drivers. The next section discusses the event-based model and how this affected the implementation of the request manager. The following section discusses the LFS module and its interactions with request manager.

3.7.2. The event scheduling mechanism

All flow of control in the Sawmill file system goes through the *event scheduler* module. Sawmill control flow is organized around *events* such as the completion of a disk operation or the arrival of a network request. Events have associated *event handlers*, the code that is run when the event occurs. For instance, when a network request event occurs, the associated event handler will start processing the request. The event scheduler module queues up event handlers waiting to be executed. It then sequentially calls these event handlers, using a single thread of control.

Under the event-based model, the request manager is implemented as a collection of event handlers. The request manager waits for events to come in. Events include the establishment of network connections, receives of data, completion of sends, and completion of disk reads and writes. For example, when a connection establishment occurs, the server must create a data structure to handle this connection; this is done by the associated event handler. The request manager then waits for more requests. When an open request arrives, for instance, the request manager handler opens the file on the RAID and prepares to handle requests.

Figure 3-8 illustrates the flow of control through the event scheduler and the event handlers. To fit the event model, the request manager module is implemented as event handling routines and interrupt-level “callback” routines. Instead of procedures with a call-return sequence, each action started by an event handler has an associated interrupt-level callback that is called when the action is completed and an interrupt occurs. This callback then calls the event scheduler to add the appropriate new event handlers to the wait queue. At non-interrupt level, events are pulled sequentially off the queue by the event scheduler and are handled by calling the appropriate routine.

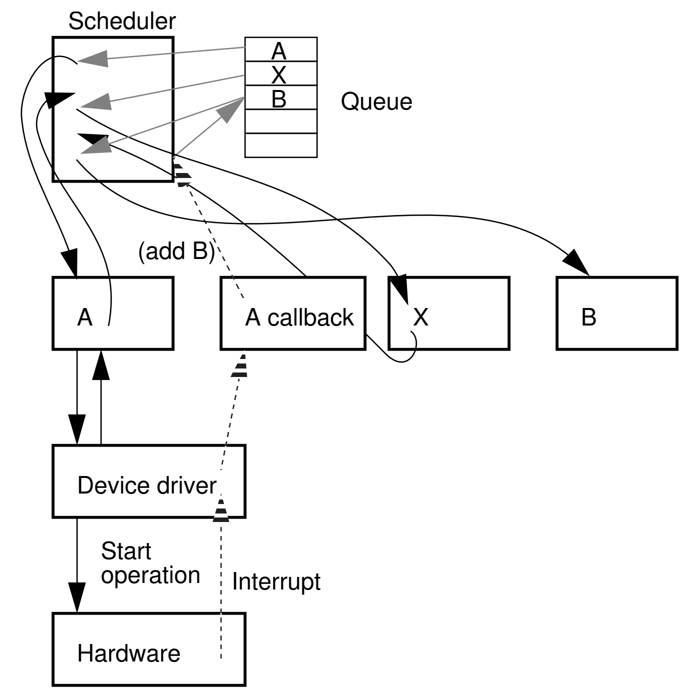


Figure 3-8. The event handler structure

This figure is an example of the flow of control through a chain of event handlers. In this example, first procedure A is invoked to handle an event; it initiates a disk operation, specifying that “A callback” should be invoked when the operation completes. While the disk operation is in progress, procedure X is invoked to handle some other event in the queue. When the disk operation is completed, “A callback” is invoked at interrupt level; it adds B to the event queue. When the interrupt routine completes, the event scheduler removes B from the event queue and calls procedure B to handle it. Solid lines indicate procedure calls during normal execution, while dotted lines indicate interrupt-level calls. Gray lines indicate events added to or removed from the event queue.

The event scheduler is in contrast to multi-threaded systems. In these systems that use multiple threads or processes, any thread can block but context switches among threads are required. Figure 3-9 illustrates how context switches occur with a multi-threaded approach. Multiple threads could be used in several ways. For instance, in the Sprite file

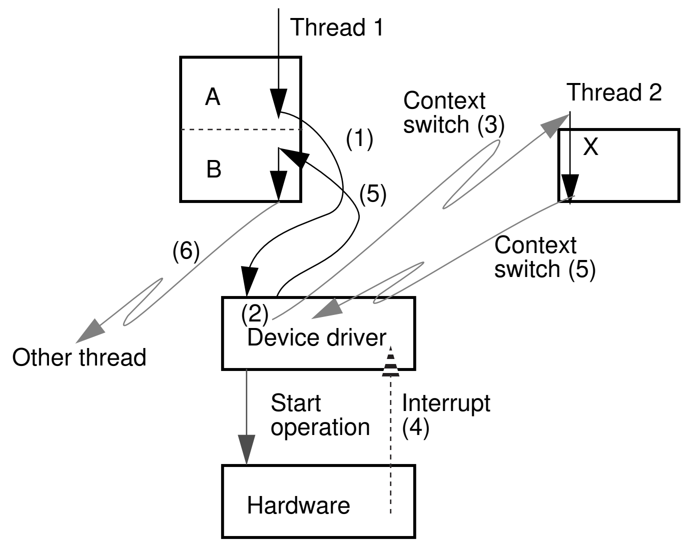


Figure 3-9. The multithreaded approach

This figure illustrates a multithreaded implementation of the example illustrated in Figure 3-8. Routine A calls the device driver (1), which starts a disk operation and blocks until completion (2). The processor context switches (3) to an unrelated thread X. When the disk operation completes, an interrupt results (4) and the device driver is unblocked. A context switch (5) back to the first thread occurs, and execution of the original procedure continues. When it finishes, the context switches to another waiting thread (6). Note the many context switches required compared to Figure 3-8.

system, each incoming request has a separate thread. The file system has additional threads: the cache writeback code runs as a separate process, as does the log cleaner and any prefetches. Thus, a multi-threaded system may have many threads, with the associated cost of context switching among them and locking shared data structures.

The event-driven structure has several advantages. First, it minimizes context switching among multiple processes. Because events are handled by a single thread and callbacks are handled at interrupt level, there is no need to switch among multiple processes. Any action that could block in the event model, however, must be broken apart with callbacks so other actions can run while the first is blocked.

The second advantage of the event structure is that it simplifies overlapping asynchronous actions such as prefetching or overlapping network and disk operations. The network and disk actions can both be started and will generate new events as they complete. This requires a synchronization point to decide what happens next. As each disk or network operation completes, the flow of control returns to the synchronization point to determine if anything new should happen. For instance, if a disk read completes, this data can then be used to satisfy an incoming request. If a network send completes, a buffer can be freed and perhaps more data can be sent.

Finally, the event model reduces locking costs. In a multi-threaded system with a preemptive scheduler, shared data structures must be locked to ensure that two processes do not try to change the data structure at the same time. With the event model, a thread will continue execution until it ends, so two threads will not simultaneously access a data structure. Thus, most locking of data structures can be avoided. If Sawmill ran on a multiprocessor, however, locking would be required to allow concurrent execution of threads.

There are several disadvantages to the event-driven model. First, because it requires routines to be broken apart wherever they can block, it makes the program structure harder to follow. Instead of putting a synchronization call into the code, the code is broken into separate procedures to handle the different events. Related to this, state cannot be preserved in a stack frame, but instead must be explicitly packaged up and passed around from routine to routine. Finally, the event-driven approach makes debugging more complex, since there is no stack trace to examine to determine the calling sequence.

To evaluate the effects of the thread-based model on the number of context switches a comparison can be made with the existing Sprite file server. Measurements of the existing Sprite file system show that a data rate of 750 KB/s requires about 290 context switches per second. Each context switch on the Sun 4 takes about 240 µs. Thus, at the current context switch rate, the Sprite file system would spend all its time context switching to support only 11 MB/s (750KB/s ÷ 290 switches/s ÷ 240 µs/switch). While the context switch cost could be reduced by, for example, increasing the block size so fewer context switches are required, the cost would be likely to remain high. Thus, reducing the number of context switches through the thread-based model is important, since context switches would otherwise result in a performance bottleneck.

To summarize, the event-based model improves performance, due to the reduction in context switches. The downside is that this structure makes design and debugging of the software significantly more complex, compared to a standard procedure-based structure.

3.7.3. The Sawmill LFS module

The Sawmill LFS module implements the log-structured file system used in Sawmill. The Sawmill request manager uses the LFS module to determine where blocks are stored on disk for reads, and to determine where blocks should go for writes.

This section provides a brief overview of the log-structured file system used by Sawmill and discusses the interface between the log-structured file system and the rest of Sawmill. Chapter 4 provides more information on how the LFS module supports reads and Chapter 5 discusses how it supports writes.

Log-structured file systems were introduced in Section 2.6. To review, LFS is a disk layout technique that writes all data to a sequential append-only log. Unlike traditional file systems, where each block has a fixed location on disk, the location of a file block in LFS changes every time it is written. Because LFS uses a sequential log, garbage collection is necessary to remove stale data from the log.

For reads, one interface between the LFS module and the rest of the file system is as a cache backend; this method is used in Sprite. With this approach, all file reads go through the block cache. If a read cannot be handled by the cache, the cache module gives the LFS module an empty cache block and requests the required block. The LFS module performs the disk operations and returns the block filled with the required data.

The cache backend approach for reads has several disadvantages. First, cache operations have a significant processing overhead, resulting in per-block costs that can limit performance through a CPU bottleneck. Second, performing individual block operations limits performance since large disk operations provide much higher bandwidth, especially on a RAID. These issues will be explored in more detail in Section 4.2.

Instead of an interface that operates on individual cache blocks, Sawmill uses an interface that allows the file system to make an arbitrarily large data request to LFS in one operation without moving data through a cache. This raises the design question of how to handle large requests. On the one hand, the file system should issue as large a request as possible to the LFS module, since this increases the opportunity for parallelism. On the other hand, a large request increases the latency to handle the entire request, and reduces the ability to pipeline the request by sending the first part over the network while the second part is being fetched. This is especially a factor if part of the request has been prefetched, so it can be immediately provided from memory, while the remainder must be slowly fetched from disk.

For these reasons, Sawmill uses a hybrid approach to handling large requests. The file system can request a large sequence of blocks, but the LFS module will only return as many as can be efficiently provided. The file system can then start sending the first blocks while re-requesting the remaining ones. Thus, the returned sequence may only be part of the request; it is the number of contiguous bytes that can be supplied. This sequence of bytes may be from the kernel cache, the segment cache in controller memory, or the disk. When the data is ready, a callback calls the request manager, which sends the bytes over the network. While this data is being sent, the request manager can request more data. In this way, network transfers can be overlapped with disk transfers.

To write data, there are several possible interfaces. The first is the cache writeback layout approach used in BSD-LFS and Sprite-LFS. After each block of data comes in, the file system calls the cache code to enter the data into the block cache. As a separate process, the cache backend lays out data from the cache into the segment write buffer and the segment is written to disk. As will be explained in Chapter 5, this approach has the disadvantage of high CPU load, since each block must be processed individually through the cache. In addition, with the RAID-II controller, the block cache required an extra copy operation to arrange blocks in the log.

Instead of a cache, Sawmill uses an interface that allows “on-the-fly” layout. With this approach, incoming data is stored directly into the write segment buffer, rather than going through a cache. When all writers have finished writing to a segment, the segment is written to disk. The motivation and details of this write approach will be discussed in more detail in Chapter 5.

To conclude, the interface with LFS is substantially different in Sawmill because it doesn't operate on cache blocks, but handles large requests. Other important differences between the standard Sprite LFS and the Sawmill LFS are the controller data path of Sawmill, the layout techniques, the new metadata movement, and the event-based control structure. Because of these factors, which will be discussed in more detail in the next two chapters of this thesis, the Sprite LFS had to be extensively modified to create the Sawmill log-structured file system module.

3.8. The two data paths

Besides the data path through the RAID-II controller, the Sawmill file system also provides access to data through the file server. Because this path does not provide the fast, direct data access RAID-II provides, the path is called the "slow path." While some details of the slow path are specific to RAID-II, a similar path is likely to be useful in any storage system that has a high-speed network separate from the regular network. This section discusses the motivation for providing a slow path and the implementation of the slow path. The main idea of the implementation is the memory management of data along two data paths, leading to the "split cache," which will be discussed below.

The slow path consists of accesses through the file server to data stored on the controller's disks, as shown in Figure 3-10. Slow path operations either arrive over the Ethernet connected to the file server, or are generated by programs running on the file server. This is in contrast to fast "bypass" path requests, which access data through the high-speed network connected to the controller. This bypass path keeps the file server's memory out of the data path, increasing the potential bandwidth. To access data over the slow path, on the other hand, the data must be copied between the controller and the kernel memory on the server.

The first reason for implementing the slow path is to provide access from the file server itself to files on the disk. Applications on the file server may wish to access this data, for instance. Since our architecture does not have a high-speed network connection directly to the file server, but only to the RAID-II board, the server cannot access data through the same network path used by clients. There must be a separate path for the file server to access data, and this is provided by the "slow path."

A related advantage of the slow path is that by allowing the file server to access data off RAID-II directly, the standard file system operations and utilities can run on the file server and use files on the RAID-II disks. This makes experimentation with the file system much easier.

Finally, the slow path allows the file server to act as a gateway for clients that aren't on the high-speed network. In our system, this is an Ethernet that is attached to the file server. To handle this path, data must be moved to and from the server's memory.

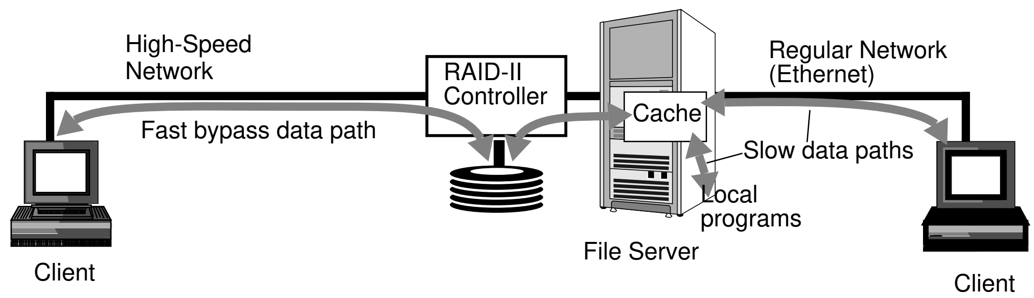


Figure 3-10. The slow and fast data paths

For clients on the high-speed network connected to the controller, Sawmill has a fast bypass data path. Sawmill has a separate slow data path to handle requests over the slow network attached to the file server or requests from applications running on the file server. For the slow data path, file blocks are copied from the controller to the server cache and then the cache can service the requests as in a standard file system.

There are several alternatives for providing access to RAID-II data to the file server. One mechanism would be to provide a connection to the bypass data path and then the file server could access data in the same manner as any other client. A second alternative would be for a separate machine to operate as a gateway and provide data to the file server. The alternative used in Sawmill is to integrate these file accesses into Sawmill.

There are several design alternatives for providing data to clients that aren't on the fast network. One alternative would be to ignore these clients. A second alternative would be to use a different machine as a gateway between the two networks. A third alternative would be to use the file server as a server for both networks. Providing a gateway on a separate machine would simplify the design of the file system. Instead, service over the slow network is integrated into the Sawmill file system.

3.8.1. Implementation of the slow path

The previous section gave the motivation for providing access to Sawmill files over the "slow path", that is, from the file server or clients attached to the file server. There are, however, several different designs possible for how the operating system could implement operations over the slow path on a Sawmill file. The key design decision is how far the request will propagate through the standard file system path before being redirected to the Sawmill file system. This section discusses the alternatives and explains the decision to link the two systems together just below the block cache. Although this implementation of two data paths was done with RAID-II in mind, it could be applied to any storage system that has a controller attached directly to the network, giving a data path separate from the file server.

Sawmill files can be opened by the regular file system without any extra effort because the high-level file system data structures are shared between Sawmill and the regular file system. Because metadata must be handled by the file server in order to open a Sawmill file, there is no special processing that must be done to open the Sawmill file. The *file handle*, the kernel data structure associated with an open file, has a field to indicate if the file is stored on a Sawmill file system or on a regular file system. Thus, each subsequent operation on the file can be directed to the appropriate file system.

One alternative for handling two types of file systems would be to switch between file systems at a very high level. That is, almost immediately, the operating system would determine if the operation used a Sawmill or regular file. If the operation used a Sawmill file, the control would pass to new interface code. This code would copy the data to the controller memory and then perform the operation using something similar to the standard Sawmill path.

A second alternative would be to perform the decision between Sawmill and regular file systems at a very low level, such as the device driver. In this approach, Sawmill files would be treated like regular files by the rest of the system until the data is about to be transferred to or from the disk. At this point, device driver code would determine if the file were actually on a Sawmill disk. If so, the data would be copied between kernel memory and controller memory, and the operation would take place on the controller. The main disadvantage with this approach is that it requires the Sawmill file system to be basically identical to the old file system. The data structures must be shared between the regular file system and Sawmill and data layout on disk must be identical. Since Sawmill was intended to use new disk layout techniques, this technique is not possible with Sawmill.

A final alternative would be to switch requests to Sawmill at an intermediate level, after the file system has performed some processing on the request, but before the details of device layout have been applied. In the Sprite operating system, a natural point to do this is below the block cache, between the cache and the disk storage module. At this point, file system requests have been broken down into file system blocks, but are ignorant of the details of how the data is stored on disk. At this level, requests to Sawmill can be handled by copying blocks to or from the controller memory and performing Sawmill operations on the blocks.

There are several trade-offs between this approach and the approach introduced earlier of switching operations at a very high level. The main advantage of performing these operations at the cache block level is that it provides a convenient abstraction. Instead of having to handle several different file operations (e.g. read data to network, write data from user memory), the system need only handle reading and writing of cache blocks from kernel memory. The main disadvantage of the cache approach is that it results in data being stored in the file system kernel cache. This results in the need to maintain consistency between data in the file system cache and data stored in the controller or on disk.

Sawmill was implemented with the cache block method for several reasons. First is the convenience described above. Another reason is that Sawmill was originally designed to use a block cache in controller memory, as in the “data abstraction” design discussed in

3.6.1. In this design, the controller cache and the kernel cache were viewed as two components of a single cache, that is a *split cache*, split between the two types of memory. In this approach, it is natural to process slow path requests to Sawmill through the block cache. Although the data abstraction design was replaced by a stream-based design, the "split cache" implementation of the slow path remained, as well as the term "split cache," even though controller memory is no longer used as a block cache.

To summarize, Sawmill provides access to file data over two paths: the fast bypass path through the controller that bypasses the file server's memory, and the slow file server path that moves data through the file server's memory. The slow path requests are merged into the Sawmill file system below the block cache. Thus, the file server kernel cache is split between Sawmill data and regular file system data. Since Sawmill data is split between controller memory and file server memory, the mechanism of caching Sawmill data in server memory is called the "split cache."

In retrospect, the best solution would probably be to avoid providing access over the slow data path to Sawmill files, or to provide this access through a gateway outside the file system. If this access is required, avoiding the block cache and providing the transfer to Sawmill at the high level would probably be easier than doing the transfer at the block level. This would avoid complicating Sawmill with consistency issues that could otherwise be avoided.

3.8.2. The split cache

The slow path is implemented by providing the standard file system block operations for files stored using Sawmill LFS. With this interface, the Sawmill file system looks like a normal Sprite file system to the rest of the operating system. Thus, the file server and clients accessing the file server through Sprite RPCs can access Sawmill files. The code for Sawmill LFS copies blocks between server memory and controller memory to service these requests.

In this implementation, server access is accomplished by tagging blocks with an indication of whether the blocks are in the server or RAID-II memory. For reads, the request goes through the server block cache. When the block cache attempts to load the block, it determines that the file is stored on RAID-II and the request is passed to the Sawmill code. Because file server blocks may, in general, be a different size from Sawmill file blocks, the blocks may have to be broken up when loaded into the cache. Once the block is loaded, it can be sent over the Ethernet or provided to the server application as appropriate.

There are two buffers for memory: the memory in the controller, and the memory in the server. Data can be stored in either place or both places. Data may be in the server memory for several reasons: the server itself may use the data, or the server may be providing the data to other clients over a different network. As a result, data and metadata may reside in the file server's memory as well as in the controller memory. Since this data is cached in two places, we call this the split cache.

The split cache complicates the design in several ways. First, server memory and controller memory must be kept consistent; this will be discussed in Section 4.6.4. Second, data must be moved efficiently between the two caches. Finally, the caches must deal with the fact that they use different block sizes.

The need to move data between the controller and the file server limits the performance available over the slow path. This is primarily a limitation of the RAID-II hardware, since the VME link between the controller and the file server is fairly slow. As will be discussed in Section 6.1.1, the link takes about 1.7 ms to move a 4 KB block, so only 2.4 MB/s of blocks can be moved.

The third problem occurs because different parts of the file system use different block sizes. The slow path uses standard file system blocks of 4K, which are much smaller than the RAID file system blocks. Thus, when data is moved between the RAID and the slow path, translation must occur. To read a small block is fairly straightforward; the large RAID block is read, and only the required part is used.

Handling multiple block sizes is significantly more complex for writes. When a small block is written to RAID, the large block must first be read into memory. This large block may entirely be in memory, or parts of it may be in kernel memory, or it may reside on disk, or the block may be at the end of the file and only partially filled. For whatever combination, the large block must collect the fragments to create the proper large block. Thus, a write from the old file system to Sawmill may result in multiple blocks being read from Sawmill. The file system must ensure that this happens efficiently for sequential writes, or else unnecessary reads will take place.

Providing the split cache to support the separate slow path complicates the implementation of the file system considerably. Some of these complications would be required to implement the metadata cache, since metadata must be stored in file server memory and copied between the server and the controller. The slow path, however, introduces many additional problems. For instance, because of the slow path, operations from outside Sawmill can change Sawmill data; implementing this required close links between the Sawmill and non-Sawmill code. In addition, the slow path makes consistency much harder, since non-Sawmill operations can cause Sawmill data to become inconsistent. Because of these problems, the benefits of the slow path are probably not worth the implementation effort.

3.9. Conclusions

The Sawmill file system is a high-bandwidth log-structured network file system that runs on top of the RAID-II disk array. Sawmill uses a log-structured file system to improve write performance on the disk array. Sawmill is also designed to take advantage of the bypass data path of the RAID-II controller.

Two main hardware features motivated the Sawmill design. First, a disk array provides high bandwidths for large reads and writes, but performance of small writes is limited. This motivated the use of a log-structured file system to avoid small writes. Second, a controller architecture such as RAID-II provides a bypass data path between the disks and the clients, bypassing the file server's memory and CPU. This motivated the use of buffering in controller memory, avoidance of the block cache, and the use of metadata caching.

The main ideas in the Sawmill implementation are the techniques used to take advantage of the new bypass data path. Instead of using a block cache like previous log-structured file systems, Sawmill uses a stream-based approach that moves large quantities of data in a single logical operation.

The Sawmill file system uses an event-based control flow model, with a single thread of control. The event-based model has several advantages, such as minimizing context switches and handling asynchronous events well. However, it made the implementation more difficult.

Sawmill uses two separate data paths and a "split cache" to provide access to data both through the fast network and through the file server. While this approach provides increased flexibility over a system with just the bypass data path, it significantly increases the complexity of the file system. On the whole, the benefits of the two data paths are probably not worth the increased complexity.

4 Using controller memory

"I've a grand memory for forgetting." – Robert Louis Stevenson

Storage systems often have memory in the controller that can be used for buffering data. In particular, the RAID-II controller contains high-speed buffer memory; all data transfers must go through this memory. This chapter explores techniques that high-bandwidth file systems can use to take advantage of controller memory.

This chapter fist explains how controller memory can be used for streaming and prefetching. In Sawmill, instead of the cache-based approach used by most file systems, requests are treated as large streams and memory is used for buffering these streams. This reduces the CPU processing overhead and allows efficient large transfers. Sawmill also uses prefetching to reduce latency in handing requests and to perform efficient large accesses to the disk array.

The second idea discussed in this chapter is how a file system can use controller memory that is separate from the file server's memory. Although these techniques were implemented with the RAID-II controller in mind, they are applicable to any I/O system that uses controllers that have buffer memory. Controller memory can be used as a readahead buffer, as a writebehind buffer, and as a segment layout buffer. Because controllers may have only a limited amount of memory, the file system must allocate it efficiently among these potential uses.

Finally, this chapter explains how to handle metadata and how to maintain consistency of data and metadata. These two issues arise in regular file systems, but additional complications arise when controller memory is used, because information may be stored in both controller memory and kernel memory.

This chapter concentrates on the use of controller memory for reads; Chapter 5 will cover the use of controller memory for layout of write data in the log. Section 4.1 gives an overview of the controller memory and lists its various uses. Section 4.2 discusses the possibility of using controller memory for caching. Section 4.3 describes how Sawmill uses memory in ways other than caching, such as prefetching and grouping requests. Memory allocation is discussed in Section 4.4. Section 4.5 covers the movement of metadata. Section 4.6 explains the consistency mechanisms used in Sawmill.

4.1. Controller memory

Two architectural approaches can be used to provide a “bypassing” data path in a storage system controller, where data is moved between disks and the clients without passing through the file server. In the first approach, the disks are connected directly to the network, without any memory in between. In the second, high-speed memory is interposed between the disks and the network. Figure 4-1 illustrates these two approaches. This thesis examines the second type of system, of which RAID-II is an example. In this type of architecture, the controller memory can be used for buffering, caching, and prefetching. This improves performance over a storage system without controller memory. On the other hand, controller bandwidth must be doubled to support moving data in and out of memory. As well, latency may increase due to copying data through memory.

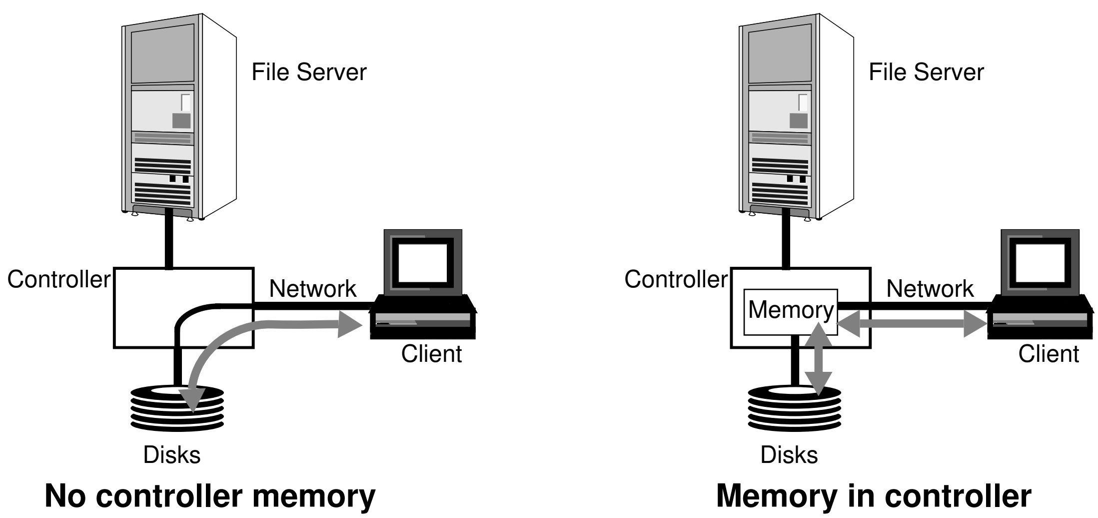


Figure 4-1. Data movement through the controller

This figure compares two alternatives for the controller architecture. In the first, data is moved directly between disks and the network. In the second, the controller has memory that is used for buffering.

There are two key issues with controller memory in such an architecture. The first is how controller memory can be used by the file system. There are many possibilities for memory usage, such as a block cache, prefetch buffers, and write buffers. The second is how to allocate memory among these multiple uses to maximize performance.

Although controller memory can be used in ways similar to file server memory, such as for a block cache, there are several important differences between controller memory and server memory. First, data stored in controller memory is normally not directly accessible by the kernel. Thus, as mentioned before, metadata that the file server must access needs to be handled specially. Second, controller memory will typically be smaller than file server kernel memory and will service higher bandwidths. This yields different performance tradeoffs for the memory, as will be described later.

Examining the hardware used by Sawmill, the RAID-II controller board has a 32 MB memory buffer, as introduced in Section 2.3. This memory is organized as four banks of interleaved memory to increase the available bandwidth. Each bank can support sustained simultaneous transfers of 40 MB/s, for a total sustainable memory bandwidth of 160 MB/s. The motivation behind this memory is to avoid the performance bottleneck that would otherwise be caused by the file server's slower, general-purpose memory. Note that the controller memory is relatively small compared to the memory that a traditional file server could have. For example, a typical Sprite file server has about 100 MB available for a file system cache, compared to the 32 MB of memory in RAID-II. This disparity is even greater compared to the file system bandwidths of the systems: Sprite supports a 1 MB/s Ethernet, while RAID-II provides over 20 MB/s of data. Thus, while Sprite could buffer about 100 seconds of data, RAID-II can only buffer about 1.5 seconds.

All data operations on RAID-II go through the controller memory. Disk and network operations use memory as either the source or the destination. Exclusive-or parity computations go from memory to memory. Finally, transfers between the controller and the file server go in or out of controller memory.

Because controller memory is used for every type of RAID-II operation, only a portion of the controller memory is available to the file system, and even less is available for buffering reads. The controller memory must be shared between the RAID striping driver and the Sawmill file system. The RAID striping driver uses memory to buffer operations from kernel memory, to compute parity, and to reconstruct data after a disk failure. The Sawmill file system uses the remaining memory for several purposes: buffering network requests and replies, holding LFS segments before they are written to disk, buffering data during reads, and holding LFS segments for cleaning. The result is that only a fraction of the total controller memory is available for a cache or other file system uses.

To summarize, memory in a storage system controller can improve performance by allowing the file system to perform caching and buffering. This memory, however, may be of a limited size, especially in comparison to the bandwidth of the controller. Thus, the file system must make efficient use of this memory. The RAID-II storage system provides an example of such a controller, with 32 MB of controller memory.

4.2. Caching in controller memory

In the original Sawmill file system design, the plan was to implement a standard file system block cache in controller memory. This section discusses the implementation of a block cache in controller memory and the reasons why Sawmill did not take this approach.

Many file systems use block caches to improve performance. As discussed in Section 2.7.1, a file system block cache stores blocks of files in memory so requests can be handled by memory without going to disk. That is, read blocks can be provided by the cache instead of the disks, if the access "hits" a block in the cache. In addition, the cache can be used to delay writes, so in some cases writes will "die" in the cache by being overwritten or deleted before the block is flushed to disk, reducing the disk traffic. Thus, a block cache can improve both read and write performance.

Controller memory could be used to implement a file system block cache. This cache would operate similarly to a normal block cache, except the blocks would be in controller memory instead of file system kernel memory. In this controller memory block cache, reads and writes would be broken into blocks and would pass through the cache in controller memory on their path between disks and network. A block cache in the controller would provide performance benefits similar to a traditional block cache, by servicing requests at memory speed rather than disk speed.

With a controller memory block cache, the data blocks would be stored in the controller memory, while the information about the blocks would be stored in file server kernel memory for use by the file system. That is, the file server would hold a hash table for looking up blocks, as well as information about each block such as validity, the modification time, the file that contains the block, a dirty/clean flag, and locking information for the block.

One important difference between a regular cache and a controller cache is how the controller cache interacts with the file server's cache. The first alternative is to use controller memory for an entirely separate cache. The second is to integrate the controller cache with the file system's regular kernel memory cache. These alternatives were discussed in Section 3.8.

A second, related difference between a regular cache and a controller cache is that the file system can't directly access metadata stored in the controller cache. This affects directory blocks and indirect blocks that a normal file system may access through the cache. With a controller cache, this access path may not exist. The metadata path used in Sawmill is discussed in Section 4.5.

4.2.1. The decision against a cache

The Sawmill file system doesn't currently use a block cache because of several serious disadvantages to the cache approach. First, the block cache requires significant CPU processing overhead. Second, breaking requests into cache-sized blocks reduces the possibilities for performing large, efficient transfers. Third, the memory available for a cache would be too small for a useful cache. Finally, client caches can already provide most of the performance benefits of the cache.

The first reason to eliminate the block cache is that a block cache requires a significant amount of CPU overhead for each block to check whether a block is in the cache and to update the cache information. This is especially a problem with the small blocks typically used in workstation file systems. If the file system must handle 20 MB/s of 4 KB blocks, then 5000 cache blocks per second must be processed, or 200 µs. For comparison, performance measurements on the Sun 4 used for Sawmill show that looking up an existing block in the cache takes about 80 µs, while creating a new cache block takes about 200 µs due to the additional overhead of discarding an existing block. Thus, the cost of creating cache blocks alone would take the entire processing power of the system. With a faster CPU, though, the cost of processing blocks would not be as significant.

A second disadvantage of using a block cache is that breaking large requests into cache blocks reduces the ability of the file system to perform large transfers. This problem is much more important with a disk array than with a single disk, because a disk array gets much better performance with large transfers that can be striped across multiple disks. This problem is also more important for high-bandwidth storage systems than for typical file servers since network bandwidths are also generally better for large transfers. Thus, while block-sized transfers are suitable for lower-bandwidth systems, to get peak bandwidth out of a disk array storage system transfers should be as large as possible rather than limited by block size. This will be discussed further in Section 4.3.5.

The third reason against using controller memory as a cache is the effectiveness. It is likely that applications that require a high-bandwidth file server will also require a large cache. With RAID-II, the memory available for a cache would be about 20 MB; this is probably too small to provide a high hit rate with high-bandwidth applications. Note that with a data rate of about 20 MB/s, the lifetime of cached data would be only one second. That is, after about a second, the cache will be replaced with entirely new data. This yields a very small window in which cached data can be reused. In comparison, the existing Sprite file server has a 100 MB file block cache and supports data rates of under 1 MB/s, leading to a lifetime of hours [Bak94] for many cached blocks.

The performance of the controller cache compared to a traditional file system block cache depends, of course, on the application access patterns and how they scale with the data rate. If the high-bandwidth clients have the same access patterns as low-bandwidth clients, just much faster, then the cache size does not need to increase with the bandwidth to maintain the same hit rate. It is more likely, however, that the applications using a high-bandwidth server will need more memory to maintain the same rate, for two reasons.

First, one application for a high-bandwidth file server is to support many more clients than a low-bandwidth server can support. In this case, the additional clients are unlikely to all share the same data, so the hit rate will decline. The exact decline in hit rate depends on the access patterns for the clients, since they could all use exactly the same files, but the hit rate will typically decrease.

Second, a high-bandwidth file server may service clients that use much larger amounts of data than before. For instance, supercomputing applications may operate on much larger data sets. Also, new applications such as multimedia and video require very large amounts of data without much locality. In these cases, the cache is unlikely to provide much performance benefit, even if it is large, since data is typically only read once.

Even with a relatively low hit rate, though, it may be worth providing a cache for the performance improvement. In a system with more processing power, a controller cache might be beneficial with even a low hit rate. For example a 5% hit rate could provide close to a 5% performance improvement; this is small, but it may be worthwhile in some cases. With the RAID-II configuration, however, the performance loss due to the processing overhead would cancel out the benefits from the cache.

These factors suggest that a cache of several megabytes would probably not be large enough to provide high hit rates for Sawmill under most workloads. With more memory available for a cache, or more CPU power to support the cache, the tradeoffs would lean more towards providing a file system cache.

For some workloads, one way around the cache size problem would be to cache only some data. That is, the data that is likely to be reused would be cached, while other data, for example high-bandwidth streams, would not be cached. This would work best on a system that has a mixed workload of very large requests that are not reused (such as video-on-demand), and small requests that are reused (such as normal file system accesses). The cache would provide a performance benefit for reused data, without being flushed by the high-bandwidth data. There are several ways that this dual-mode caching could be implemented. The most effective would be to have the application specify if the stream should be cached or not. The downside of this is that it would require modifications to the programs to provide this information. A second technique would be for the file system to examine the access patterns to decide if a file should be cached or not. For example, one way would be to cache requests under, say, a megabyte. Another would be to cache only accesses to small files.

Finally, client caching can provide most of the potential benefits of a server data cache. By providing caches on each client, the cache sizes are not limited by the server architecture. In addition, client caches can improve performance more than server caches because cache performance is not network limited, since requests are serviced from local memory. That is, client cache hits do not require network traffic as file server cache hits do.

The main performance disadvantage of client caches compared to a server cache is in an environment where many files are shared among clients. Each time a file is first accessed by a client, the result is a miss for a client cache, but accesses after the first would be hits in a server cache. In addition, more cache memory is needed in total with client caches, since each client can have a separate cached copy of the file. With a server cache, only a single copy of the file will be stored.

In conclusion, due to the CPU overhead, the performance loss from block-sized transfers, and the limited hit rate expected from the cache, a server cache in the controller memory was not implemented in Sawmill. Instead, the memory is used for other types of data buffering, which will be discussed in the next section. Client caching could provide most of the benefits of a server cache, without the problems associated with a server cache.

4.3. Alternatives to caching

In place of a block cache, Sawmill uses several techniques to manage the controller memory and improve read performance. Sawmill pipelines transfers in order to overlap disk and network activity. Sawmill uses prefetching to minimize the latency of successive sequential requests. Sawmill also batches together out-of-order reads, so multiple reads can be replaced by a single read. Sawmill performs some caching in the form of a segment cache. The idea behind these techniques is to replace the traditional file system approach of acting on individual cache blocks with an approach that treats requests as large streams of data movement that can be handled efficiently. This section describes these uses of controller memory and their effects on performance.

4.3.1. Pipelining

The Sawmill file system uses controller memory for pipelining read requests. That is, for large reads, disk operations are overlapped with network operations to hide the latency of disk and network operations. Pipelining in Sawmill uses multiple read buffers so data can be read into one buffer while another buffer is being sent across the network. That is, for a read of more than one segment, as soon as the first segment is read, the data can be sent out over the network while the next segment is being read. This is in contrast to the old Sprite file system, which reads the entire amount of data to satisfy a request before any of it is sent.

The implementation of pipelining in Sawmill was explained in Section 3.7.1. By processing disk requests and network requests asynchronously, pipelining occurs automatically, since data starts going over the network as soon as a buffer is ready. Currently, the Sawmill file system uses a segment as the unit of pipelining, but this could be easily modified.

4.3.2. Prefetching

With prefetching, as discussed in Section 2.7.2, data blocks are read into memory before they are requested so later reads for them can be serviced without waiting on the disk. That is, the file system predicts which blocks will be needed, and starts loading those blocks into memory, so if the blocks are requested later they can be provided with low latency.

Prefetching can be done for several types of access patterns. Prefetching is easiest for sequential accesses, since the file system can simply prefetch the next bytes of the file in order. That is, the file system assumes that the application will read additional sequential bytes of the file, and these bytes are read into buffers before they are requested. Client hints (discussed in Section 2.7.2) can be used to inform the file system when sequential prefetching will be useful and how much data should be prefetched. For nonsequential accesses, the application can provide hints to the file system to indicate what data addresses to prefetch. Alternatively, asynchronous I/O can be used as a mechanism for the application to control prefetching; the application can issue a request before the data is required, giving the file system time to prefetch the data while the application performs other computation.

Prefetching is more important for a file system using a disk array than a file system running on an individual disk. A large prefetch read can replace many small reads, resulting in larger transfers. While a single disk performs better with large reads, the improvement is much more dramatic for a disk array, since the array can operate in parallel on the large request. For example, if the disk array processes sequential requests smaller than the stripe unit size, the requests will be serviced by a single disk at, say, 1.6 MB/s. If the disk array prefetches entire parity stripes, they can be read with the full disk array bandwidth of over 20 MB/s and then rapidly transferred to the client. Also, because the bandwidth of a disk array is higher, latency results in much more wasted potential. For example, suppose a client has 5 ms network latency. In this time, at 1.6 MB/s the disk could have read 8 KB. A disk array operating at 20 MB/s, however, could have read 100 KB. Thus, it is much more important with a disk array to use prefetching to avoid wasting potential bandwidth, and more data must be prefetched.

The two main issues with prefetching are "space" and "time". The question of "space" is how to use available memory wisely for prefetching. The question of "time" is when to prefetch and how to allocate disk time for best performance. Space is especially important on storage systems with controller memory, since as explained earlier, this memory may be much smaller than the memory of a typical file server. The time trade-offs also change on a high-bandwidth RAID server as compared to a traditional file server since much more data can be transferred during a time interval.

One issue with space and prefetching is how to ensure that prefetching doesn't result in more-useful blocks being discarded to make room for the prefetched blocks. Normally a file system cache replaces data using a least-recently-used approach. That is, the data in the cache that has been least recently used is discarded to make room for the new data, on the assumption that the discarded data block is the least useful. If data is prefetched, however, this is usually the wrong thing to do. For example, suppose data is being prefetched into memory. The data that should be discarded is the data that has already been read out of the cache, since it will not be used again under a sequential access pattern. However, the least recently used data is the oldest prefetched data that hasn't been used; this block will be read next in sequence. Thus, if memory fills, the least recently used data is the worst choice for discarding. Therefore, prefetch data should be discarded according to a most-recently-used policy, not least-recently-used.

A related question is how to partition memory for prefetching, given multiple clients and multiple request streams. One method would be for each client (or each stream) to have a fixed amount of space for prefetching. This maximizes fairness, since one stream can't deny another the ability to prefetch. However, it probably would not be the most efficient approach, since streams may have widely differing access patterns and data rates.

For instance, if one stream is performing entirely sequential accesses, while another stream is performing a mixture of sequential and random accesses, prefetched data for the former stream is more likely to be useful.

An alternative would be to allocate buffers to the streams that are making most active use of them. This would improve performance if, for instance, fluctuations in server load occur: the more active stream would need more prefetched data to cover the gap while the server is overloaded. Two techniques are possible for handling this competition over memory space. One is to decide how to allocate buffers as they are freed up. Another is to actively take buffers from other streams, discarding their prefetched data. On the one hand, stealing full buffers can severely damage prefetch performance, if the disks are busy loading data that gets discarded before it is used. On the other hand, if a slow stream has many buffers allocated, it makes sense to reallocate them, even if some data gets discarded in the process. A compromise is to not steal a buffer if the stream is making active progress towards accessing it, but to allow the buffer to be stolen if the stream starts making non-sequential accesses or a period of time goes by.

Instead of these approaches, Sawmill currently assigns buffers in first-come-first-served order for simplicity. A more complex algorithm could be implemented and would probably improve performance with multiple streams.

The second issue for prefetching is "time": when to prefetch and how to allocate the time available for prefetching. Even given unlimited memory for prefetching, the limited disk bandwidth also constrains when to prefetch and what should be prefetched.

The file system should avoid prefetching if there are still parts of a request outstanding. That is, if a stream has requested data, the file system should read this before doing any prefetching. Otherwise, performance will suffer as requests wait on prefetches. This will hurt latency, since active requests will be waiting. In addition, bandwidth will suffer as the disks read data that is only potentially useful, rather than reading data this is known to be useful.

In the Sawmill file system, prefetching is handled by the read code in the request manager. This code provides pipelining and prefetching by allowing disk and network operations to be overlapped and to take place out of sequence. Figure 4-2 illustrates how the read code operates. The fundamental idea is that Sawmill issues disk reads and network sends independently, by using the event-based structure. When a new event indicates the completion of a disk or network operation, a synchronization module decides if a network send can now take place, if a new disk operation should be started, or both. This approach provides pipelining because disk and network operations can overlap, with data being sent as soon as possible. It also provides prefetching because disk operations can be started before a request arrives for the data; if a request later arrives, the data will be sent. Thus, the policy decisions of when to prefetch are encapsulated in the read synchronization routine, which decides when to issue disk requests.

In more detail, a request stream in Sawmill uses two data structures to keep track of read progress, as shown in Figure 4-3. The first structure keeps track of what region of the file has been requested by the client, what part of the request has been serviced, what has been

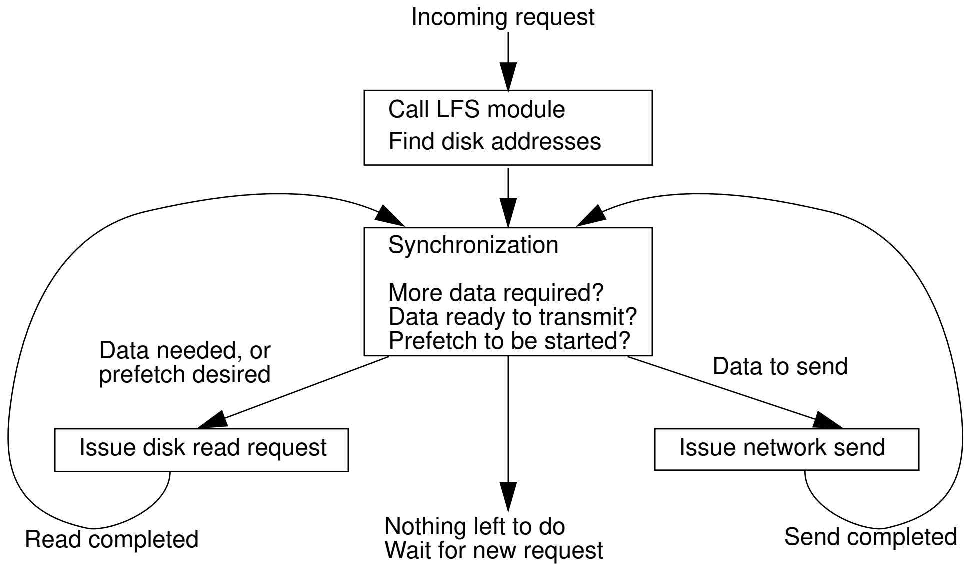


Figure 4-2. Flow of control for reads

Disk read requests are handled with overlapping disk reads and network sends. The key to read processing is the synchronization routine, which determines if any fetched data can be sent over the network and if any additional reads from disk are required. It can issue both disk and network operations in parallel. The synchronization module also looks after prefetching, which is done by issuing disk reads beyond the required requests. Note that because of the event-based structure, network sends and disk reads can happen simultaneously.

---

requested from the disks, and what needs to be sent to the client. The second structure contains the blocks that have been read off disk into a buffer. These two lists are matched up and updated to satisfy the request. That is, when a read request comes in, it is first sent to the LFS module, which determines where on disk the data is stored. The request manger then starts issuing the appropriate disk requests. As disk reads complete, they are matched up with the outstanding data required by the client and the data blocks are sent to the client. At this point, new disk operations can be started. When interrupts indicate the network send has completed, the request manger checks if the next requests can be completed.

The read process in Sawmill can be contrasted with reads in Sprite. The existing Sprite file system handles these synchronously; a read request first reads all the data into the cache and then sends all the data over the network. However, this prevents overlapping of disk and network activity for a particular request. By using the asynchronous event-handler, disk and network activity for a single operation can both take place at the same time. While it would be possible to have overlapping traffic of this sort in a synchronous model, it is easier to achieve in the asynchronous model.

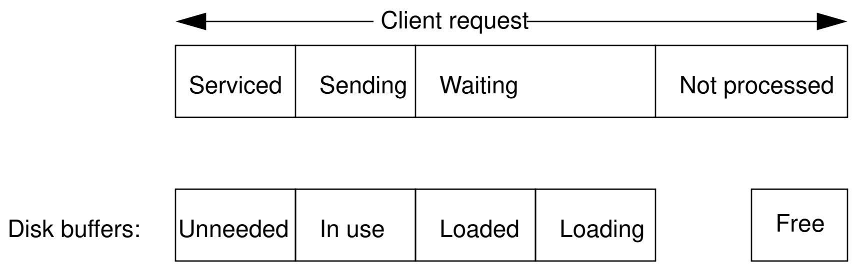


Figure 4-3. Read data structures

This figure illustrates the data structures used to handle reads in Sawmill. In the above figure, a client is performing a large read. Part of the data has been sent to the client, part is currently being sent, and the remainder is still being handled. For each outstanding request, Sawmill keeps track of how much of the request has been handled, what is being sent over the network, what is coming off disk, and what remains to be handled. Each disk request buffer has a data structure that indicates the state of a buffer. A buffer can be empty, contain valid data, or be in the process of being loaded through a disk read. A buffer can also be needed for the client request, be in the middle of a network send, be wanted for a prefetch, or be unneeded. As disk and network events occur, the read code updates these data structures and uses them to determine what to do next.

4.3.3. Batching of reads

A related method of using controller memory to improve read efficiency is to batch multiple reads into a single operation. Unlike prefetching, which reads data that has not been requested, batching attempts to read data that has been requested, but handles the request as a large disk operation. For example, in a typical Unix file system, a large read will be broken into block-sized disk operations. These small operations would be very inefficient on a disk array. Sawmill, in contrast, batches these block operations together and performs them as a large, efficient read. Batching requests together minimizes the performance cost of the seek and rotational latencies of the disk, and reduces the processing overhead.

For blocks stored sequentially on disk, it is straightforward to group consecutive reads together into a single large read. Unfortunately, the blocks of a logically sequential request might not be stored sequentially in the log. This is especially a problem when “on-the-fly” layout is used, as in Sawmill, because blocks are written in the order they are received, rather than in their order in the file.

Fortunately, there will often be some locality of writes, so the data may be stored in the same segment, even if the blocks are not stored sequentially. That is, sequential blocks will often be written at about the same time, even if they are not exactly in sequence, and they will end up in the same segment. In this case, it would be more efficient to read the entire segment or a large fraction of the segment and then send the pieces from memory over the network in the proper order, rather than perform multiple disk operations to fetch the blocks in order. Thus, by batching together reads and buffering them in controller memory, read performance can be improved.

Figure 4-4 illustrates several cases in which reads could potentially be batched together. In the first case, the blocks are stored sequentially in the segment, so it is straightforward to read them all at once. In the second case, two factors intervene: there is an unwanted block in the way, and some blocks are out of order. In the final case, there are so many blocks in the way that grouping the reads together would harm performance; it would be better to do two reads separated by a seek. It can be seen from Figure 4-4 that there are several performance tradeoffs in deciding when to batch together reads. For contiguous reads batching is clearly advantageous, but when unneeded blocks intervene the decision to batch becomes more complex since it depends on how the time to read unneeded data compares with the time to seek and handle a new request.

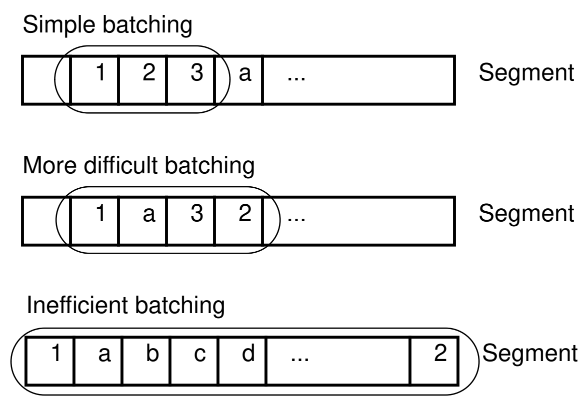


Figure 4-4. Batching of reads

In many cases, multiple reads can be batched together to improve performance. In the first case, batching together the three reads would help performance. In the second case, batching might help, while in the third figure batching is unlikely to be worthwhile. In each case, three logically sequential blocks are being read from the file.

In Sawmill, a fairly simple algorithm is used to group reads. The file system loops over each block in a request, checks its position on disk, and determines if the gap between it and the previous block is smaller than a fixed limit. When it collects as many blocks as possible, it reads the segment into memory as a single large read. The original smaller read requests can then be serviced from the segment in memory. The maximum allowable gap between batched blocks in Sawmill was arbitrarily set to three blocks.

More complex algorithms are possible to improve performance when reading out-of-order data. For instance, batching may be combined with prefetching to read blocks before they are requested. Suppose a read request accesses some blocks of a file. If the next blocks of the file are stored sequentially in the segment, then the prefetch can be batched together with the original request and performed as a single read. In addition, the degree of batching could be controlled to prevent batching from increasing latency excessively.

The grouping of reads described above happens on the segment level; that is, file block reads to a particular segment are examined and may be grouped together. This is in contrast to [MK91], which also performs batching, but does the grouping on raw disk blocks. One potential disadvantage of performing the batching inside a segment is that two reads in different segments may physically be nearby on disk, for example to the last block in one segment and the first block of the next segment; low-level batching would detect this, but segment-level batching would not. This type of batching unlikely to occur in practice, since segments are not allocated sequentially on disk. The main advantage of doing the block grouping at a high level, as in Sawmill, is that the number of low-level system operations is minimized since multiple operations don't get passed down to the low levels, as shown in Figure 4-5. This reduces the latency as well as the processing overhead.

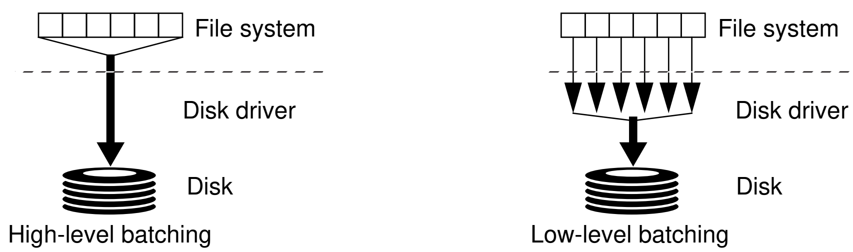


Figure 4-5. High-level vs. low-level batching

This figure illustrates the difference between grouping operations together in the file system versus grouping them in the disk driver. By batching them together at a high level, fewer procedure calls to the device driver occur, reducing the latency and CPU overhead.

4.3.4. Segment cache

Although the Sawmill file system does not use a block cache, it does use a different type of caching. Rather than caching file blocks, Sawmill uses a *segment cache*. In a segment cache, entire LFS segments are cached in memory, rather than individual blocks; these segments are typically a megabyte in length, much larger than cache blocks.

The underlying motivation behind a segment cache is locality, that references to a segment are likely to be followed by more references to the segment. In contrast to a block cache, which requires references to the same blocks, the segment cache provides benefits for references to different blocks in the same segment. In this way, the segment cache goes beyond what a client or server block cache could provide.

There are several advantages to a segment cache over a block cache. First, the processing cost for a segment cache is very small compared to a block cache, since the number of cached segments to be processed is much smaller than the number of cached blocks would be. That is, the segment cache must be updated once per segment of, say, 1 MB, compared to once per block of, say, 8 KB for a block cache. Second, because segments are already built up in memory for writes, there is little additional cost to keeping the segment around. Third, for disk array reads, a large segment can be read into the cache efficiently, compared to the cost of reading individual blocks. Finally, the file system must be able to provide data out of a write segment before it goes to disk in order to provide write consistency. This mechanism can also be used to provide data from a cached segment.

The original Sprite implementation of LFS also uses segment caching. This cache, however, is very small, holding only a single segment that has been read in to be cleaned. The performance of the Sprite-LFS segment cache, however, is very poor. Less than 1% of read requests can be handled by the segment cache. Two factors account for the low hit rate. First, the cache is very small, holding only a single segment. Second, the cache doesn't fit access patterns well; a segment being cleaned is unlikely to happen to be a segment that is also being read. This is in contrast to a normal block cache, which holds blocks that are in active use.

The segment cache in Sawmill goes beyond the Sprite-LFS segment cache in two ways. First, it holds multiple segments rather than just the last segment written. Second, it holds data loaded from reads and completed write segments, rather than segments being cleaned.

The implementation of the Sawmill segment cache is fairly straightforward. If a Sawmill read request uses a large amount of a segment, the entire segment is read into the buffer. There is a trade-off of when to read only the requested data and when to read the entire segment. In Sawmill, this is determined by an adjustable threshold, currently set at half a segment; if the amount of a segment being read is larger than this threshold, then the entire segment is read.

The segment cache is managed as a small pool of buffers, used in least-recently-used order. In addition, these buffers can hold partial segments, so the same buffer mechanism can handle the batching of reads as discussed in the previous section. That is, a buffer can hold an entire segment and be used as a cached segment, or a buffer can hold part of a segment to service batched reads. Sawmill currently uses 10 buffers; since a buffer must hold a full segment (960KB), the number of buffers is limited by the total controller memory.

4.3.5. Streaming instead of caching

This section and the previous section have discussed several different methods for using memory to help efficiently read data: a file system block cache, pipelining, prefetching, read batching, and a segment cache. These techniques all take advantage of controller memory to improve performance. Table 4-1 summarizes these techniques and how they were used in Sawmill.

File system block cache	Rejected due to high processing cost and low expected benefit.
Pipelining	Used to hide latency.
Prefetching	Used to hide latency.
Batching	Used to improve transfer efficiency.
Segment cache	Used to speed up multiple references to a segment.
Streaming	Used to improve transfer efficiency.

Table 4-1. Memory usage design decisions

This table summarizes the techniques for the use of controller memory that were discussed in this section and why they were used or not used in Sawmill.

One of the overall design goals in Sawmill is to treat data movement as a continuous stream, rather than as individual blocks. That is, whenever possible, the file system keeps a large request together to provide efficiency in disk transfers, network transfers, and processing overhead. Segment caching, read batching, pipelining, and prefetching provide an important advantage that a block cache does not: they allow transfers to take place in large units, rather than in individual blocks.

By combining the various techniques discussed above, many transfers can be turned into large streams, even if they were originally block-based transfers. For example, prefetching allows data to be accessed in larger units than the original requests. Similarly, batching of reads provides large transfers where they would not otherwise be possible. Chapter 5 will illustrate layout techniques to allow writes to take place as efficient streams rather than individual blocks.

Because large transfers are more important for high-bandwidth systems such as disk arrays than for traditional file systems, the advantages of a cache-based approach become less important compared to the disadvantages of the smaller block transfers. Thus, Sawmill uses the stream-based approach rather than the cache-based approach of most workstation file systems.

4.4. Memory allocation

The Sawmill file system must allocate memory among the various potential uses described earlier. Memory must be assigned to the prefetch buffers, segment cache buffers, and so forth. The two problems of memory allocation are how to allocate memory among these competing uses, and what to do if there isn't enough memory. These issues are more of a problem with controller memory than file server kernel memory because of the relatively limited size of the memory in the RAID-II controller. Thus, ensuring that this memory does not run out is more important.

There are several alternatives for memory allocation. They can be divided into *static* allocation, where memory usage is fixed at start-up time, and *dynamic* allocation, where memory usage is flexible and changes during use.

Static allocation is conceptually straightforward. The available memory is partitioned into fixed amounts for the various types of uses. For example, two log write buffers and two segment cache buffers could be assigned, and the remainder of available memory used for prefetch buffers. The main advantage of static allocation is the simplicity, since the buffers can be allocated at start-up time, rather than using a run-time memory manager. A second advantage is speed, since no time will be spent performing memory allocation. The disadvantage of static allocation is that it is inflexible; memory usage cannot change with access patterns.

In dynamic allocation, the different uses allocate memory from the general pool, so memory is shared flexibly among the various types of buffers. Although dynamic allocation is more flexible and can adapt to changing usage patterns, it is more complex. Some arbitration mechanism must decide how to apportion memory among the competing uses in order to maximize performance. In addition, dynamic allocation has a performance penalty due to the overhead of managing memory. Finally dynamic allocation can encounter difficulty with running out of memory as usage patterns change; the allocator may find that it has over-committed memory to the wrong type of buffer and doesn't have enough free to immediately service a necessary request.

Sawmill uses static allocation of the various types of buffers. The main motivations for this were the simplicity, the reduced overhead, and the difficulty of knowing how to best partition memory. One topic for future work is to examine the tradeoffs of memory usage to determine how these uses could be balanced during use.

In either case, one key issue is what to do if a buffer is unavailable. In the static case, this happens if all buffers of some type are allocated; in the dynamic case, this happens if no free memory is available. If a buffer is unavailable to satisfy a request, there must be a mechanism for the request to wait until memory becomes available. One very important factor is to avoid deadlock, where two processes mutually wait for each other to free a resource before they can make any progress. A second issue is to avoid starvation, where a particular request can wait arbitrarily long before being serviced.

The event-driven model of Sawmill makes waiting for a memory buffer somewhat more difficult than in a standard procedural model. Figure 4-6 compares these alternatives. In the procedural model, a routine can simply call a buffer manager that will block until a buffer becomes available. In the event-driven model, routines are not permitted to block, since there is a single thread of execution. As a result, if the request would block because a buffer is not available, all the state for the request must be packaged up so the request can be suspended, to be continued by a new event when a buffer becomes available. For each buffer type allocated from a pool, there must be a queue holding the waiting requests. When a buffer becomes free, a request is removed from the wait queue and the waiting request handler is added to the event queue. Thus, the event-driven model requires considerably more complexity in the code to handle waiting on buffers. In contrast, since the procedural model can use wait and wakeup calls, the CPU scheduler handles the queueing for buffers without explicit management in the file system.

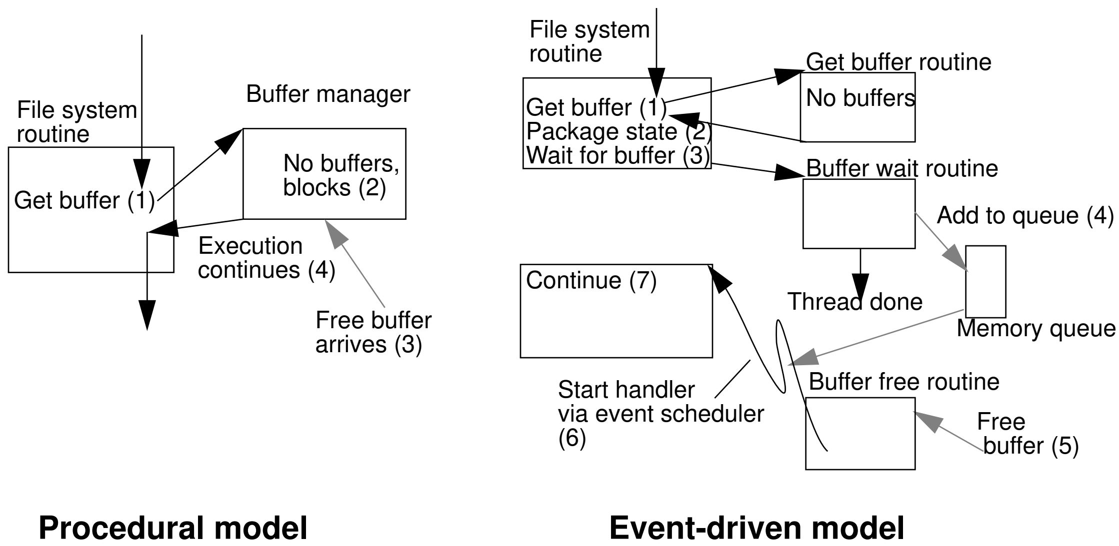


Figure 4-6. Waiting for a memory buffer

If a memory buffer is not available, the request must wait for a free buffer. This figure shows how a procedure-based implementation and an event-driven implementation wait for a free buffer. In the procedural model, a routine calls the buffer manager (1) to get a buffer. If none are available, the thread blocks (2) until a free buffer arrives (3). The manager then unblocks and execution continues (4). The event-driven model is much more complex. If the routine cannot get a buffer (1), it packages up the request state (2) and calls the buffer wait routine (3). This routine adds the request to a buffer queue (4). When a buffer arrives (5), the queued request is passed to the event scheduler (6) so execution can continue (7).

Deadlock avoidance is simpler in the static case than the dynamic case. With static buffers, two approaches are possible. For some buffers, the file system can ensure that there are always enough buffers available. For instance, with network request buffers, it makes sense to assign a buffer to each request stream, rather than constantly allocating and freeing them, since each request stream will be making heavy use of the buffer. For other buffers, by ensuring that buffers are allocated in a particular order, a circular hold-and-wait cannot arise, so deadlock is prevented. In Sawmill, each stream is allocated critical buffers at open time so, for example, network operations can always complete. Thus, the only buffer that a request can wait on is the read buffer for data coming off disk. Since only one buffer is required to handle a request, some stream can always finish and then the buffer is returned to the read buffer queue. This avoids both deadlock and starvation.

In the dynamic case, deadlock is harder to prevent, since different requests could all end up waiting for memory. Two techniques are possible here. The first is to limit the number of concurrent requests to the available memory. That is, make sure a request can get the memory it needs before it is started. The second is to allow unnecessary buffers to be preempted if memory is running out. For example, prefetch buffers and cached segments could be discarded at any time, and their memory used for other purposes.

The event-driven model simplifies deadlock avoidance compared to a procedural model. In the event-driven case, the state is explicitly packaged up before waiting; at this point, the file system can ensure that the thread isn't holding resources that another thread requires. In the procedural case, waiting is implicit, so it is more difficult to ensure that a thread won't block while holding a needed resource.

Starvation occurs if one request ends up waiting indefinitely for other requests to complete, while other requests continue to take place. One way this could occur is if one request is waiting for a buffer, while other request streams have the buffers and don't relinquish them. In Sawmill, starvation is avoided by making sure that buffers are freed after use and that requests go onto a first-come-first-served queue. Thus, any waiting request stream will eventually get the necessary buffers.

4.5. Metadata movement

Metadata consists of information other than raw data that gets stored to disk. Besides handling file data, the file system must handle various types of metadata such as information on files (inodes), information about where blocks are on disk, the directory structure of the file system, and information about the contents of LFS segments. The important factor about metadata is that it must be accessed by the file system in order to provide file operations. Metadata must both be read off disk, such as when a pathname is looked up, and written to disk, such as when a directory is changed.

Metadata poses several problems on a controller architecture that doesn't move data through the file server, since the file server must be able to access metadata. In a traditional architecture, the file server can immediately access any data it reads. With a bypass architecture such as RAID-II, however, data normally is transferred between the disks and network via controller memory without ever entering the server. Thus, if the server wishes to access metadata, it must copy the data from the controller into its own memory. This can become a performance bottleneck, if the file server accesses metadata slowly. In addition, metadata complicates memory usage because metadata may be stored both in controller memory and server memory. This section discusses the metadata processing used in Sawmill.

Several types of metadata are used in Sawmill. Some are general file system metadata:

- An inode (called a file descriptor in Sprite) is associated with each file and holds information about the file, such as the permission bits, modification time, and disk addresses of the file's blocks.

- Large files have indirect blocks, as discussed in Section 2.4.1, to hold additional file block disk addresses.
- Directories are used to convert symbolic file names to file descriptors. Directories are stored as files in the file system, even though they are used as metadata.

Besides the general file system metadata, several metadata structures are associated specifically with management of the log-structured file system:

- Each segment of the log has segment summary information describing the segment's structure for use in cleaning.
- Checkpoints are a structure stored on disk for use in crash recovery. The LFS stores directory modifications in a directory log.
- Two in-memory LFS data structures are backed up on disk: the segment usage map, which keeps track of how much free space is in each log segment, and the descriptor map, which indicates the position of the most recent inode for each file.

The two key issues with metadata processing in Sawmill are efficiency and consistency. In any file system, accessing metadata off disk is much slower than accessing metadata that is cached in memory. With RAID-II, however, even accessing metadata stored in controller memory is relatively slow, since the VME link between the controller and the file server has a high latency (about 700 μs) and moderate bandwidth (about 4 MB/s) as will be discussed in Section 6.1.1. Thus, to provide efficient access to metadata, metadata is cached in file server memory.

Caching metadata in the file server results in the second problem, consistency. Metadata may reside in controller memory for several reasons: while being read off disk, while being sent to a client (to provide file attributes or directory contents, for instance), or while a segment is being prepared for writing to disk. The Sawmill file system must maintain consistency among metadata in file server memory, metadata in controller memory, and metadata stored on disk.

Most metadata is cached in the file server using the standard Sprite file system block caching structure. That is, blocks of metadata are read into the block cache when metadata is accessed, and are written to disk when required. Sprite uses this mechanism for indirect blocks and directory blocks.

Sawmill uses this mechanism, but with modifications to handle the separate controller and file server memories. The main change is that each block is tagged to indicate if the most recent copy is in controller memory or in file server memory. Then, whenever the file system accesses a block, it passes the cache code a flag indicating whether the block is needed in controller or server memory. The cache code then copies the block from one place to the other as necessary. Thus, by maintaining a single copy of the block, consistency problems are eliminated.

For increased efficiency, Sawmill caches parts of the inode block map; this map specifies where each block is located on disk. In Sprite-LFS, each block is looked up individually, which involves a cache search and locking and unlocking of the block. With Sawmill, because the data rates are much higher, looking up blocks became a performance bottleneck. Caching of the block map, however, eliminated the bottleneck. For each open stream, Sawmill performs the lookup once, efficiently processing a group of blocks together. Sawmill caches the block indices in the stream's in-memory structure, so multiple blocks can be accessed without reading the block map explicitly. Cached block addresses must be invalidated, however, whenever the file is written, since the block locations will change.

In addition to server caching of metadata, client metadata caching could also be implemented. For example, clients could cache directories and file attributes, allowing name lookup to occur locally. Client caching minimizes the load on the server and network, and reduces the time for clients to perform lookups, but makes the consistency policies more complicated. Simulations of client caching of name and attribute information have shown that even a small client cache dramatically reduces the number of metadata operations that must be processed by the server [SO92]. Client metadata caching could be added to the Sawmill file system, but is not discussed in detail here; client name and attribute caching is discussed in more detail in [SO92].

Besides the approach taken in Sawmill, an entirely different approach would be to keep metadata on a separate disk. That is, instead of storing metadata and data together on the controller's disk array, metadata would be stored on a disk attached directly to the file server while only file data would be stored on the disk array. There are several advantages to this approach. First, this would increase the total data bandwidth available off the disk array, since the disk array would no longer have to devote any of its bandwidth to reading and writing metadata. In particular, small metadata requests would be handled by the separate disk, leaving the disk array for the large requests that it does best. Second, metadata could be accessed directly by the file server, rather than having to copy it through the controller over the VME link. Finally, disk layout could be improved, since data would no longer be interspersed with metadata.

Storing metadata on a separate disk, however, has several disadvantages. First, the file system becomes considerably more complex, since it now stores data on two separate disks. Second, balancing the system for peak performance becomes more difficult, since the metadata disks must be fast and large enough that they don't become a performance bottleneck, but if they are too fast and large they are wasting resources. This raises interesting questions for a log-structured file system, since of the metadata disk and the data disk, either or both could be log-structured. Finally, network client access to metadata would become slower and more difficult, since it would have to come off the metadata disk and be copied down to the controller. These accesses would be required fairly often, since metadata accesses (to stat a file or read a directory) are common. [BHK+91] found that about 0.5% of file traffic goes to directories, and the measurements in [SO92] show that about 1 name lookup occurs for about every 2 KB of file traffic, mostly due to `stat` and `open` calls. Thus, it is important that metadata accesses be efficient, or else total system performance will suffer.

4.6. Maintaining consistency

Because data and metadata may be stored in several locations in the Sawmill file system, some consistency mechanism must manage these copies of data to ensure that an out-of-date copy doesn't get used. This must be done, however, without excessive overhead, or else the performance benefits of the file system will be diminished.

Four types of consistency are relevant to Sawmill. First, multiple copies of data stored in the controller memory must be kept consistent. Second, any copies of data cached on clients must be kept consistent. Third, the split cache may result in a copy of data in file server memory. Finally, metadata must be kept consistent between the controller and the kernel.

4.6.1. Controller data consistency

Multiple copies of data in the controller can arise because reads can be stored in memory in prefetch buffers and writes can be stored in the segment write buffers. As a result, reads could obtain an old copy of the data. This read-after-write inconsistency could potentially occur in several ways.

First, suppose a write is stored in the segment log in memory, but hasn't gone to disk yet. Then, a read from disk of this data would access the old on-disk copy of the data rather than the new. This isn't a problem in the Sprite-LFS, since both reads and writes are integrated with the block cache, so the read will get the current cached copy rather than the old on-disk copy. In Sawmill, however, neither reads nor writes use a block cache, so a problem can potentially arise.

A second type of read-after-write inconsistency arises when the write updates the on-disk copy of the data, but there is an old in-memory copy. This copy could be a cached segment, a prefetch buffer, or part of a large read. If these locations have an old copy of data, a read request could receive an old copy of the data.

Another potential type of inconsistency is write-after-write inconsistency. This occurs when two writes to the same data occur, but the earlier one is kept rather than the later. This is not a problem in Sawmill; because the log-structured file system keeps track of the latest copy of a written block, multiple copies of write data will not result in inconsistency for writes. Even if both copies of the block appear in the log, the file's inode will point to the logically later one on disk. This does require, however, that writes are processed by the file system in the order they arrive.

There are several issues to maintaining consistency. The main issue is how to perform consistency efficiently. One technique would be to have a data structure that indicates for each block of the file where the block is kept in memory. Then, for each block that is read, the data structure would be examined to ensure that the most recent copy is obtained. For writes, the data structure would be modified to indicate the location of the new copy. In a traditional file system, the block cache is such a structure, since all requests normally go through the cache. The disadvantage of this is that it requires processing for every block of the file. In Sawmill, per-block operations are avoided whenever possible, to permit high-bandwidth file access while minimizing the CPU as a bottleneck.

In Sawmill, consistency checks are generally only performed when there is the potential for consistency problems. At open time, the file is checked to see if inconsistency could result. As long as a single client has the file open, and only for one of reading or writing, there won't be consistency problems. As shown in [Bak94], sharing of files is very rare; only about 1% of bytes are read or written from shared files. By skipping the consistency checks for "safe" files, the performance won't suffer in the normal case. If multiple clients have the file open, or if a client has it open for both reading or writing, then per-block consistency is applied. That is, each block accessed will be compared against the file's list of prefetched blocks and against the cached segment list. In this way, the proper copy of each block will be used. Per-block consistency is also necessary if a client opens the file for reading after the file has been written but before the log has gone to disk. This consistency mechanism provides high performance in the normal unshared case, but performance can drop significantly for shared files.

4.6.2. Client consistency

If clients keep cached copies of data, several consistency problems can arise. For example, clients can read old data if one client modifies data while another client has the old data cached. Also, if two clients write to their own copies of the same data, then the file system may lose track of which data is the current and which is logically overwritten.

Maintaining consistency of multiple client copies of data is not explicitly addressed in Sawmill. As mentioned earlier, client caching is not implemented in Sawmill, although it would be desirable in a production system. If client caching were in use, a consistency mechanism would be necessary.

Many approaches to client cache consistency have been implemented in other systems, as discussed in Section 2.7.1. One of these techniques could be added to Sawmill in a straightforward manner. For example, the Sprite mechanism could be used. In this, all opens are processed by the server. If an open could result in inconsistent access to data, then client caching is disabled or the client data is flushed through to the server, as appropriate.

4.6.3. Metadata consistency

The implementation in Sawmill of metadata was discussed in Section 4.5. This section discusses how cached metadata is kept consistent. Some aspects of metadata consistency can be taken from standard file systems; the controller memory requires additional consistency mechanisms, however.

First, inodes and indirect blocks must be kept consistent to ensure that the copies on disk and any copies in memory reflect changes. A potential problem is that an inode or indirect block could be modified in memory and then another operation could use the old version on disk or an old version it had in memory. Since the inode and indirect block caching in Sawmill is based on the Sprite metadata caching, Sprite consistency mechanisms carry over. In Sprite and Sawmill, inodes (and indirect blocks) are accessed through the file server block cache. By always accessing the inode through the block cache, file system accesses will always get the most recent version. If a copy is not in the cache, then the only copy is on disk and can safely be read.

Since directories can be cached by the file server in kernel memory but also accessed directly by clients, they pose a consistency problem. Consistency can be maintained by ensuring that changes to the cached kernel copy are immediately forced to controller memory. Thus, whenever a client reads a directory, it will obtain valid data. Since clients don't normally directly write directories but instead perform directory modifications through file system operations such as creating or renaming files, the file system can ensure both copies of metadata are kept consistent.

One type of metadata caching unique to Sawmill is the cached inode block map entries, which hold the disk addresses of blocks as described in Section 4.5. Because these entries are just used as a performance optimization for reads, they can be discarded at any time. Thus, if a file is opened for writes, the block addresses are discarded and each read address is determined from the inode, which is kept up-to-date. An alternative would be to update the cached addresses after every write. This would improve read performance following writes, but would decrease write performance. Since concurrent reads and writes are rare, this optimization was not implemented in Sawmill.

4.6.4. Split cache consistency

The split cache, which was described in Section 3.8.2, results in some Sawmill file data being stored in the kernel cache. This causes consistency problems since different versions of data could potentially reside in kernel and controller memory and accesses could obtain stale data.

Because the Sawmill file system moves data very quickly, any caching policy must not provide significant extra overhead, or performance will suffer. Non-shared files are marked so operations can take place without consistency action in the usual case. If a file is shared, each block access from Sawmill results in a check to determine if multiple copies exist. If a kernel copy exists for a read, this block is copied into controller memory to service the request. For a write from Sawmill, a cached kernel block is invalidated when the Sawmill copy of a block is updated. Thus, kernel accesses that use a kernel-cached block will always obtain current data. Finally, the kernel cache writeback code is integrated with Sawmill to ensure that dirty cache blocks are written to the Sawmill disk properly; this is discussed in the next chapter.

4.6.5. Summary of consistency

Whenever multiple copies of data can exist, consistency of these copies becomes an issue. This is especially the case in a bypassing I/O system, since data may be in controller memory as well as kernel memory. Thus, techniques to ensure consistency are an important part of the Sawmill file system.

The main problem with guaranteeing consistency is how to minimize the performance penalty. To maintain consistency in different parts of the Sawmill file system, different algorithms are used. For the most part, the Sawmill consistency mechanisms attempt to bypass expensive checks in the usual case, and only perform consistency operations when file sharing could lead to problems.

4.7. Conclusions

File systems can use memory in the data path in several ways to improve performance. Traditional file systems make use of kernel memory to cache and buffer data transfers, for instance. In a storage system with a fast "bypass" data path, however, the memory in the data path will be in the controller, rather than the kernel. As a result, the use of this memory is different from traditional file system uses.

Sawmill attempts to move data as large streams, rather than individual blocks. This allows large transfers, which are much more efficient than small transfers on a disk array and with a high-bandwidth storage system. Instead of a file system block cache, controller memory is used for several types of read buffering. This avoids the CPU overhead of performing cache operations on each block, which would limit performance. In Sawmill, read requests are prefetched into memory, network transfers are pipelined, and memory is used to batch together reads into a single request and to cache entire segments. These techniques enhance the stream-based approach by increasing the possibilities for performing large transfers.

Handling metadata is more complicated with a storage system that has controller memory, since metadata must be moved between the controller memory and the kernel memory. Sawmill uses various metadata caching techniques to make metadata handling more efficient. Inodes, indirect blocks, and directories are cached in kernel memory. Sawmill also maintains a separate cache of the disk addresses of file blocks for open files to speed up read accesses.

Controller memory also complicates maintaining consistency of file system data and metadata, since they may reside both in kernel memory and controller memory. Sawmill's consistency techniques are designed to avoid paying a high per-block cost that could limit system bandwidth. By checking at file open time if consistency problems could arise, Sawmill avoids later consistency checks for the usual case of a non-shared file. Only for shared files are slower per-block consistency checks required.

5 Efficient disk layout for writes

"It's Log, Log; it's big, it's heavy, it's wood. / It's Log, Log; it's better than bad, it's good!"

– The Ren and Stimpy Show

Write performance is an important part of storage system design. Sawmill uses a log-structured file system to improve the speed of writes, especially small writes. The techniques in Sawmill are different from previous log-structured file systems, however, for three reasons:

- Use of a RAID disk array makes small writes especially expensive. The size of a write should match the parity stripe size for best performance.
- The "bypass" data path of the controller requires new data movement techniques to take advantage of the data path.
- The high bandwidth of the storage system requires per-block processing overheads to be minimized.

This chapter discusses three issues related to the layout of write data. First, the physical layout of data on disk is very important, both for write performance and read performance. The intent of laying out data using a log-structured file system is to improve write performance compared to a read-optimized file system. The layout of write data, however, can be optimized for a disk array. In particular, the block size, the segment size, and the position of segments on disk should be selected to match the configuration of the disk array.

The second issue is how the controller architecture influences the layout process. The Sawmill file system uses a controller with a fast data path and with controller memory in the data path. Because the path of write data to disk is different, new layout algorithms can let writes make best use of the storage system architectures. In particular, Sawmill uses a new technique called "on-the-fly" layout to reduce the cost of laying out data.

The final issue is that a log-structured file system requires a cleaning process to free up unused disk space. There are several ways the cleaner can be made more efficient for a RAID. For example, the cleaner can reorganize data on disk in order to make later reads more efficient. The cleaner can also use new parity techniques to reduce the cost of writing out partial segments to disk.

This chapter is organized as follows. Section 5.1 describes segment layout in a log-structured file system, and explains the trade-offs in the disk arrangement of data. Section 5.2 explains how the controller architecture affects data movement for writes. This section covers the motivation and implementation of on-the-fly layout. Section 5.3 introduces cleaning, which is required in a log-structured file system to reclaim free disk space. Section 5.4 concludes the chapter.

5.1. Physical disk layout

Sawmill uses a log-structured file system to improve performance of small writes. As described in Section 2.1, a RAID disk array performs poorly on small writes due to parity computation. While large writes can compute parity and write data in a single set of disk operations, small writes require a two-phase read-modify-write cycle, substantially increasing the cost. This section provides a detailed look at how a log-structured file system arranges data on disk in the layout process, and how this layout can be modified for efficiency with a disk array.

To summarize a log-structured file system, which was introduced in Section 2.6, data is written to disk in a large sequential log. This log contains all the file system information; metadata such as inodes is also written sequentially to the log. Since data in the log may become "dead" or unneeded, a garbage collection mechanism is needed to "clean" the log by reading in "dirty" segments and rewriting the live data to create empty, "clean" segments. The log is broken into large segments, on the order of a megabyte, in order to simplify cleaning. Each segment contains summary information describing the contents of the segment for use in cleaning.

The are several important issues in the layout. Layout must be performed efficiently for good write performance. Layout should also arrange data in the segment for efficiency in reading. Each segment must have enough information on layout for the segment cleaner to rapidly determine the organization of the segment and clean it. Finally, the layout must provide reliability and a way to recover after a system crash.

The organization of a log-structured file system segment in Sprite-LFS is shown in Figure 5-1; the Sawmill layout is similar. A segment may consist of several *partial segments*. If data must be written to disk before an entire segment of the log has been collected, the portion written is structured as a partial segment. Each full or partial segment consists of two components: the data and the segment information. The data portion contains file blocks, descriptors (inodes), directory log blocks, segment usage information entries, and descriptor map entries (specifying the position of inodes). At the end of each segment is a segment summary, which contains information summarizing the contents of the segment. This summary is used during segment cleaning to determine the organization of the segment. Each type of data (file, segment, descriptor) stored in the segment has a segment summary block describing the type and size of the data.

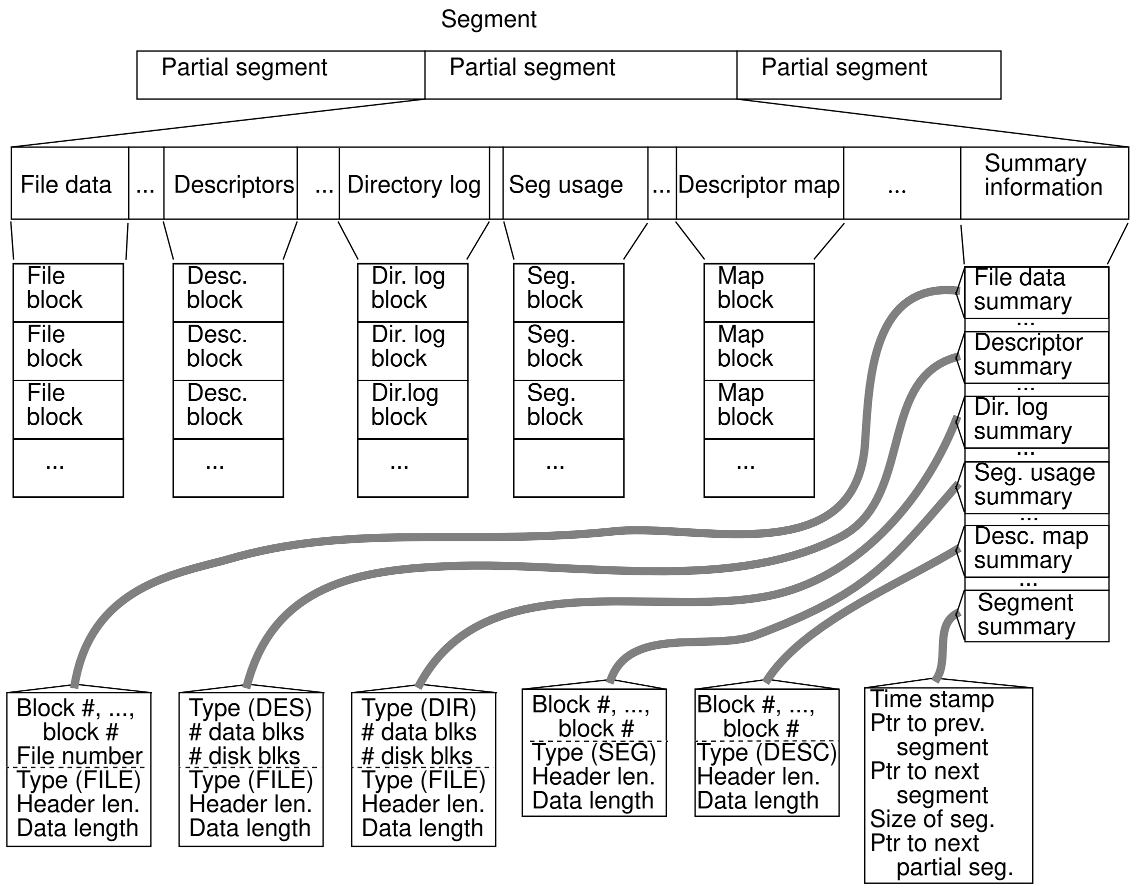


Figure 5-1. Arrangement of a segment

This figure shows the composition of a log-structured file system segment in Sprite-LFS and Sawmill-LFS. A segment of the log may consist of partial segments; these occur if the log is forced to disk before enough data is collected to fill a segment. Each partial segment contains various types of disk data and ends with summary blocks to describe each region and a summary block describing the segment. These summaries are used during cleaning to determine what data is stored in the segment. The types of data stored in the segment are regular file data, descriptors (inodes), directory updates, information on segment usage, and information on the location of descriptors.

There are alternatives for the organization of the data in the segment. For example, BSD-LFS simplifies the data layout by reducing the number of different types of data [SBMS93]. The segment usage map and the descriptor map are replaced by a special file (the IFILE) containing this information, and the directory log is replaced with an atomicity mechanism for directory operation writes. These changes considerably simplify the file system by reducing the number of different structures that the file system must handle.

Sawmill keeps most of the segment organization of Sprite-LFS. This allowed much of the layout code from Sprite-LFS to be used in Sawmill. Also, by preserving most of the aspects of Sprite disk layout, many of the file system utilities could be used without major modifications. If the log-structured file system were being redesigned from scratch, many changes could be made from the disk layout used in Sprite. In particular, the disk layout in Sprite-LFS is excessively complex, with multiple types of data. Using the on-disk structures of BSD-LFS would substantially reduce the complexity of the layout and cleaning code.

5.1.1. Block size

One of the important trade-offs in a file system is the file system block size. This block size is the unit in which data is allocated on disk, indexed in the inode, transferred to and from the disks, and stored in a cache. Although some file systems treat files as arbitrary extents, most file systems break files into fixed-size blocks, largely for ease of disk organization and caching. In a Unix file system, these blocks are typically 4 KB or 8 KB long.

Several parts of the system, such as the disk system, the file system, and the application's data access patterns, interact to determine the best block size. In many cases, however, the block size is constrained by backwards compatibility.

One factor in selecting the block size is the disk track and sector sizes. The block size should be a multiple of the sector size, since disks normally transfer data in entire sectors. In addition, if the block size divides evenly into the track size, blocks won't straddle track boundaries; changing tracks while reading a block could reduce performance by causing a missed rotation. With a RAID, the block size should divide into the stripe unit size, so a block transfer won't be split across two disks.

Using large file system blocks improves performance in several ways. Because the file system must perform processing on each block, increasing the amount of data per block reduces the processing overhead. In addition, for each block, the file system must store the mapping between the file address and disk address mapping in the inode or indirect block. Thus, increasing the block size reduces the amount of this mapping metadata required. This minimizes the amount of disk overhead to store the metadata, reduces the number of disk accesses to fetch indirect blocks, and increases the amount of data that can be accessed without requiring indirect blocks.

A larger block size, however, increases the cost for small updates to a file. For small writes, the entire block must be rewritten. If less than the entire block is being modified, then the old block must be read, the data modified, and then the new block written out. For small reads, the file system normally reads the entire block, so a large block size will increase the cost for small reads too.

Large blocks also increase disk storage complexity for a traditional file system (but not for a log-structured file system) since a mechanism is required to store small files efficiently. Since many files are very small, a large block size would result in much wasted space if a small file takes an entire block. Large blocks also waste space at the end of files. As well, large files decrease disk allocation flexibility, since there aren't as many blocks, reducing the chances of allocating blocks together. For these reasons, the BSD-FFS breaks large blocks into smaller fragments for small file allocation and for the ends of files. With larger blocks, however, the number of fragments will increase, making disk space management less efficient. With a log-structured file system, in contrast, large blocks do not cause these allocation problems; since blocks are written sequentially to the log, unused space can be removed as the blocks go into the log.

For these reasons, very large blocks are not a general solution. Instead, a trade-off must be selected between the benefits of large blocks and the additional cost for small operations. In many systems, a block size of 4 KB is used for historical reasons (many programs make transfers in 4K units), rather than because it optimizes performance.

The block size in Sawmill was increased from Sprite-LFS, but not dramatically. The Sawmill file system uses a block size of 16 KB. The original block size of 4 KB limited peak performance of the system; the CPU became a bottleneck because of the block processing costs. With 16 KB blocks, the per-block overhead is no longer a performance limit. Changing the block size does have drawbacks, though. As explained in Section 3.8.2, the change in block size requires the file system to perform translations between the new block size and the old block size, since some parts of the file system use the old block size.

5.1.2. Variable block sizes

One approach that could be used to minimize per-block overheads would be to let the file system block size vary from file to file. As explained in Section 5.1.1, a large block size minimizes the metadata processing cost and storage cost, but increases the cost of small writes and the fragmentation of small files. Large files that are accessed in large units could have a very large block size, while small files or files that are modified in small units could have a small block size. This allows the block size trade-off to be maximized on a file-by-file basis, rather than making a single size fit all files.

A variable block size is much easier to implement in a log-structured file system than in a traditional file system. The main reason is that when a traditional file system assigns disk blocks to a file, multiple block sizes will make it much more difficult to ensure that a block will fit in the available space. Normally, Unix-based file systems use a single block size to allocate disk space, with some blocks broken into smaller fragments (as explained in Section 2.4.1). If there are multiple, variable block sizes, the free space allocation will become more complicated.

In a log-structured file system, in contrast, multiple block sizes have little effect on disk allocation. Since blocks are written sequentially into the log, the size of these blocks is not especially important. Very large blocks might increase fragmentation where a block will not fit into the remaining space in the log, but this is unlikely to be an important factor as long as the maximum block size is significantly smaller than the segment size.

To implement a variable block size would require few major changes to a log-structured file system. Wherever the block size is used, the constant block size would have to be replaced by the per-file block size. The most important change to a general file system would be the interface between the block cache and the log-structured file system code. Because a block cache uses a fixed block size, the variable-sized LFS blocks would need to be broken apart going into the cache and joined together when leaving the cache. (This is analogous to the block size problem in the split cache, which is discussed in Section 3.8.2.) Thus in Sawmill, since transfers don't normally go through a block cache, variable-sized blocks would be simpler than in a cache-based file system.

Variable block sizes are not currently implemented in Sawmill, but would be an interesting experiment for future work. The main issues to explore are how performance is affected by the block size, and how to determine the best block size for each file.

5.1.3. Interactions of segment size with RAID configuration

The characteristics of a disk array critically influence the choice of segment size for a log-structured file system. With an LFS running on a single disk, there are no fundamental constraints on the size of the log segment. When the LFS is writing to a RAID, however, it is very important that a full parity stripe be written in each operation so parity computation can be done efficiently, rather than with a read-modify-write. Thus, the log segment size must be a multiple of the parity stripe size. This is illustrated by Sawmill; at one point, write performance was cut in half because the segment being written was slightly less than the parity strip size due to a bug.

Note that the best segment size for writes is not the best for reads. With N+1 disks, reading N+1 stripe units at a time would provide peak performance since all disks will operate in parallel. Writing N stripe units at a time plus parity, however, gives the best write performance since parity overhead is reduced. Thus, a LFS segment of N stripe units will optimize write performance, but one disk will remain idle when reading full segments. This read penalty, however, is much smaller than the performance penalty of writing a wrong-sized segment.

Large log segments will perform better than small segments because more data will be transferred per long disk seek between segments. The IBM 0661 disks [IBM91] used in RAID-II, for example, have an average seek time of 12.5 ms, worst-case rotational latency of 13.9 ms, and can transfer 2 MB/s. To seek and read a small 128 KB segment spread across 16 disks would take about 30 ms, while a 2 MB segment would take about 89 ms. (This assumes that the disk arms are roughly synchronized due to the striping, so they all require average seek time, but the spindles are not synchronized, so some disk requires worst-case rotational latency.) The respective average transfer bandwidths would be 4.3 MB/s and 23 MB/s. Thus, increasing the segment size yields over a factor of 4 improvement in potential performance. This illustrates that a rather large segment size is required to reduce the effect of seek time.

Large log segments, however, have disadvantages as well. First, since segments must be buffered in memory before going to disk, and must be read in for cleaning, large segments increase the memory usage correspondingly. This may especially be a concern since controller memory may be limited in size. Second, because file systems often force data to disk for reliability (as with the Unix `fsync`), large segments may rarely get filled to capacity, negating the potential benefit.

Large log segments also decrease the flexibility of selecting which clean segment to fill with data, because there are fewer segments to choose from. Although existing log-structured file systems simply select empty segments from a free list, performance could be improved by using more intelligent segment layout that could reduce seek time. As shown above, seek time is a significant component of the total time to access a segment. By allocating segments according to some locality policies, performance could be improved. One such policy would be to group related segments together. That is, when a segment is being written, it should be allocated as close as possible to other segments containing parts of the same file or parts of the same directory. These segments could also be allocated near the center of the disk, to minimize the average seek distance to them. In addition, cleaned segments, if they contain old data that is not being actively used, could be allocated near the edges of the disk, where seek times are maximized. Although optimization of segment allocation hasn't been implemented in Sawmill, it would be straightforward. When a segment is required, instead of selecting the first segment off the free list, all the segments would be scanned to determine which would work best for the data being written. If scanning all segments is too inefficient, a faster solution would be to break the disk into a small number of bins, store free segments in a list associated with each bin, and select the best bin. One downside of segment allocation is that it would perform best with a large number of segments to choose from; as the disk fills up, there will be fewer free segments and it will become difficult to choose a good one.

Another problem with large log segments is that the flexibility of cleaning is reduced, increasing the cleaning cost. With a smaller segment, it is more likely that a segment with most or all of the data dead can be found; this minimizes cleaning cost by increasing the amount of free space obtained per segment cleaned. Smaller segments are better for three reasons. First, there will usually be locality in data written to a segment; these blocks are likely to be related and have a good change of being overwritten or removed at the same time, resulting in a large free unit in the segment. With a smaller segment, it is more likely for these blocks to comprise a large part of the segment.

The second factor is statistical; assuming a random block distribution, a smaller segment has better odds of having more free space. This occurs because smaller segments are likely to have a wider range of free space available. To model this, consider a block to have probability p of being dead, and suppose a segment contains n blocks. This model is only an approximation, as the distribution of blocks is not random and it doesn't reflect the major changes in the distribution as segments are cleaned [Ros92], but it provides a rough illustration of the effects of increasing the segment size. Under this model, the distribution of free blocks in a segment will have a binomial distribution with mean np and variance np(1-p). This can be approximated by the normal distribution N(np, np(1-p)). Dividing by the total number of blocks yields the free fraction of the segment, which will have the distribution N(p,p(1-p)/n).

For example, consider a disk 50% used, with segments of 256 KB or 2 MB, and blocks of 4 KB. Figure 5-2 shows the distribution of free space under this model. With these segment sizes, p=0.5 and n=64 or 512, so the fraction of free space in a segment will be approximated by the distributions N(.5,0.004) and N(.5,0.0005) respectively. Thus, the probability of a particular segment being, say, 65% free is about 0.8% for the former segment size and about 10^{-11} for the latter. This illustrates that the chance of finding a segment with a large amount of free space can be much better with small segments.

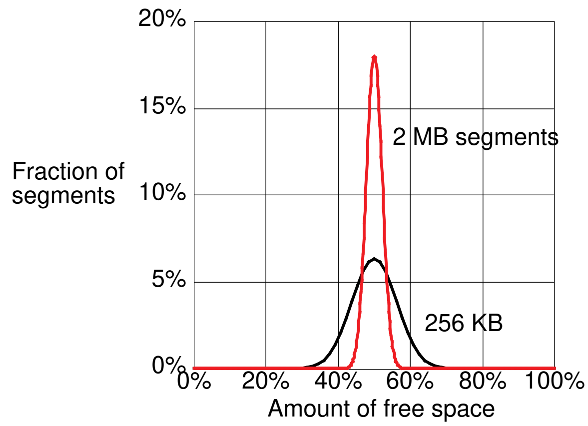


Figure 5-2. Free space vs. segment size

This figure illustrates the distribution of free space in segments under the simple model described in the text. Because smaller segments have fewer blocks, they have a wider distribution of free space. Thus, the odds of finding a segment with a large amount of free space are higher with smaller segments.

Finally, a small segment size increases the number of segments on the disk, so there will be more segments to choose from. This improves the chance of finding a segment with a large amount of free space.

In conclusion, many factors affect the choice of segment size. Sawmill currently uses a 960 KB segment, which is a single parity stripe. A larger segment size would probably improve performance when long seeks between segments are required. Future work can explore the trade-offs of segment size on performance.

5.1.4. Fast partial writes

The second aspect of log layout in Sawmill is how partial log writes are handled. In some instances, such as `fsyncs`, it is necessary to write the log to disk without waiting for a full segment. Normally this is relatively expensive, because writing a partial segment has high overhead from parity computation. However, because the log is sequential and these updates only occur at the end of the log, the full overhead of partial segment parity update can be avoided.

This section describes two methods that allow the parity computation to take advantage of the log structure to speed up partial segment writes. One technique, the *known data write*, caches the segment in memory, so parity can be computed without reading the unmodified data from disk. This replaces the read-modify-write with just a write. A second technique, *partial parity*, computes parity only over the written part of the log and not the full segment. In the event of a crash, the information in the unwritten part of the log is unimportant, so it does not need to be protected by parity.

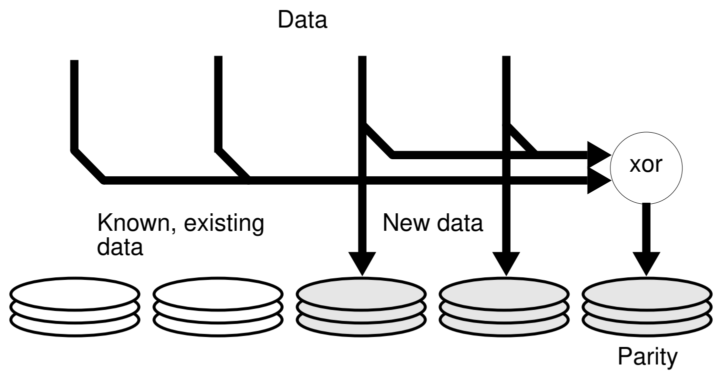


Figure 5-3. Known data write.

If the file system is writing a partial stripe, but knows the data on the unmodified part of the stripe, it can avoid the overhead of reading the old data. This allows the write operation to take place efficiently, even though it is a partial stripe write. The cost of writing m stripe units is m+1. This write could be used, for instance, to append data to a partially-written log segment.

5.1.4.1. Known data write

Figure 5-3 illustrates the "known data write". In this write, the file system writes a partial stripe to disk, but it knows what data is in the previously-written parts of the segment. In other words, the stripe is in memory and part of it is being updated. The partial write can be performed efficiently in this case because parity computation doesn't require reading the old data off the disk. Thus, the write cost is merely the cost of the data written plus the overhead of the parity stripe unit. This write is analogous to a reconstruct write, except the unmodified data is obtained from memory instead of disk. A variant of the known data write is shown in Figure 5-3. This variant is related to the read-modify-write technique of writing to a RAID, except the old data is obtained from memory instead of the disk.

The main disadvantage of the known data write is that it requires the operating system to know what is on the disk. One way this could occur is after cleaning, if the file system keeps the recently cleaned segments in memory before being overwritten. This would, however, take up controller memory. Alternatively, the file system could zero out cleaned

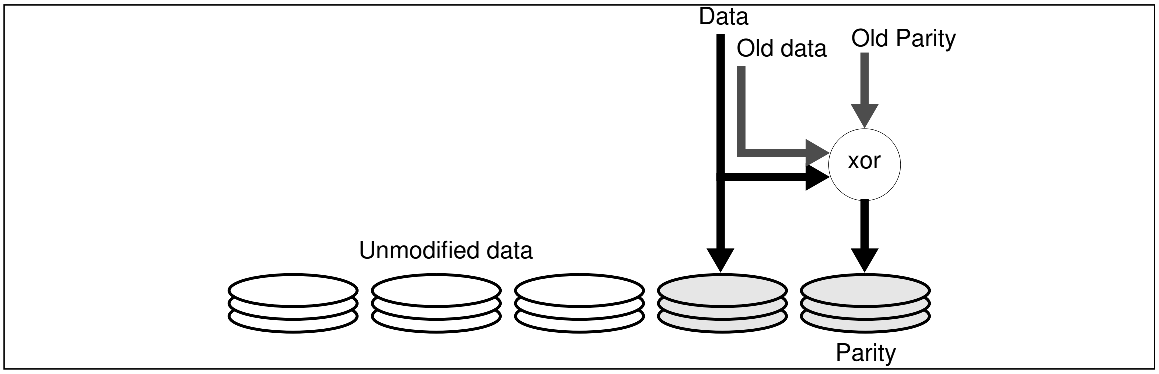


Figure 5-4. Known data write variant

A second method of performing a known data write is to use the old data and parity for the exclusive or computation. This reduces the amount of data exclusive or'd, but requires the old data and parity to be stored in memory.

segments on disk, so it would know what is stored there. This would require an additional write operation as part of cleaning, increasing the cleaning cost. If cleaning takes place at times of low load, though, then this additional write would not be significant.

5.1.4.2. Partial parity computation

An alternative to the known data write is to only compute parity over part of the segment. That is, some of the stripe units in the parity stripe will be treated as logically zero for the parity computation. If a crash occurs, the data in these stripe units will be unrecoverable. This is not a problem though, if these stripe units are in the unwritten part of the log, since they contain no valid data. By performing a partial parity computation, the performance penalty of a partial segment write is avoided. The write can be done in a single step, as for the full segment write. The partial parity write is illustrated in Figure 5-3.

One disadvantage of the partial segment write is that the recovery code must know which stripe units are included in the parity computation in order to recover data after a disk crash. This information cannot be simply written into a stripe unit, since it could be lost if that disk fails. Also, writing this information to all the disks in the stripe would negate the performance advantage of the partial write.

One solution would be to use a time stamp to determine which disk information is current. For example, a bit mask specifying the disks included in the parity computation and a timestamp could be written to the end of the updated stripe units and the parity stripe unit. Then, after a disk crash, the stripe units could be scanned to determine the last mask written, and then recovery can use those disks to re-create the lost data. One disadvantage of this approach is that it requires several bytes of data to be appended to all stripe units, including the parity stripe unit. This takes up disk space and may complicate stripe unit

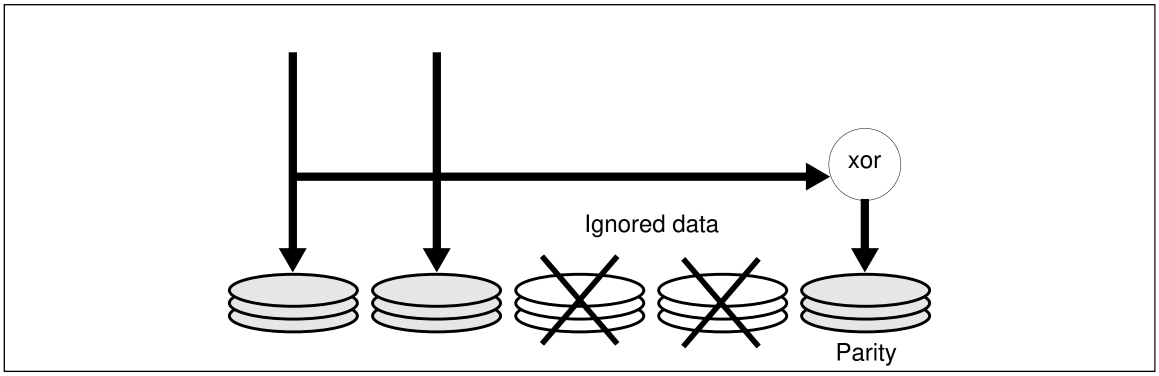


Figure 5-5. Partial parity write

If the data in some stripe units is not important, then parity does not need to be computed across those stripe units. This partial parity computation avoids the overhead of accessing the unneeded disks. In the above figure, parity is computed only over the first two stripe units, and the last two are unprotected.

---

allocation. More importantly, the parity stripe unit cannot be computed simply by an exclusive-or of the data stripe units, since this timestamp must be appended.

An alternative solution is to store the end-of-log information in some sort of stable memory, such as battery backed up RAM. Then, the stable memory could be examined after a crash to determine which stripe units are part of the parity stripe in the last segment. This is a simple solution, but it requires that the hardware have stable memory.

Finally, the stripe units in the parity computation could be determined by using a checksum. That is, whenever a segment is written to disk, a checksum could be computed across the stripe units of the segment and stored with the segment summary. After a disk crash when partial parity computation is used, the file system can scan the log segments for the end of the log. If it finds a segment with a bad checksum, it can then try the partial segment combinations to determine if any of them are valid. That is, it would examine stripe unit 0, then stripe unit 0 and stripe unit 1, then stripe units 0, 1, and 2, and so forth. Since there are only as many combinations as disks, this process will not be excessively expensive. The failed disk stripe unit in the segment being examined will need to be reconstructed for each of these trials, but as this is just a single stripe unit, reconstruction will also be inexpensive.

A similar checksum mechanism was used in BSD-LFS to determine when the log is valid in case the system crashed part way through a disk write, for normal disk writes, not partial parity writes. The problem is that a crash could result in a segment containing a mixture of old and new data, damaging file system integrity. This bad segment must be detected and discarded during crash recovery. In Sprite-LFS, faulty segment writes are determined by writing the segment with the summary last; writing the summary commits the change. This assumes that the segment is written sequentially so if the summary reaches the disk then the entire write has completed successfully. As pointed out in [Sel93], this assumption is not valid if disks reorganize operations. It is especially invalid for a RAID, when the operations to multiple disks take place independently. For this reason, BSD-LFS computes a simple checksum over the written segment. If the checksum doesn't match the data, then the file system can tell that the segment was not written correctly.

Note that a checksum algorithm must be used that won't be falsified by parity computation. That is, if a disk is reconstructed using parity over a partial segment, the checksum must not falsely claim the segment is valid. To illustrate the problem, consider a partial segment of three stripe units S_0, S_1, and S_2, the parity computed across them is $P=S_0 \oplus S_1 \oplus S_2$, and the checksum of these three stripe units will be $C(S_0,S_1,S_2)$. Suppose the disk containing S_1 is lost. It can be correctly recovered from the parity P and S_0 and S_2. It can be incorrectly recovered from, for example, S_0 and P, under the assumption of a short segment of two stripe units. This incorrect reconstruction would yield checksum $C(S_0, S_0 \oplus P) = C(S_0,S_1 \oplus S_2)$. If $C(S_0,S_1,S_2)=C(S_0,S_1 \oplus S_2)$, then the checksum cannot distinguish the valid recovery. Thus, a checksum must be used that doesn't satisfy this sort of exclusive-or property. An algorithm that exclusive-ors the stripe units together would have this problem, for example. Most standard checksum algorithms, for example summing the bytes, won't have this problem. There is the additional problem with any checksum that the checksum could erroneously match just by chance. With even a short 4-byte checksum, however, the odds of this are vanishingly small (2^{-32} or about 10^{-10}).

5.2. Efficient data movement

The data movement in Sawmill differs from previous log-structured file systems in two ways: the "bypass" I/O controller provides a new path for data, and the high bandwidth of the disk array requires correspondingly low file system overheads. These two factors affect write performance and led to the design of a new technique, *on-the-fly layout*, for efficiently writing the log to disk.

This section discusses the effects of this data path and of high bandwidth on writes. The first topic is how previous log-structured file systems collect write data and lay it out in the log. The second topic is how on-the-fly layout minimizes data movement overhead for writes, and how on-the-fly layout was implemented. Finally, this section describes how the split cache in Sawmill handles writes.

5.2.1. Log layout

A log-structured file system must arrange new file data into the sequential log that is written to disk; this process is called log layout. Write data is collected until a complete segment has been built up, and the segment is then written to disk. Section 5.1 discussed the physical layout of data on disk; this section covers how data gets arranged into this layout before going to disk.

In a standard LFS implementation such as Sprite-LFS [Ros92] or BSD-LFS [SBMS93], write data is stored in the file system block cache. A backend process pulls blocks out of the cache and places the blocks in the log. In more detail, layout is performed in several steps, as illustrated in Figure 5-6. First, when the cache has enough dirty data, a cache backend cleaner process is started. This process loops through the cache's list of dirty files. For each dirty file, it collects the dirty cache blocks. The cleaner process also collects directory modifications and updates to the segment usage map and descriptor (inode) map. The cleaner builds up a segment in memory through indirect references. That is, the segment in memory consists of pointers to the dirty cache blocks, other types of data, and the metadata describing the segment contents. Once the segment has been filled (or is forced to disk earlier by a `fsync`), the cleaner starts writing the segment to disk. The blocks of the segment are copied into a write buffer to create a contiguous buffer to be written to disk. Because the segment may be larger than the maximum disk transfer size, multiple write buffers may be used to write the segment in smaller parts. When the segment has been entirely written, the segment structures are freed from memory.

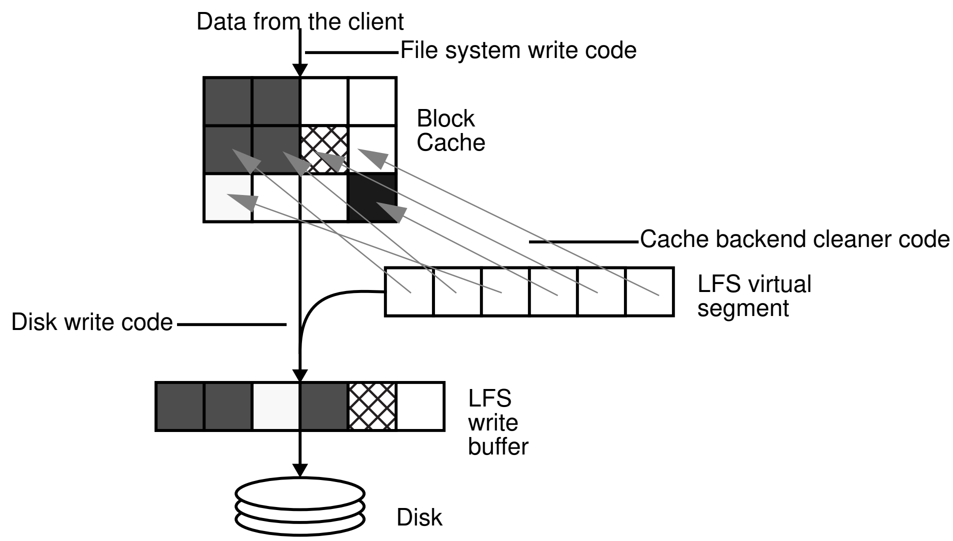


Figure 5-6. Log layout through a cache

In a traditional log-structured file system, the file system write code places blocks into the file system block cache. Next, a backend process collects dirty blocks out of the cache and lays them out in a segment buffer; this is done virtually with pointers. Finally the disk write code copies the blocks to the segment buffer, which is then written to disk.

There are several advantages to moving data through a cache for writes. First, data can be reorganized before it is written. For instance, if writes to a file are received in random order into a cache, they could be taken out of the cache sequentially and written in order.

This is beneficial for a log-structured file system, since it helps the performance of a sequential read following random writes by ensuring that the blocks are sequential on disk.

The second advantage occurs if data blocks are modified or deleted before being written to disk. With a cache, only the live blocks will be written to disk. That is, blocks can “die” in the cache if they are overwritten or deleted. Since many files exist a very short time before being deleted [BHK+91], delaying writes in the cache will improve performance.

On the other hand, writing through the cache is slow, since each block is processed three times. First, each block must be entered into the cache. Next, the backend cleaner must collect a reference to each block. Finally, the write code must copy each block into the write buffer and send it to disk.

5.2.2. On-the-fly layout

Sawmill uses a new method of performing log layout, called on-the-fly layout, to get more efficiency from writes. In contrast to existing log-structured file systems that use the cache backend to put blocks into the log, Sawmill assigns a position in the log to each data block when the write request comes in. That is, instead of moving blocks individually through a cache, the data for write stream is loaded directly from the network into the correct position in the log. This minimizes the processing cost to perform layout. Figure 5-7 illustrates the data path used by blocks with on-the-fly layout.

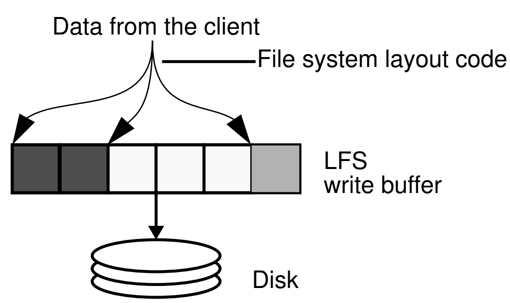


Figure 5-7. On-the-fly layout

Sawmill performs log layout “on-the-fly.” That is, blocks are assigned a position in the log as they arrive from the client, rather than being moved through a cache. This figure can be compared with the cache-based layout in Figure 5-6.

In more detail, on-the-fly layout has several steps. When a write request arrives, the file system immediately assigns a position in the log for the incoming data, even before the data has arrived over the network. The file system informs the network controller driver of the proper address in the log segment buffer in controller memory, causing incoming data to go directly from the network into the proper position in the log. At the same time, the file system reserves space in the log for any needed metadata. When a complete segment's data has arrived, the file system finishes the segment layout by copying the necessary metadata and segment summary information from kernel memory to the buffered segment. Finally, the segment is written to disk.

By eliminating the cache, the cost of layout is greatly reduced. With layout through a cache, two processing operations are required for each block. First, the block must be entered into the cache. Later, the block must be removed from the cache. Both of these operations require looking up the block in a hash table, locking the block, and then deciding what to do with the block. While these costs are not important for the bandwidths of typical workstation file servers, the processing may cause a CPU bottleneck when high bandwidths are involved.

In some cases, as in Sprite-LFS, cache layout is even more expensive because blocks must be explicitly copied between the cache and a buffer. This copy operation significantly reduces the potential bandwidth. With the RAID-II storage system, the RAID striping driver requires the segment written to disk to be stored contiguously in memory. Thus, a copy operation would be required. Some systems, however, can use a device driver that accepts a scatter-gather array so layout could be done with pointers and the blocks would not have to be copied.

A second advantage of on-the-fly layout is that layout for a large write can be done as a single operation, instead of many per-block operations. Because a large write can be assigned a position in the log as a single operation, most of the per-block overhead can be eliminated. In contrast, in the cache approach, each block needs to be assigned a position individually. For large requests, performing the layout as a single operation results in a large reduction of processing cost. The remaining per-block cost of updating each block's disk address in the inode can be nearly eliminated by updating all the entries at once. Then, the overheads of accessing the inode and locking it need only be done once, rather than for each block. These overheads account for most of the update cost; the actual operation of changing the disk address field in the inode is very small.

Finally, on-the-fly layout allows large requests to be processed as large requests all through the file system, rather than being broken into small blocks. Thus, writes can be viewed as a stream of data flowing from the network into the log. This approach improves bandwidth, since large network transfers can be handled more efficiently than small ones; the latency of starting the transfer is spread across more data, reducing its impact on the bandwidth.

There are a few disadvantages to on-the-fly layout, since it doesn't use a cache. First, a cache allows blocks to be reorganized before being written to disk. If blocks are placed in the log immediately, on the other hand, they will be written in the order in which they were received. This may slightly decrease performance if the blocks are later read in a different order. For many workloads this is not significant since most reads and writes are sequential.

A second disadvantage of writing blocks directly to disk is that blocks can't die in the cache. By delaying writes, the cache can prevent dead blocks from going to disk. However, with on-the-fly layout, the dead blocks will have a position in the log and will go to disk. These blocks won't cause correctness problems, of course, since the log-structured file system keeps track of the most recent copy of any blocks. It will, however, cause unnecessary disk traffic.

These disadvantages are not a significant problem in practice. Because of the high data rates, blocks are likely to go to disk before they would be modified or deleted, even with a cache, due to the very short cache lifetime of the blocks. In other words, the cache would have to be very large to provide much benefit for writes. In addition, if there were a cache on the clients, the client cache could do the processing and provide the benefits of removing dead blocks and reorganizing live blocks.

On-the-fly layout had a large impact on performance of Sawmill; an earlier implementation used a block cache for writes but due to copying and per-block overheads, performance was limited to under two megabytes per second. By performing layout on the fly, this overhead was reduced enough to transfer 15 megabytes per second for large writes.

5.2.3. Data movement and on-the-fly layout

On-the-fly layout provides a new write path that doesn't use a cache. Implementing this path required modifying the server's network interface code to move blocks to the segment buffer directly. The cache writeback code had to be replaced with code to collect the segment and write it out when ready. One issue in the implementation is how the code that determines the layout interacts with the data movement code. A second issue is how the network and disk drivers handle on-the-fly layout. These issues are discussed in this section.

In Sprite-LFS, the segment layout is done by the cache backend. As explained in Section 5.2.1, the backend routine lays out a segment by adding pointers to the dirty blocks to the segment structure. It also collects dirty segment usage map and descriptor map entries to the segment. While doing this, it builds up the segment summary metadata. When the segment has been filled, the disk write code copies the data out of the cache into the write buffer according to the layout arrangement. Thus, layout is done on a block-by-block basis, with blocks organized sequentially into the segment. Although the layout code selects the blocks that go into the segment, the actual movement of the blocks into the segment is done later and at a lower level.

There are several possibilities for changing this control organization to perform on-the-fly layout, as shown in Figure 5-8. One alternative would be to have incoming data passed to the layout code, which then would copy it into the segment. A second would be for the layout code to create a scatter-gather array that points to the data. Finally, the layout code could provide layout information so the incoming data could go from the network directly into the right position in the buffer.

In the first alternative, the request handler receives the incoming data and then passes it to the LFS module, which decides how to lay it out. This is roughly analogous to the lay-

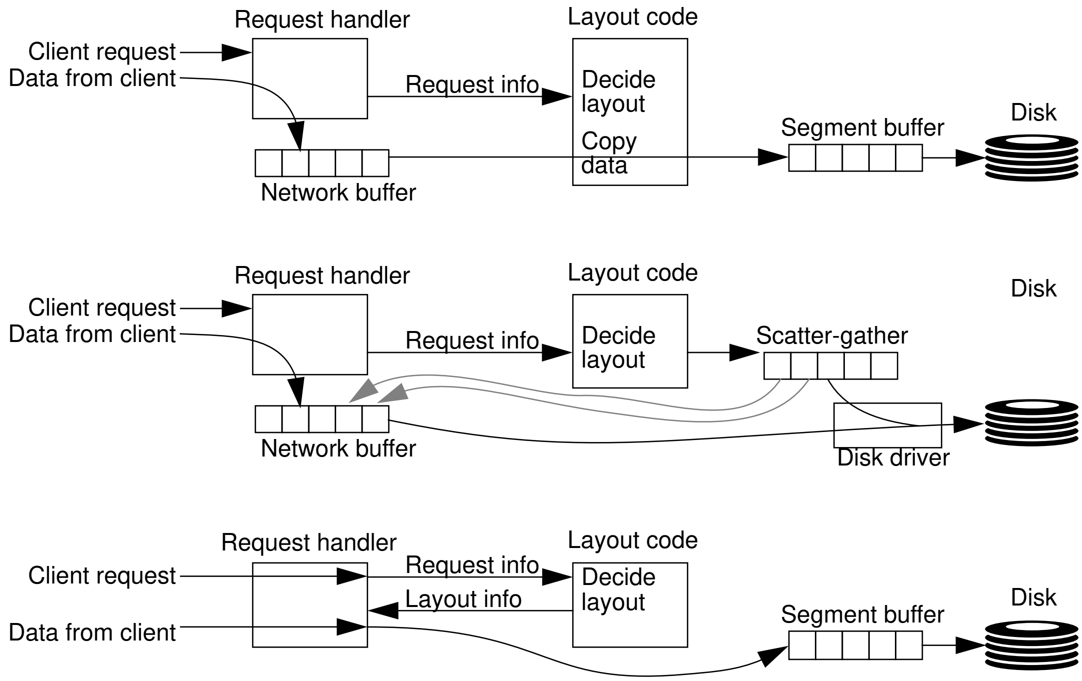


Figure 5-8. Layout and data movement

This figure shows three alternatives for integrating the log layout code with the data movement code. In the top alternative, the request handler tells the layout code what data has been received, and the layout code decides how to lay out the data and copies it into the segment buffer. In the middle alternative, the layout code does not copy the data into the segment buffer. Instead, it creates a scatter-gather array that points to the buffered data. The disk driver then accesses the data from the buffer without requiring a copy operation. In the bottom alternative, which Sawmill uses, the request handler tells the layout code information about the incoming request. The layout code decides how to lay out the data and tells the request handler, which directs the incoming data into the segment buffer.

out process in Sprite-LFS, which takes data blocks that are in the cache and lays them out. The main advantage of this approach is that it separates the tasks of the request handler (receive the data into a buffer) and the layout code (move the data into a segment). The main problem is that this approach is inefficient because it requires the layout code to perform a copy operation on the data.

A second, similar alternative is to arrange the data into the log logically rather than physically. That is, incoming write data will be stored in various input buffers; the log layout will consist of arranging pointers to the appropriate data. The disk driver can then use this collection of pointers as a scatter-gather array and can collect together the physically separate data at disk I/O time, without performing a copy operation. This layout technique is similar to that used by Sprite-LFS, which lays out the segment through a collection of pointers into the cache. Sprite-LFS doesn't avoid the copy operation, however; the disk write routine copies the blocks into a contiguous segment just before they go to disk. The problem with this approach is it requires a disk driver that supports scatter-gather, which RAID-II does not have.

The third approach, used by Sawmill, is to query the layout code which decides what to do with the data, and then to load the data from the network into the right place. This avoids the performance problem from copying. In more detail, the write process in Sawmill starts when a write request arrives. The request handler informs the LFS layout code of the size of the incoming request. The layout code decides where to place this data, starting a new segment buffer if required, and tells the request handler where the data should go. When the data has been received, the request handler notifies the layout code. If enough data has been received to fill a segment, the layout code starts the segment write process. It first copies any required metadata down from the server into the segment buffer. Then, it writes the segment to disk. When the write is finished, it frees the segment buffer.

These approaches interact with the network and disk drivers in different ways. The third approach requires cooperation from the network driver code, since the incoming data stream may need to be broken apart into multiple components and sent to different memory addresses. The network hardware must have enough buffering capability or flow control to allow the file system to process the request header before deciding where to direct the following data. In Sawmill this approach is used because the network provides flow control and the disk driver does not provide a scatter-gather ability. With a disk driver that allows a scatter-gather array, the second approach could be used. If neither driver can handle any data reorganization, then the file system will have to use the first approach and perform a copy operation to organize the data.

To conclude, Sawmill performs on-the-fly layout through interactions between the layout code, which decides where incoming data should go, and the request handler, which directs the incoming client data into the proper location in the log. This approach takes advantage of the network driver, which allows incoming data to be redirected after examining the request header.

5.2.4. Writes over the slow path

As discussed earlier, Sawmill provides a "slow path" through the file server to access Sawmill data. One complication with the slow path is how to integrate writes over the slow path with writes directly through the RAID-II controller. The problem with slow path writes is that the Sawmill file system lays out write blocks directly into a segment stored in controller memory. On the other hand, writes that go through the normal file system mechanism function quite differently: the blocks are buffered in the file system cache, and then a separate cache cleaner process writes them to disk.

There are several alternative approaches to handling this problem, depending on the level at which blocks from the two sources are merged together. These approaches are analogous to the slow path read alternatives discussed in Section 3.8.1.

One approach would be to keep the blocks separate all the way to disk; kernel "slow-path" blocks would be written to separate LFS segments. That is, each segment on disk would either contain fast path data or slow path data. A background process similar to the standard LFS cache writeback routine would fill a slow path segment with data from the split cache. One disadvantage of this approach is that if data is arriving very slowly over the slow path, then the slow path segment may not fill up very fast. This would result in writing nearly-empty partial segments, which are more expensive. This approach also makes segment management more complex, since there are two active segments instead of one, and the segment code must keep track of them.

A second approach would be to treat slow path writes the same as fast path writes, and copy them to controller memory as soon as they occur, before they go into the cache. The disadvantage of this approach is that as in the read case, the high-level write code is fairly closely linked with the cache block code. Modifying the high-level writes to go directly to Sawmill would require substantial modifications to how regular file system writes are handled.

A third approach would be to have the slow path cleaner lay out the blocks into the controller segment. That is, the slow path uses the same segment that is being filled by the bypass path's direct layout. The background cache cleaner gives the blocks to the Sawmill layout code, which copies the blocks into the current log segment buffer. This implementation minimizes the changes to the existing file system code, since the high-level code still uses the block cache. One disadvantage is that it complicates the Sawmill file system code, since the code must integrate writes from these two sources. This third approach is used in Sawmill. The first motivation for choosing this technique was to use the file system block cache; this minimizes the changes to the high-level file system code and cooperates with the "split cache" that is used for reads. In addition, this approach allows most of the existing Sawmill writeback code to be used.

5.3. Cleaning

As discussed in Section 2.6.2, the cost of cleaning old data from the log is a very important factor in a log-structured file system. This section describes the techniques that would be used to implement cleaning in Sawmill. (Cleaning was not implemented in the prototype.) First, because cleaning would be done using controller memory rather than kernel memory, a new cleaning path is required. Second, the cleaner itself can reside at user level or kernel level. Third, the cleaning process can be used to reorganize data to get more efficiency out of the disk array. Finally, cleaning can take advantage of the efficient parity techniques described in Section 5.1.4.

5.3.1. Controller memory

The controller buffer memory is an important factor in the cleaner with Sawmill. Unlike existing log-structured file systems that read dirty segments into kernel memory, a Sawmill cleaner would read segments into controller memory. Only the segment summary information necessary to clean the segment would be copied into kernel memory.

Thus, in the Sawmill cleaner, one or more dirty segments would be read off disk into controller memory. The metadata specifying the segment layout would be copied into kernel memory, where it would be examined by the cleaner. Any dirty blocks in the segments would be copied from the dirty segment into new segments. Then, the dirty segments would be discarded from memory and the segments on disk marked as clean.

One disadvantage of this approach (and of cleaning in other log-structured file systems) is that the entire segment is read off disk, even if most of it is stale, unneeded data. An optimization would be to read the segment summary information first, and then only read the live data off the disk. This would probably improve performance if the segment is mostly empty, but if the disks have to seek to read much of the segment there will be little performance penalty to read the entire segment.

Because Sawmill doesn't use a block cache, the cleaner would have to be modified to support the new write path. Sprite-LFS and BSD-LFS move live blocks from the segment being cleaned into the block cache, where they are written back out using the standard write path. As will be explained in Section 5.2.2, though, Sawmill writes data directly into the log rather than using a block cache to improve performance. Thus, the cleaner in Sawmill would need to be modified to use this new write path by copying dirty blocks into the log buffer rather than a cache and the cleaner would have to interact directly with the write code, rather than with the cache code.

5.3.2. User-level cleaning

The second issue is the structure of the cleaning code, which can either be in the kernel or at user-level. Sprite-LFS has the cleaner in the kernel with the rest of the log-structured file system code, while BSD-LFS has the cleaner at user level. Putting the cleaner at user-level allows flexibility in the choice of cleaning algorithms. However, it requires copying of segment data between kernel and user space.

A third approach would be to split the cleaner between the kernel and user space. In this approach, the user-level cleaning code selects the segment to be cleaned, allowing the cleaning policy to be modified at user level. However, the actual cleaning of the segment occurs in the kernel, for two reasons. First, it simplifies the code to have the cleaner in the kernel. There are several different log summary data structures that must be processed for cleaning or writing. If the cleaning code is in the kernel, this code can be shared between writing and cleaning, rather than duplicating code between kernel and user space. Second, cleaning in the kernel avoids copying summary data between kernel and user space; this overhead is especially important with Sawmill because of the high data rates.

The interface between the kernel portion of the cleaner and the user portion would be straightforward. The user process is woken up by the kernel when a segment should be cleaned. The user process then receives information on the segments (in particular their age and the amount of free data). Finally, the user process informs the kernel which segment should be cleaned.

One drawback of the split kernel-user cleaner is that it is not as flexible as an entirely user-level cleaner. For instance, one potential modification to the cleaner would be for the cleaner to coalesce blocks together to improve later read efficiency. With the BSD-LFS cleaner, this could be done without kernel changes, but with the cleaner split between kernel and user levels, it would require kernel modifications.

5.3.3. Cleaner reorganization

LFS is a write-optimized file system; by writing arriving data sequentially to a log, it optimizes the layout of data to improve write performance. On the other hand, read-optimized file systems, such as the BSD fast file system discussed in Section 2.4.1 organize data with a layout that helps reads by minimizing seek time. For example, they may group blocks from a single file together, put the contents of a directory together with the directory, and place the most accessed files closer to the center of the disk.

One method for improving read performance in a log-structured file system would be to reorganize the written data at a later time, to provide a better layout for reads. In a log-structured file system, a natural time to do this reorganization is during log cleaning, when old data blocks are read in and written back. Reorganization can be divided into three types: rearrangement of the data inside a segment, packing new data into the segment, and repositioning the segments on the disk. This section discusses how the log cleaner can perform these types of reorganization.

The simplest type of reorganization is to rearrange the data inside a segment. For instance, the cleaner could sort the blocks in a segment into a good order before the cleaned segment is written back out. In Sprite-LFS and BSD-LFS, this sorting happens at write time; the writeback code writes on a file-by-file basis, with the blocks in a file written in order. Sawmill, however, uses on-the-fly layout, which writes the blocks in the order received. But, by doing this sorting during writeback, Sawmill could achieve much of the benefit of block sorting.

A second way in which the cleaner could reorganize data during cleaning would be to collect blocks from multiple segments so the data written back out will have a better organization. For example, sequential blocks from a file could be collected together before the blocks are written back out to disk. This would significantly increase read performance if the blocks are later read sequentially. One way to group together multiple segments would be to clean several segments in parallel. The segments to clean could be selected based on their contents to improve the grouping. Alternatively, blocks could be "stolen" from active segments to form sequential runs, without necessarily cleaning those segments at that time. The additional seek time to read the stolen blocks, however, is unlikely to make it worthwhile.

The final type of reorganization is to arrange the segments efficiently on disk. This was discussed in Section 5.1.3. Data that is rarely used could be written during cleaning to segments near the edge of the disk, while commonly used data could be written to the middle of the disk. This would reduce the average seek distance. Also, segments with related data could be positioned close to each other.

5.4. Conclusions

A log-structured file system is used in Sawmill to improve the performance of small writes. With a RAID, small writes are expensive because of parity computation. By grouping these writes together into a log, a log-structured file system avoids this performance penalty for small writes.

Laying out the information into the segment is an important part of write performance. This chapter has described several techniques for making layout more efficient. Some of these techniques are applicable to any storage system that uses a RAID, while others are specific to controller architectures with a bypass data path.

The primary difference between writes in Sawmill and writes in previous log-structured file systems is the use of on-the-fly layout in Sawmill instead of cache-based write techniques. This was motivated by the elimination of caches for reads, and by the computational and data movement overheads of moving blocks through a cache. With on-the-fly layout, incoming data blocks are immediately assigned a place in the log.

Partial writes, where a log segment is forced to disk before the segment is full, are a potential factor for limiting performance. By using new parity computation techniques, the expense of partial writes can be reduced, making them comparable to efficient full-stripe writes.

The cleaning process is a significant overhead in a log-structured file system. The data path used in cleaning must be modified to take advantage of controller memory. Cleaning is also an opportunity to reorganize the data on disks to improve the performance of later reads.

In conclusion, the hardware architecture of a storage system such as RAID-II has several important effects on how to obtain high performance of writes; in particular, the disk array and the bypass data path motivate new write techniques.

6 Performance evaluation

"... but it takes away the performance." – Shakespeare

This section describes the performance of the Sawmill file system running on the RAID-II storage system. The goals of this chapter are to:

- Compare the Sawmill file system to the raw disk performance and determine how effective Sawmill is at delivering the potential bandwidth.
- See if a log-structured file system improves write bandwidth to a disk array.
- Determine the bottlenecks of Sawmill and how they will be affected by changes in computer technology.
- Test the performance of a standard file system on RAID-II.
- Examine the effectiveness of a file system block cache for a server with many clients.
- Evaluate the Sawmill disk block address cache.

This chapter is organized as follows. Section 6.1 gives measurements of the RAID-II disk array. Section 6.2 covers the performance of the Sawmill file system running on RAID-II. Section 6.3 discusses how the performance of Sawmill would scale with changes in technology. Section 6.4 compares Sawmill to a standard file system running on RAID-II. Section 6.5 contains trace-driven simulation results of a file system block cache with multiple clients. Section 6.6 evaluates the Sawmill disk address cache. Section 6.7 concludes the chapter.

6.1. Performance of the RAID-II disk array

This section describes the performance of the RAID-II disk array without a file system. These measurements give an upper bound on the potential capabilities of the file system and provide parameters for modelling the system. Section 6.1.1 examines the disk array treated as a collection of independent disks. Section 6.1.2 gives measurements of the disk array when treated as a RAID, that is, with data striped across the disks with parity computation. Other measurements of RAID-II may be found in [DSH+94] and [CLD+94].

The measurements in this chapter were made on the RAID-II disk array, which was described in Section 2.3. The disk array contains 24 IBM 0661 disks [IBM91], but most of the measurements use 16 of the 24 disks as will be explained below. Table 6-1 summarizes the performance characteristics of these disks. The disks are connected to four Interphase Cougar disk controllers. Each disk controller handles two SCSI strings and each string has 3 disks, as illustrated in Figure 6-1. The file server controlling RAID-II is a Sun-4/280. This workstation is relatively slow by current standards (about 9 SPEC89 integer SPECmarks).

Parameter	Value
Disk rotational speed	4316.06 RPM
Average rotational latency	6.95 ms
Average read seek time	12 ms
Average write seek time	14.5 ms
Maximum transfer rate	1.679 MB/s
Capacity	320 MB

Table 6-1. Disk parameters

This table summarizes the characteristics of the IBM 0661 disks used in RAID-II.

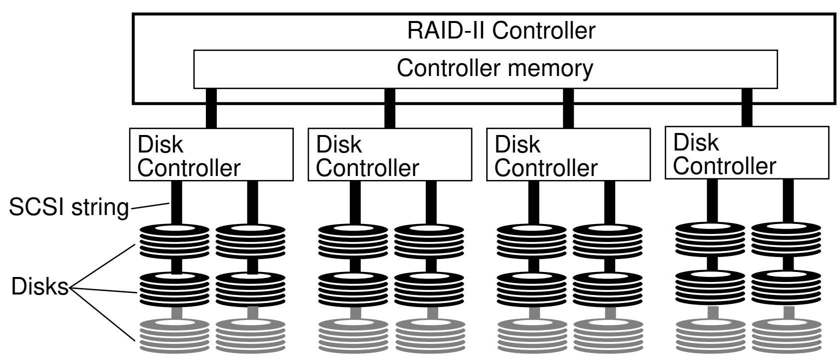


Figure 6-1. RAID-II configuration

This figure shows the configuration of RAID-II for raw disk tests. Most of the measurements used 16 disks configured as 2 per SCSI string, but some of the measurements in Figure 6-2 use three disks per string (indicated in gray).

6.1.1. Raw disk performance

This section provides low-level measurements of the RAID-II disk array. In this section, the disk array is treated as a collection of independent disks, without a file system and without RAID parity or striping.

The first set of measurements illustrate the peak bandwidth through the components of the RAID-II system. For these measurements, large sequential reads were sent to the disks to obtain their best performance. An individual disk read data at 1.6 MB/s. A single SCSI string read at 3.14 MB/s with two disks. With three disks active on a string, bandwidth dropped to 2.9 MB/s; this illustrates that the string reached its maximum bandwidth and became a bottleneck, confirming the string bottleneck found in [CLD+94]. One disk controller with two strings of two disks each provided 6.24 MB/s. With 16 disks on 8 strings and 4 controllers, the read bandwidth was about 24.5 MB/s. Thus, the SCSI string is the key limiting factor for peak read bandwidth from the array. Performance of the remaining components of the system scales nearly linearly as the number of disk drives increases, up to a peak of 24.5 MB/s with 16 disks.

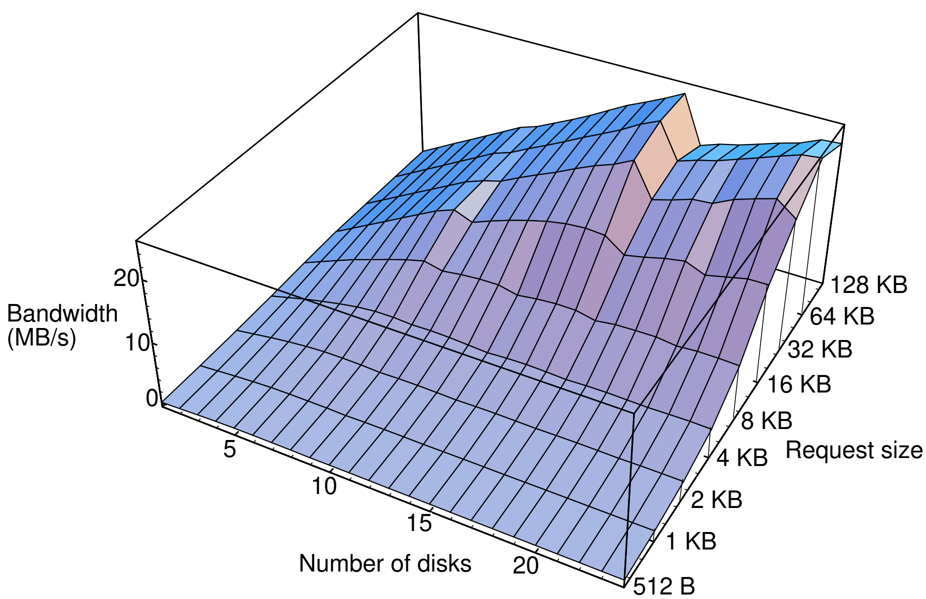


Figure 6-2. Disk array read bandwidth vs. number of disks

This figure shows how the bandwidth of the disk array varies with different numbers of disks and different request sizes. Note that bandwidth is much better for large requests. Peak bandwidth is obtained with 16 disks due to the SCSI string bottleneck.

Figure 6-2 shows how the read bandwidth of the disk array varies with the request size and the number of disks. Requests of the specified size and with random offset were sent to each of the active disks in parallel, so each disk had one outstanding request. The request size was varied from 512 bytes (1 sector) to 128 KB (the maximum SCSI transfer size). Read requests took place from the disks into controller memory. The number of disks was varied from 1 to 24. To reduce controller or string contention, disks were assigned to different strings and controllers when possible. For example, the four disk measurement used one disk on each controller, rather than four disks on one controller.

Figure 6-2 illustrates several characteristics of the disk array. First, bandwidth increased dramatically for large transfers. This is mainly because small request performance is limited by the positioning time of the disk. Second, note that peak bandwidth was obtained with 16 disks, due to the SCSI string bottleneck described above. Thus, the remainder of the measurements of the disk array in this chapter use 16 disks.

Figure 6-3 shows the performance of the raw disk array as a function of request size. For these measurements, requests with a random offset and the specified size were made to each of the 16 disks. As well as bandwidth, this figure shows the CPU load and disk utilization. The CPU load shows the usage of the file server CPU. The disk utilization indicates the percentage of time the disks were in use, averaged across all 16 disks (e.g. 8 disks in use 50% of the time would be 25% utilization).

Bandwidth was highly dependent on request size. Peak read bandwidth was about 24.5 MB/s and peak write bandwidth was 19.5 MB/s. For small requests, seek time was the main factor affecting performance. With the raw disk array, the processing overhead was minimal, so the performance was almost entirely dependent on the disk speed. Writes were slower because of the rotational positional cost. That is, the disk must rotate to the proper location before starting a write. On the other hand, the disks have a track buffer that minimizes missed rotational costs for reads; the disk can start reading data from any rotational position, buffer the track, and then rapidly provide the data in order from the buffer.

The CPU load is fairly high: load for 16 parallel raw disk operations is about 70% for small operations, drops to 20% for 128 KB requests, and then climbs slightly. This indicates that although the RAID-II architecture allows data rates greatly exceeding what the file server could move, our current file server CPU is just barely able to handle the high interrupt rates of small disk operations. The CPU load is determined by a fixed cost per disk operation: starting the disk operation, handling the interrupt, and concluding the disk operation take about 900 μs in total. Thus, small operations have a higher CPU load because the disk time is smaller compared to the fixed CPU time. Requests up to 128 KB result in one disk operation, but larger requests result in multiple disk operations since the SCSI controller has a maximum transfer size of 128KB.

The characteristics of the disk array shown in Figure 6-3 can be distilled down to the simple model shown in Table 6-2. In this model, disk bandwidth is determined by a latency factor plus a cost per byte. The CPU load is determined from a baseline load plus a processing overhead for each request. The model assumes that CPU load does not affect the bandwidth. This assumption holds for most requests but breaks down for very small

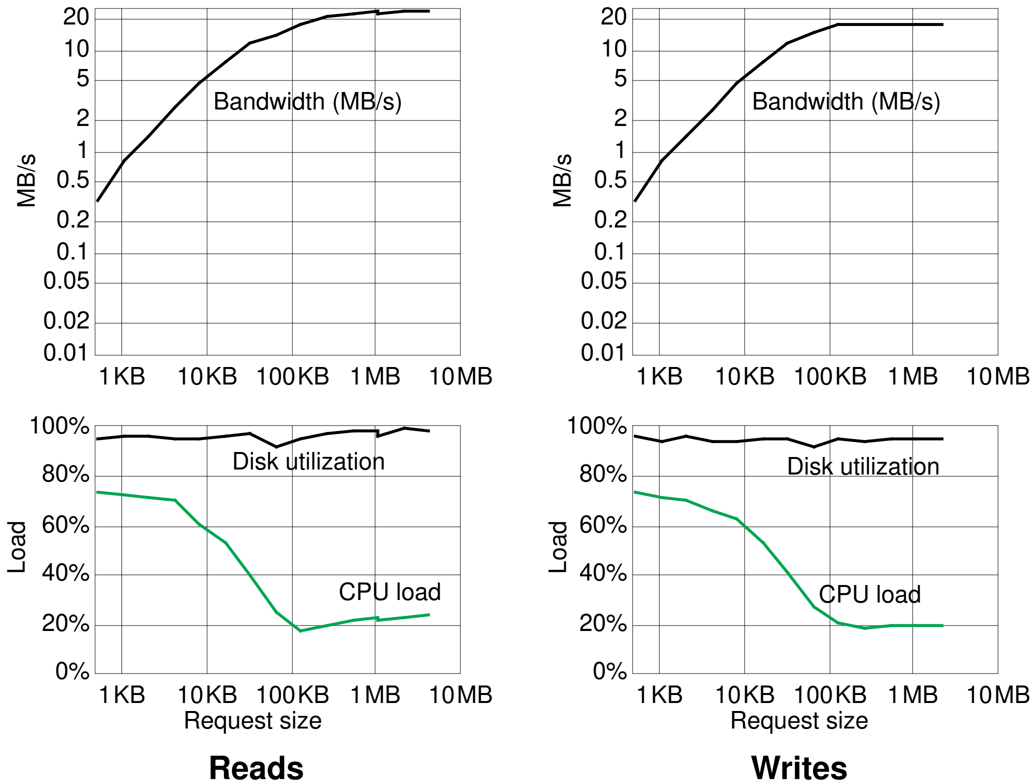


Figure 6-3. Raw disk array measurements

This figure shows the bandwidth available from the disk array when it is treated as 16 independent disks. Request size and bandwidth are plotted on logarithmic axes, while the load axis on the lower graphs is linear. The disk utilization line gives the average percentage of the 16 disks that are in use. The seek time is the main limiting factor for small requests. Note that CPU load drops with larger requests, since disk interrupts are less frequent. Peak read bandwidth is about 24.5 MB/s and peak write bandwidth is about 19.5 MB/s.

---

transfers, where the high load delays disk requests and slightly reduces bandwidth from the model. Otherwise, the model predicts bandwidth within about 15% and load within about 4%. The parameters in Table 6-2 were determined by least-squares fitting the model to the measured performance. For example, the CPU time was computed from the measured CPU load and the measured time to complete the request. Then a least-squares fit generated the idle load and per-operation CPU time parameters.

A final measurement that affects file system performance is the cost of memory to memory movement in the RAID-II controller. Data can be copied between file server kernel memory and RAID-II controller memory over the VME link between the file server and RAID-II. Measurements show that copying data over this link has a latency of about 700

time to complete request = *seek time* + *request length* ÷ *disk bandwidth*
number of disk operations = 16 × min(1, *request length* ÷ 128 KB)
CPU time = *operation CPU time* × *number of disk operations*
CPU load = *idle load* + 100% × *CPU time* ÷ *time to complete request*
bandwidth = 16 × *length* ÷ *time to complete request*

Parameter	Read value	Write value
seek time	20 ms	20 ms
disk bandwidth	1.6 MB/s	1.2 MB/s
operation CPU time	900 µs	900 µs
idle load	6%	6%

Table 6-2. Model of disk array performance

This simple model describes the bandwidth of the RAID-II disk array and the load on the file server CPU under various request sizes.

µs and a transfer rate of about 4 MB/s. Thus, copying a 4 KB block between kernel memory and controller memory takes about 1.7 ms. Copies between RAID-II memory and RAID-II memory take place at about 19 MB/s with about 500 µs latency. Thus, moving a 4 KB block in RAID-II memory takes about 700 µs. Most of the latency for copies is due to the overhead of starting a RAID-II operation, taking an interrupt when it completes, and then signalling the process that issued the operation.

6.1.2. Performance of the striped RAID disk array

In the previous section, the disk array was viewed as a collection of independent disks. In this section, the disk array is treated as a RAID, but without a file system. The RAID striping driver receives requests and breaks them up into stripe units spread across multiple disks. For writes, parity is computed. These measurements are significant because Sawmill is built on top of the RAID. Also, comparing these measurements with the previous section shows the cost of striping data and computing parity.

In this section, a 64 KB stripe unit is used. That is, data is striped across the disks in blocks of 64 KB. With 16 disks, this yields a parity stripe (data plus parity) of 16×64 KB = 1 MB, of which 15×64 KB = 960 KB is data. A 64 KB stripe unit was selected to provide a large enough stripe unit that seek time wouldn't dominate disk accesses, but small enough that the parity stripe size wouldn't be excessively large, since stripes must be buffered in controller memory.

The graphs in Figure 6-4 show the read and write performance of a single stream to the RAID device. That is, a single stream of random requests of the given size is passed to the RAID striping driver, which stripes the request across one or more disks. Bandwidth, CPU

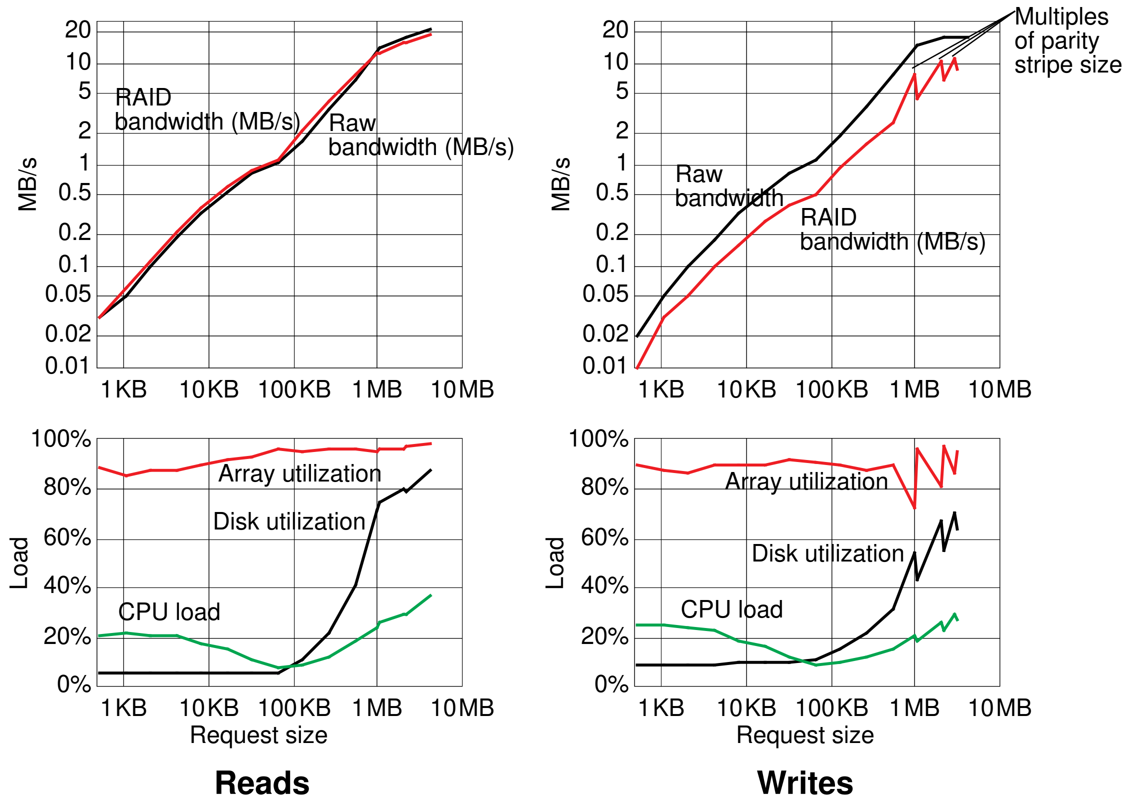


Figure 6-4. RAID and raw disk single stream performance

This figure shows the performance of the disk array when configured as a RAID with a 64 KB stripe unit. For comparison, it shows the performance of the raw disks under comparable requests. The disk array received a single stream of random reads or writes of a particular size. The array utilization line gives the percentage of time that the disk array is performing any operation. For reads of 64 KB or less, only a single disk is active, reducing the disk utilization and the bandwidth. For writes, RAID performance has peaks in bandwidth at multiples of 960 KB, when a full parity stripe is written. For writes of other sizes, a read-modify-write is required to update parity. This can be seen in the lower bandwidth and higher disk utilization.

load, and disk utilization are given as in the previous section. In addition, the graphs show the disk array utilization, which is the fraction of the time that any disks in the array are active. For example, if one disk were constantly active, disk utilization would be 1/16 = 6.25%, while array utilization would be 100%.

For comparison, these graphs also show the performance of the raw disks scaled by the stripe size. Scaling is necessary because a small RAID request only uses a single disk, while the earlier raw measurements used all the disks in parallel, resulting in an automatic factor of 16 better performance for the raw disk array. In the scaled raw measurements, the disk usage matches the striped RAID accesses. That is, requests up to 64 KB use a single disk, requests from 64 KB to 1 MB use 64 KB accesses on multiple disks, and requests above 1 MB use accesses to 16 disks that are 1/16 the size.

For a single request stream of small reads, RAID bandwidth is much lower than the total disk array bandwidth, but nearly identical to the scaled raw bandwidth. This illustrates that for reads there is little performance lost due to the RAID striping driver. Since requests of 64 KB or less are striped onto a single disk, small request bandwidth is a factor of 16 worse than the raw disk bandwidth in Figure 6-3 and disk utilization does not exceed 6.25%. As the request size increases beyond 64 KB and more disks are used, the RAID bandwidth climbs accordingly.

RAID write performance is significantly different from that of the raw disk array due to parity computation. These measurements show that small writes are very inefficient on the RAID. For efficient writes, the file system must perform large operations in multiples of the parity stripe size. Most RAID writes take about twice as long as writes to the raw disk due to the read-modify-write parity update. Note the performance peaks for a write that is a multiple of the parity stripe size, for which parity and data can be written at the same time: the spikes in the graph illustrate the better write speed at multiples of 960 KB. Even for parity stripe writes, RAID performance doesn't match raw performance; due to the processing overhead and parity computation of writes, the disks are only 50-70% utilized. For the efficient full stripe writes, the disk utilization climbs since writes go to all the disks at once; there isn't the read operation going only to the parity disk. Because the disk time to complete the operation drops, the CPU overhead becomes a larger fraction of the total. Thus, disk array utilization drops and CPU load increases for the efficient full stripe writes.

To transfer a given amount of data, the CPU load is significantly heavier with the RAID driver than with the raw disk, especially for small requests. Although for small transfers the raw disk CPU load in Figure 6-3 appears higher than the RAID CPU load in Figure 6-4, this is misleading since 16 disks are active in the raw disk case but only one disk is active in the RAID case. To examine the load in more detail, Table 6-3 provides a model of the CPU usage for requests to the RAID striping driver. This model was derived from additional measurements of system performance that used varying numbers of disks to separate the effect of request overhead from disk overhead. The model parameters were then obtained by least-squares fitting to the measurements. Note that in addition to the CPU overhead for each disk operation, there is additional overhead for each request to the striping driver due to the block computation the driver must perform. Also note that the per-disk overhead is larger for the RAID striping driver than for the raw disk because of the additional computation for each request. Finally, because of parity computation, the RAID requires more disk operations for a write than the raw disk does. Because of these factors, small RAID transfers have about 3 to 9 times the CPU overhead per byte of raw disk transfers. The additional load of the RAID driver is very important, since it limits the number of RAID transfers that can happen concurrently.

CPU time = disk op CPU × *number of disk ops* + *request CPU* × *number of requests*
CPU load = idle load + 100% × *CPU time* ÷ *time to complete request*

Parameter	Read value	Write value
disk op CPU	1.1 ms	2 ms
request CPU	2 ms	2 ms
idle load	2%	1-3%

Table 6-3. Model of RAID CPU usage

This model approximately describes the bandwidth of the RAID-II disk array and the load on the file server CPU under various request sizes. The CPU load consists of an overhead for each RAID request and an overhead for each disk operation that is part of the request.

Because a single stream will not keep a RAID disk array busy, especially for small requests, I also measured RAID performance with multiple streams. That is, several request streams were sent to the RAID driver at once, allowing more parallelism. Potentially, 16 small reads or 8 small RAID writes could take place in parallel, improving bandwidth by the same factor. Requests larger than the 64K stripe unit, however, are striped across multiple disks and become inherently parallel. Thus, multiple requests improve the performance much more for small requests than large requests.

Figure 6-5 shows the results of the multiple stream measurements. In these graphs, multiple requests were used to drive the system closer to saturation than in the single stream graphs. Parallelism was much less than the potential because the CPU became a bottleneck. The measurements used 8 request streams for requests up to 256 KB for reads and 64 KB for writes; the number of streams then decreased. For small requests, 8 streams were enough for the CPU to become a bottleneck. The number of streams was progressively decreased for larger requests because the disk started to become a bottleneck, led to reduced performance in some cases, and the size of controller memory limited the number of streams possible.

Compared to Figure 6-4, there is a large increase in RAID bandwidth due to the parallelism. The RAID bandwidth is significantly lower than the raw disk array bandwidth, however, because the raw disk performance benefitted much more than the RAID from multiple streams. RAID writes are slower both because of the lesser parallelism with RAID and because of the read-modify-write for most request sizes. For large reads and parity stripe writes, RAID performance is very close to the raw disk performance. Note that CPU load is much higher for RAID than for the raw disk accesses; this is the bottleneck for small requests. With a faster CPU, the number of streams could be increased, resulting in better disk utilization and higher bandwidth. Thus, multiple streams increase the RAID performance, but the CPU load limits the possible parallelism.

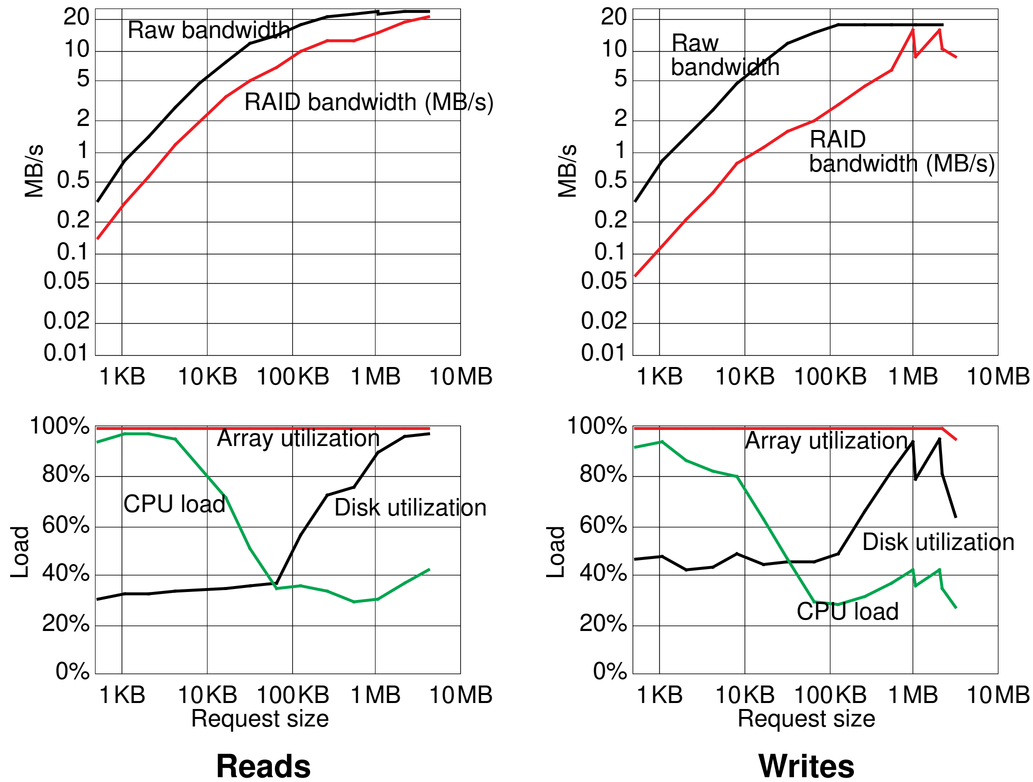


Figure 6-5. RAID and raw disk multiple stream performance

This figure shows the performance of the disk array treated as a RAID, with data striped in 64 KB units. For comparison, the performance of the raw disk array with all disks operating in parallel is also given. Note the disk utilization curve shifts above 64 KB, when requests are striped across multiple disks. In these graphs, multiple requests are used to drive the system closer to saturation than in the single stream graphs.

In conclusion, the RAID measurements illustrate several factors that are important to the file system performance:

- Large transfers are necessary to get the best performance.
- Multiple operations must take place in parallel in order to keep the disks in use, especially for small operations.
- Writes have a severe parity computation penalty unless they take place as full stripe writes.

- The processing load is much higher through the RAID driver than with the raw disks. Large transfers can use 40% of the CPU, while concurrent small transfers can use almost all of the cycles. The CPU load, rather than the number of disks, limits the number of small concurrent operations that RAID can perform.

6.2. Performance of Sawmill

This section provides measurements of the Sawmill file system. As in the previous section, these measurements used RAID-II with 16 disks and a stripe unit of 64 KB, yielding a parity stripe size of 960 KB. The LFS segment size was also 960 KB so each log segment could be written to the RAID efficiently. The measurements in this section used random requests of various fixed sizes.

There are several key results from these measurements. First, Sawmill is able to provide about 80% of the raw disk bandwidth for large requests. Sawmill is also able to handle a single stream of small read requests with bandwidth close to that of the raw disk. Because of its use of logging, Sawmill can perform a single stream of small writes an order of magnitude faster than the raw disk. However, the server CPU is a bottleneck for small writes and for multiple request streams. Small write bandwidth with Sawmill would improve dramatically with a faster CPU.

Because we don't have a client that can handle the data rates of RAID-II, the measurements in this section did not transmit data across the network, but only transferred the data between the disks and controller memory. The available clients could only transfer 3 to 4 MB/s over the network. However, the measured server CPU load to handle this network traffic was only about 2%. Thus, the server could transfer the full RAID-II bandwidth over the network without a CPU bottleneck arising due to the network traffic. Therefore, the measurements in this section should be a reasonable indication of the actual network performance with a fast client.

6.2.1. Single stream performance

Figure 6-6 shows the performance of a single stream of read and write requests to the Sawmill file system, indicating the bandwidth available to a single application. The left half of the figure gives measurements for reads. Note that the bandwidth of small reads was limited by the disk seek time, as was the case for the raw disk array and for the RAID. Peak read bandwidth was about 18.4 MB/s. Note that CPU load is fairly high, but reaches a minimum for requests around 100 KB. This shape is due to two factors: CPU overhead per byte and the total number of bytes. The CPU usage per disk operation is roughly constant. As the request size increases to the 64 KB stripe unit, each disk operation transfers more data, so the fraction of time spent with CPU overhead decreases. However, as request size continues to increase beyond 64 KB, more disks are used in parallel, causing the CPU usage to climb again. Compared to Figure 6-4, Sawmill CPU usage is about twice as high as for the RAID; this reflects the additional overhead of the file system.

Figure 6-6 also shows write performance. Large files can be written to disk at about 15 MB/s. Note that small writes are totally CPU limited. This is due to LFS: since small

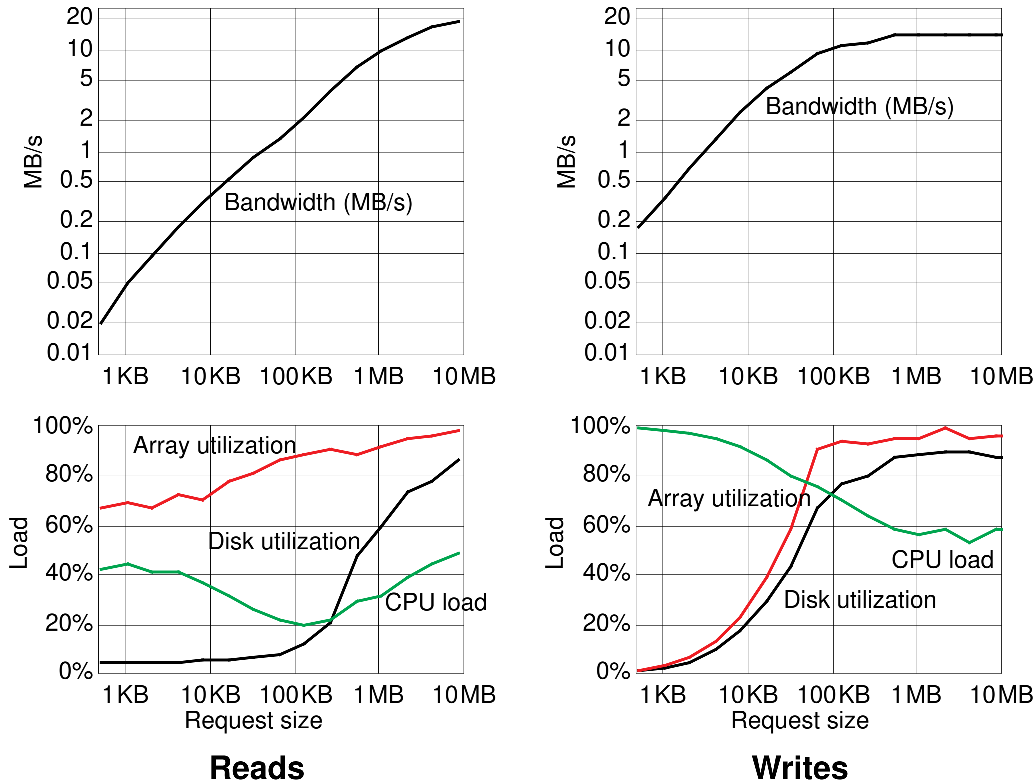


Figure 6-6. Sawmill single stream performance

This figure shows the performance of the Sawmill file system for a single stream of read or write requests. Note that the CPU load is fairly high for any request size. For small reads, the disk array is in use most of the time, but disk utilization is low since only a single disk is active. As the request size increases, bandwidth increases. Disk utilization climbs as requests get striped across multiple disks. For writes, the log-structured file system has a large impact on small requests. Note that small writes are entirely CPU limited. The time to accept requests and to lay out data into the log is the limiting factor; the time to write the large log segments to disk is negligible. With larger requests, the disk utilization becomes much higher, but CPU load is still high.

---

writes are batched together and written sequentially, very many small writes take place before a disk operation occurs. Thus, the file system overhead to perform this batching dominates small write performance. As the request size increases, per-operation CPU time becomes less important, causing disk usage to climb and CPU usage to drop. These performance measurements omit the extra cost of writes that modify less than a file block; in this case an additional read would be required to fetch the unmodified data.

In order to compare the Sawmill file system to the raw disk array and the array configured as a RAID, the single-stream bandwidths of each are shown in Figure 6-7. As for the single-stream RAID measurements, the raw disk requests are scaled to make the access patterns comparable; requests smaller than 64 KB used a single disk and larger requests used multiple disks.

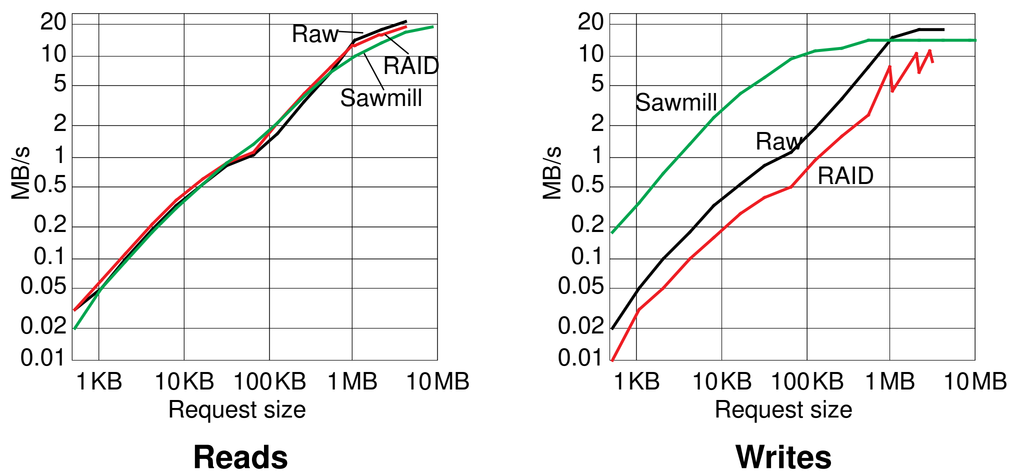


Figure 6-7. Raw disk, RAID, and Sawmill performance: single stream

These graphs show the bandwidth of the raw disk array, the disk array treated as a RAID, and the Sawmill file system for a single request stream. As explained in the text, the number of disks used for the raw array matches the number used for striping by the RAID. Request size and bandwidth are given on a log scale. For reads, the raw disk array, the RAID striped array, and the file system all have nearly the same performance, limited by disk seek times. For large requests, processing latencies reduce the bandwidth of RAID and the Sawmill file system. The right graph shows write performance. Because of logging, Sawmill performance is much better than the raw disk or RAID performance for small operations.

For a single request stream, Sawmill read bandwidth is close to the raw disk performance. There is a small performance loss for very large requests due to the per-request file system overhead of Sawmill and due to reads broken across multiple LFS segments. For large sequential reads, maximum bandwidth is 21 MB/s, slightly below the peak raw disk bandwidth of about 24.5 MB/s. For individual small random reads, as shown in Figure 6-7, performance is limited by the file system overhead and the seek time of the disks, resulting in a minimum latency of about 20 ms per operation.

For writes, Figure 6-7 illustrates the benefits of a log-structured file system and also shows the performance lost due to CPU load. Because LFS groups small writes together, bandwidth is about 20 times that of the RAID for a single stream of requests. The improvement would be even higher with a faster CPU; if there were no CPU load, the Sawmill line would be approximately flat around 15 MB/s, since the segment size written to disk is fixed. A more realistic projection is to consider a CPU that is ten times faster, such as a DEC Alpha. Small write performance would scale almost linearly, since small writes are totally CPU bound. This would result in small write bandwidth of 2 MB/s, about 200 times the performance of writing directly to the RAID. For a single stream of large requests Sawmill is slightly better than the RAID because Sawmill can have several segment writes outstanding, while the RAID driver handles the stream of requests sequentially.

The choice of LFS segment size has a critical effect on performance. In measurements with the 960K segment size replaced by a 1 MB segment size, for example, peak read bandwidth dropped to 8 MB/s and write bandwidth dropped to 5 MB/s. This is explained by the disk array stripe layout: for peak efficiency, 15 of the 16 disks must be handling data and the 16th disk must be handling parity. With the larger segment size, one disk must handle two 64KB stripe units and this disk becomes a bottleneck. It is also important that LFS always completely fill each segment. At one point, a single 512 byte sector at the end of each segment was unused and was not written to disk. Since the segment was no longer a full RAID stripe, a read-modify-write was required. This cut write performance approximately in half. The simple modification of writing the unused 512 bytes restored the full write performance.

6.2.2. Multiple stream performance

This section examines Sawmill's performance when there are multiple streams operating in parallel. This indicates the performance if, for example, multiple applications or multiple clients are using the system. Figure 6-8 shows read performance of Sawmill with concurrency from multiple requests. Because of logging, writes in Sawmill are inherently parallel, and there is no benefit from handling multiple requests. Thus, write performance is the same as in the previous section and is not shown in Figure 6-8.

The limiting factor to concurrency in our system is the CPU load. Unfortunately, the file server CPU is relatively slow and cannot handle the full potential concurrency of the system. Only 3 or 4 Sawmill operations can be run in parallel before the CPU becomes a bottleneck. Since the disk array could support 16 operations in parallel, small operation throughput would increase substantially with a faster processor.

For comparison, Figure 6-9 shows the performances of the raw disks, the RAID-configured disks, and the Sawmill file system when handling multiple requests. The degree of concurrency used depended on how much concurrency the system could handle before the CPU saturated. The raw disks handled 16 independent request streams. The RAID handled 8 independent streams for small requests, with the number decreasing as the request size increased. Sawmill handled 4 concurrent small read requests. For Sawmill, no write concurrency was required because LFS results in all the disks being used in parallel.

The effects of multiple streams can be noted by comparing Figure 6-7 to Figure 6-9. For reads, multiple stream performance is almost entirely determined by the degree of parallelism possible before the CPU saturates. Thus, the raw disk has the best performance, the RAID-configured disk array is second, and Sawmill is lowest. For large requests, however, Sawmill is able to obtain about 80% of the raw disk bandwidth. With a faster processor, Sawmill would be able to support more streams and its performance would be close to

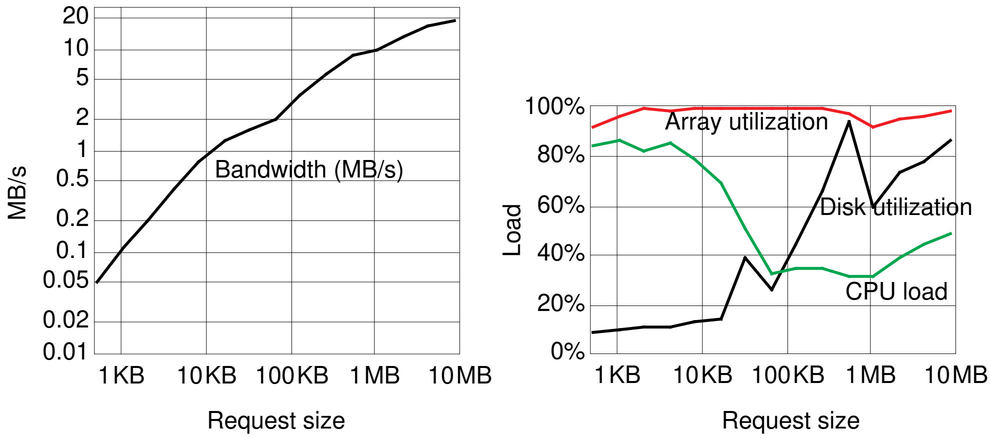


Figure 6-8. Sawmill multiple read streams

This figure shows the performance of the Sawmill file system under multiple read requests. Performance improves over a single request stream, especially for small requests, since disks can be used in parallel. The limiting factor for small requests is the CPU load, which allows only about 4 parallel streams before the CPU saturates. The disk utilization shows large fluctuations for unknown reasons. Graphs for multiple write streams are not given; because the log-structured file system batches together requests, the performance is not affected by multiple streams.

---

the raw disk performance for all request sizes, as in Figure 6-7. For writes, Figure 6-9 shows that Sawmill performs about a factor of 3 better than the RAID array. This shows that even with the additional processing cost of the file system, the benefits of logging are clearly visible. With a faster CPU, the improvement of Sawmill over RAID would be even larger and Sawmill would perform better than the raw disk for small requests. As with reads, Sawmill performs worse than the raw disk because the CPU saturates. Due to the benefits of logging, however, Sawmill's small write performance is closer to that of the raw disk than is its read performance.

6.2.3. Sawmill operation rate

Besides bandwidth, another important measurement is the total number of I/O operations the disk array can support per second. Table 6-4 shows the number of I/O operations per second the system can support, and compares a single request stream with multiple request streams. These measurements are for 4 KB random operations.

The table shows that for a single raw disk, the number of operations per second is almost entirely limited by the 7 ms rotational latency and the 12 ms average seek time. With 16 raw disks, Table 6-4 shows that the number of operations almost scales by a factor of 16, but there is about a 5% penalty. This is due to the CPU load of about 75%, which causes some delay in starting requests.

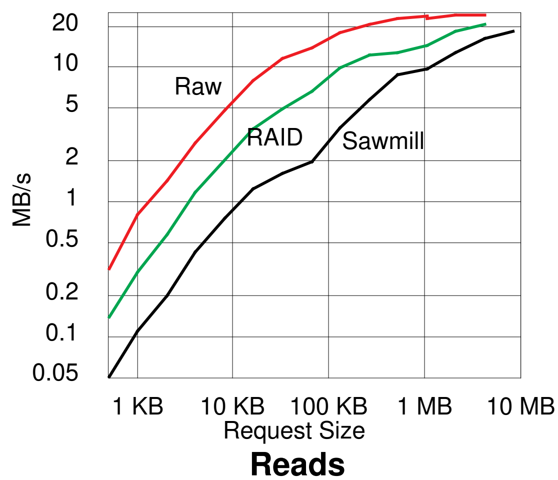


Figure 6-9. Raw disk, RAID, and Sawmill performance: multiple streams

These graphs show bandwidth of the raw disk array, the disk array treated as a RAID, and the Sawmill file system when receiving multiple requests. Read bandwidth of RAID and Sawmill are lower than for the raw disk because the CPU load limits the possible parallelism. For writes, note that because of LFS, the performance of Sawmill exceeds that of the RAID. The spikes in RAID write performance at 960 KB and 1.92 MB illustrate the increased write efficiency when a full stripe is written at once; other writes require an expensive parity update.

Type	Read operations per second	Write operations per second
1 disk	50	45
16 disk array	710	670
RAID: 1 stream	55	25
RAID: 8 streams	300	100
Sawmill: 1 streams	45	345
Sawmill: 4 streams	95	345

Table 6-4. I/O rate for small operations

This table shows the number of random 4 KB I/O operations RAID-II can handle per second under various configurations. CPU load limits the number of parallel requests that RAID and Sawmill can handle.

When the disk array is treated as a RAID, the I/O rates change significantly. A single stream of read requests to the RAID uses a single disk, so performance is approximately the same as for a single disk. (Performance in Table 6-4 is slightly better for the RAID because seek distance was shorter for the striped requests.) A single small write stream has about half of the read performance due to the read-modify-write parity update. With multiple parallel request streams, the RAID uses disks in parallel, increasing the total number of operations. Because the CPU becomes a bottleneck, the increase is significantly less than the potential factor of 16 for reads and factor of 8 for writes. A faster CPU would significantly increase the total number of I/Os per second that the RAID can support.

Finally, Table 6-4 shows the operation rates of the Sawmill file system for small requests. Comparing the read measurements to a raw disk or the RAID with a single request stream shows that there is a 10 to 20% performance loss due to Sawmill; this is due to overhead in the file system. Due to the CPU load, Sawmill reads obtained even less benefit than RAID reads from concurrency. The CPU saturated with 4 concurrent requests yielding only a factor of 2 improvement over a single stream. Thus, multiple stream read performance is much worse for Sawmill than for the RAID or the raw disk array. With a faster CPU, however, Sawmill should perform within 10 to 20% of the raw disk or RAID, as it did for a single stream. For a single write stream, Sawmill does very well; the number of I/Os per second is more than an order of magnitude better than RAID for a single stream and a factor of seven better than the raw disk. This clearly shows the benefit of logging. Multiple stream performance is limited by the CPU; Sawmill is half the speed of the raw disk, but still a factor of three faster than RAID. With a faster CPU, Sawmill would do even better, since write performance is CPU limited, not disk limited.

6.3. Scalability

This section describes how the performance measurements are likely to change with improvements in technology. Table 6-5 summarizes the results.

Large reads	Limited by disk array bandwidth. Will improve with larger arrays and faster disks.
Large writes	Limited by disk array bandwidth.
Small reads	Limited by disk positioning time. Unlikely to improve rapidly.
Small writes	Limited by CPU speed. Will improve with faster processors.
Concurrent requests	Limited by CPU speed.

Table 6-5. Sawmill bottlenecks and future trends

This table summarizes the bottlenecks in Sawmill and how they will be affected by changes in technology.

Large reads and writes already achieve close to the raw system bandwidth with Sawmill. Thus, large requests will only get faster with storage systems that have more or faster disks. The CPU speed will have to increase proportionally or else the CPU will become a bottleneck. However, since the current server is a relatively slow Sun-4, much faster CPUs are currently available, and processor speeds are rapidly increasing, it is unlikely that CPU performance would become a problem for large operations.

Small individual random reads are limited by the disk positioning time, which is likely to improve relatively slowly compared to improvements in CPU speed. Faster CPU speeds will improve the potential parallelism of small reads, which was limited by our CPU. Increasing the number of disks will increase the total number of independent reads that can be carried out simultaneously. This will increase the aggregate bandwidth of reads, but won't improve the performance of any particular request. The most likely way to improve read performance is by using prefetching to reduce the latency of individual requests. If the data is fetched before it is required, the disk latency will not delay the request. One technique for this is Gibson's Transparent Informed Prefetching [GPS93]; by obtaining hints from the application, the file system can fetch data before it is required.

Because small writes are limited by CPU speed, small write bandwidth will scale almost linearly with increases in processing power. Current high-performance workstations already have ten times the CPU power of our Sun-4 server, and even faster machines will become available over the next few years. Thus, the performance of small writes in a Sawmill-like file system should improve dramatically.

Overall, the CPU bottleneck is currently the main limitation to performance. The RAID-II design attempted to avoid a processor bottleneck by providing a fast bypass data path. This technique was successful to a large degree, since the disk array can provide data at bandwidths much higher than the file server alone could provide. Moving the CPU out of the data path, however, does not entirely eliminate the need for a fast processor. Since the current CPU is rather slow, upgrading to a more modern processor would remove this bottleneck.

6.4. A standard file system on RAID-II

As discussed in Section 3.6.1, the simplest way to use RAID-II would be to treat the disk array as a large disk attached to the server so a regular file system can be used. That is, all reads and writes would be copied over the VME link between the file server's memory and the controller. To illustrate the problem of using RAID-II as a standard disk, this approach was tested with the Unix-like Sprite file system (Sprite-FFS) and the Sprite log-structured file system (Sprite-LFS). These measurements illustrate the severe performance penalty of using a file system that isn't designed to take advantage of the controller architecture.

This section gives bandwidth measurements for Sprite-FFS and Sprite-LFS running on RAID-II. For these tests, a 5 MB file was first created with 1 MB writes. The file was then read back with 1 MB reads. The time for these operations gives the cached write and read bandwidth, since the file fit in the cache. Next, the test program wrote the file and forced the data to disk with an `fsync` operation; this provided the disk write time. The cache was flushed and the file was read back in, giving the disk read time. Performance varied only slightly with different request sizes.

	Sprite-FFS		Sprite-LFS	
	1 disk	16 disks (RAID)	1 disk	16 disks (RAID)
Disk write	230 MB/s	100 KB/s	770 KB/s	890 KB/s
Disk read	390 MB/s	380 KB/s	380 KB/s	370 KB/s
Cached write	3.0 MB/s	3.1 MB/s	2.0 MB/s	2.2 MB/s
Cached read	4.9 MB/s	4.9 MB/s	5.0 MB/s	4.8 MB/s

Table 6-6. Traditional file system on RAID-II

This table shows the bandwidth of the standard Sprite file systems (the Unix-like file system and the log-structured file system) when run unmodified and using RAID-II. The figures show the time to read and write a 5 MB file. Cached reads and writes go to the cache, not to disk. For disk writes, the file was `fsync`'d to disk after writing. For disk reads, the cache was flushed before reading.

Table 6-6 shows the bandwidth of these tests. The first result is that the bandwidth to disk is very low, less than 1 MB/s. The main reason for this is that the file systems perform small, inefficient operations that only use a single disk. This illustrates that the access patterns of standard file system do not work well with a disk array. In fact, except for writes with Sprite-LFS, the file systems get no benefit from a disk array over a single disk because they perform small I/Os that do not benefit from striping. The second result is that even reading and writing to the cache is much slower than the bandwidth available from Sawmill. That is, a standard file system without any disk accesses is much slower than Sawmill with disk accesses and can't support the data rates of Sawmill. This is largely due to the cost of copying data from the cache.

Several other interesting conclusions can be reached from Table 6-6. The overhead of computing parity is clear from comparing the 1-disk and 16-disk Sprite-FFS measurements: writes are twice as slow due to the read-modify-write required with the RAID. Also, comparing writes on Sprite-FFS and Sprite-LFS illustrates that the write benefits of using a log-structured file system on a RAID occur even with unmodified Sprite-LFS over Sprite-FFS. Finally, for unknown reasons Sprite-FFS was much faster than Sprite-LFS for cached writes.

Figure 6-9 can be used to estimate the write performance of a RAID-II file system based on the Unix FFS [MJLF84] but optimized for the RAID-II data path. Since the file system writes blocks to the RAID, the maximum potential bandwidth is indicated by the RAID performance line for the appropriate block size. (This significantly overestimates the potential bandwidth since it ignores the file system CPU load, request processing latency, and the cost of writing metadata.) Thus, with 16 KB blocks, the Unix-based file system would provide at most 1.1 MB/s for writes. In comparison, Sawmill provides 4.1 MB/s because of the large writes from logging. Even if the Unix file system uses clustering [MK91], performance would still be much below Sawmill: with 64 KB clusters, the Unix-based file system would still provide at most 2 MB/s, still half the speed of Sawmill. This clearly illustrates the benefits of a log-structured file system, which performs efficient large writes. With a faster processor, the benefits of Sawmill would be even more dramatic because small writes are currently CPU limited. Sawmill would have additional overhead due to cleaning, but even taking cleaning costs into account, Sawmill still would come out far ahead.

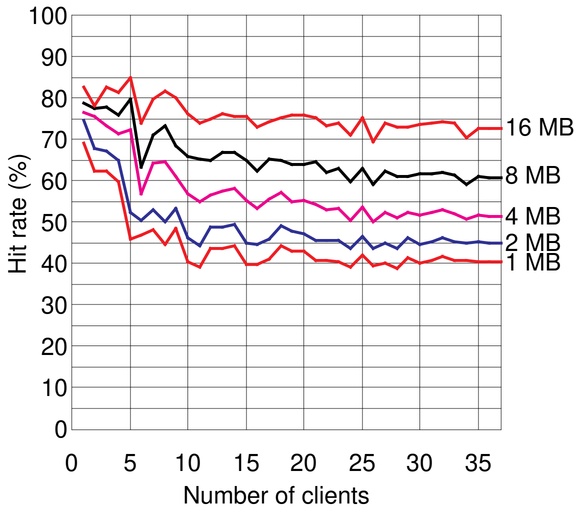
6.5. Simulated block cache hit rates

One of the motivations for not using a block cache in Sawmill was the belief that a server block cache would not be effective for high bandwidth applications. In this section, simulation results are used to examine how server cache hit rates scale with many clients in an office/engineering environment. One possibility would be for clients to share very little data, resulting in a cache size that would have to scale with the number of clients, and thus the data bandwidth. Another possibility would be for most clients to share most of the same data, resulting in a cache hit rate that is relatively independent of the number of clients. The simulation results show the former case is true in our environment: the cache hit rates show little sharing of data between clients.

To examine the behavior of a file server block cache as the number of clients increases, cache simulations were performed. The experiment used a 12 hour daytime period of system usage from the Sprite system traces [BHK+91]. To simulate the effects of different numbers of client machines, requests were processed from various subsets of the active machines. The experimental procedure was to select random groups of 1 to 37 of the client machines and trace the server cache performance over 12 hours assuming no client caching. The hit rate for reads was computed for cache sizes of 512 KB to 4 MB, using a cache simulator with a least-recently-used (LRU) replacement policy. To minimize cold-cache effects, hit rate statistics were not collected for the first half hour so these references were used to load the cache; increasing the cache warming time had little effect. Because Sprite has process migration, jobs can run on idle clients. For these measurements, migrated jobs were treated as coming from the client that started the job, rather than the client that executed the job.

Figure 6-10 shows the results of the simulations. Several general features can be noted from the graphs. First, there is a great deal of variance in hit rate, depending on which clients were selected; this indicates that hit rate is very dependent on the particular application, even in the same environment. Second, larger caches greatly increase the hit rate, especially with many clients. Finally, the hit rate variance decreases as the number of clients increases. This occurs because with more clients, the effect of any particular application on a client is reduced. As well, with a large number of clients, the clients selected in different simulations are likely to overlap.

The key result of Figure 6-10 is that cache performance drops as the number of clients increases, although this drop slows as the number of clients continues to increase. To maintain a good hit rate, the cache size must increase as the number of clients increases. Therefore, a high-bandwidth storage system such as Sawmill would require a significantly larger cache than a slower storage system if the high-bandwidth system is used to support additional clients. In more detail, for small numbers of clients, the cache size must scale



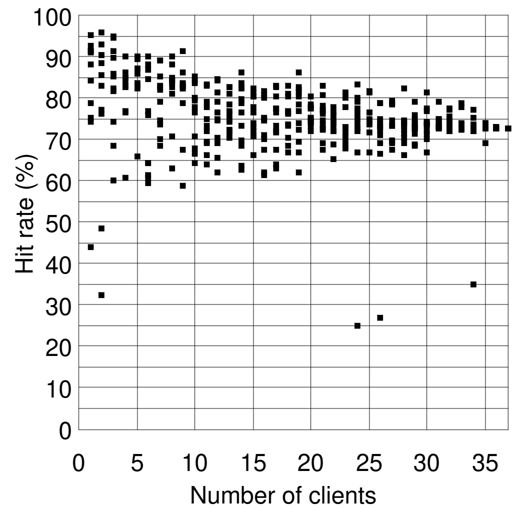


Figure 6-10. Hit rate vs. number of clients

These graphs show the block cache hit rate as a function of the number of clients. The left graph shows the average hit rate for caches from 1 MB to 16 MB. Note that the hit rate decreases as the number of clients increases. This indicates that the working set size increases and a larger cache is required as the number of clients increases. The right graph shows the hit rate for each of the 16 MB cache simulations; the top curve in the left graph is the average of these simulations. Each point indicates a different selection of clients. This graph illustrates the wide variance in the hit rate depending on the clients used.

---

approximately linearly for a constant hit rate. Consider, for example, the cache size required to get a 70% hit rate. While a 256 KB cache would work for a single client, a 1 MB cache would be required for 5 clients, a cache over 2 MB for 10 clients, and a cache under 4 MB for 37 clients. Thus, the cache size must increase with the number of clients until about 10 clients, and then the cache size can increase more gradually.

Figure 6-11 shows the average total read bandwidth as a function of the number of clients. This figure illustrates that the bandwidth required by the clients is fairly low, so Sawmill could service many more clients than in these simulations, and would require a correspondingly large cache. Not surprisingly, the bandwidth is approximately a linear function of the number of clients, although there is a great deal of variance depending on which clients are selected. The average bandwidth consumed by the clients is relatively small; less than 1 KB/s per client. Although average bandwidth is low, peak bandwidths of up to 10 MB/s occurred (serviced from client caches). This is due to the bursty nature of client requests.

These cache simulations have several implications for storage system designs. First, they indicate that for a typical university workload, the server cache size must scale

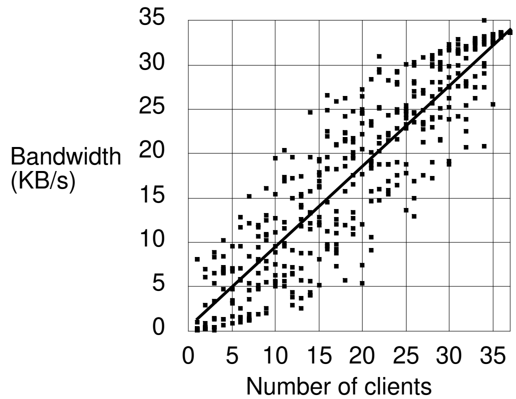


Figure 6-11. Average bandwidth vs. number of clients

This figure shows the average total read bandwidth for the various simulation runs. The solid line is a least-squares fit to the data. Note that the average bandwidth is relatively low due to the bursty nature of file system requests.

---

approximately linearly with the number of clients, for a small number of clients. As the number of clients increases further, the cache size does not need to increase as fast. Second, cache sizes must be large. To get a 70% hit rate with 35 clients required a 16 MB cache; thus with enough clients to fully utilize the bandwidth of Sawmill, a significantly larger cache would be required. This validates the hypothesis that RAID-II did not have sufficient memory for an effective high-bandwidth block cache. Finally, since client caching reduces the locality of server requests, cache performance would be even worse in a system with client caching.

6.6. Disk address cache performance

As explained in Section 4.5, Sawmill caches parts of the inode block map, which stores the disk addresses of data blocks. In the traditional Sprite file system, each time a block is accessed, its disk address is determined by looking in the inode or indirect block. In contrast, Sawmill prefetches the disk addresses of blocks into the cache when a file is opened for reading. Then, subsequent reads can take place without accessing the inode to obtain block addresses.

Measurements show the benefits of the disk address cache. Loading block addresses into the cache averages 3 μs per block, after which the time to access a cached entry is negligible. In comparison, the original file system took 40 μs to look up a block's disk address even if the inode was in the block cache. The disk address cache is so much faster because it can be loaded with the addresses from an entire inode or indirect block at once rather than accessing them one at a time as needed. Because finding and locking takes up most of the time, the saving here is very large. Thus, the cache is beneficial because block lookup is much more efficient. Note that the cache size is very small, since only a one word address is required for each block. For example, a 20 MB file requires only 5 KB to cache all the addresses.

Sawmill bandwidth measurements before the implementation of the address cache found that looking up block addresses was a substantial fraction of the computational cost of performing reads. The above measurements illustrate why: reading 20 MB/s of 16KB blocks requires 1280 lookups per second. At 40 µs each, this requires over 50 ms. That is, over 5% of processing time was spent just looking up block addresses. With the original block size of 4 KB, overhead was even worse; block lookup time was 20% of disk access time.

In conclusion, the block cache helps Sawmill both by preventing blocks from being looked up multiple times and by allowing block lookups to be performed efficiently in groups, rather than inefficiently as single lookups.

6.7. Conclusions

For large requests Sawmill provide about 80% of the raw bandwidth of the disk array, reading data at a peak rate of 21 MB/s and writing data at 15 MB/s. This illustrates that high file system throughput can be provided by the combination of a high-bandwidth controller such as RAID-II and a log-structured file system. Since the peak bandwidth of Sawmill greatly exceeds the data rates that the file server itself can handle, it is clear that a controller data path that bypasses server memory is very beneficial.

The log-structured file system improved the speed of a small write stream by an order of magnitude over writing directly to the RAID, and with a faster processor small writes would be even faster. This illustrates that a log-structured file system is effective in providing good performance for small writes to a disk array.

The high CPU load in many of the measurements shows that even with hardware support, there are still significant demands on the file system CPU, and a fast processor is still required. Our current CPU was a bottleneck for small writes and concurrent operations. However, since the Sun-4 used with Sawmill is relatively slow, processor bottlenecks are unlikely to limit performance of disk arrays if a high-performance workstation is used to run the file system.

The measurements of Sawmill suggest that performance of an architecture such as RAID-II will scale with changes in technology. With a faster CPU, write and concurrent read performance would increase accordingly. A larger disk array would increase peak read and write bandwidth for large operations. Small reads, however, are largely dependent on disk positioning times, and thus are less likely to improve with changes in technology.

A standard workstation file system performs very poorly on RAID-II. This is shown by Sprite-LFS and Sprite-FFS, which obtained less than 1 MB/s of bandwidth when running on RAID-II. This illustrates the benefit of a file system such as Sawmill that is designed to operate well with a disk array.

Trace-driven cache simulations validate the decision not to use a block cache in Sawmill. These simulations show that to obtain a particular cache hit rate, the cache size must scale close to linearly with the number of clients. A 16 MB cache would be required to obtain a 70% hit rate with 35 clients, even though the total bandwidth was much less than Sawmill could provide. Thus, with enough clients to fully use Sawmill, the cache size would have to be even larger.

Measurements show the benefits of the Sawmill disk block address cache. Without the address cache, block address lookups added a 20% overhead to reads. By caching these lookups, the cost was reduced more than an order of magnitude.

7 Conclusions

"To write and read comes by nature." – Shakespeare

Providing high-bandwidth, inexpensive, reliable file storage is an important goal for computer systems. New hardware techniques can help meet this need, in particular RAID disk arrays to provide high bandwidth and specialized controllers to move this data to the clients. The challenge for file systems is to use this hardware well: to deliver its raw bandwidth to applications.

7.1. Summary of results

This dissertation has described Sawmill, a high bandwidth file system that uses the RAID-II disk array. Sawmill runs on a cost-effective workstation file server and provides high-bandwidth access to data. By taking advantage of the controller's design, Sawmill provides data rates much higher than the memory bandwidth of the file server. The following sections summarize the key results from my work on Sawmill.

7.1.1. A log-structured file system works well with RAID

A log-structured file system works well with a RAID disk array, and can substantially improve performance. The main drawback of a RAID is the parity update overhead for small writes. By performing large sequential writes, however, the log-structured file system minimizes this write cost. The log-structured file system improved performance of a small write stream by an order of magnitude over the performance of the RAID, and with a faster processor small write performance would be improved even more.

7.1.2. Existing file systems don't work well with RAID

A standard Unix-based or LFS-based workstation file system has very poor performance when run on RAID-II, obtaining much less than 1 MB/s, as shown in Section 6.4. This illustrates that traditional file systems don't work well with a RAID because they perform small operations and they don't work well with a bypass controller architecture because they are designed to move data through kernel memory.

7.1.3. File systems can take advantage of a bypass data path

Sawmill illustrates that a file system designed to use a bypassing controller such as RAID-II can provide data at bandwidths much higher than the file server itself can move. Thus, storage systems can supply high data rates without requiring the file server itself to support the data rate. Measurements show that for large requests Sawmill operates at close to the raw bandwidth of the disk array, reading data at a peak rate of 21 MB/s and writing data at 15 MB/s.

7.1.4. Streaming can be better than caching

One main feature of the Sawmill implementation was to reduce the processing overhead by eliminating the file system block cache. This allows data transfers to take place efficiently as large streams, rather than breaking requests into small blocks. For reads, controller memory is used instead for prefetching and other types of buffering. For writes, a new technique called on-the-fly layout was used to perform layout as data blocks arrive.

7.2. Lessons about RAID-II

This section describes lessons learned about RAID-II during the implementation and testing of the Sawmill file system. On the whole, RAID-II was successful in its goals of functioning as a high-bandwidth storage server. Several components of the system, however, were performance bottlenecks or required additional effort in the file system to avoid bottlenecks.

7.2.1. RAID-II architecture is beneficial

Comparing the performance of RAID-II to RAID-I shows the benefits of the bypass data path between the disks and the network, avoiding the file server. As was shown in Chapter 6, the Sawmill file system can handle reads at 21 MB/s, while the file server itself could only move data at about 5 MB/s. This illustrates the benefits obtained by being able to move data without having the server in the datapath.

The controller memory also improves performance. By buffering requests, it smooths out the disk and network traffic, compared to a system where the two are tightly coupled. Also, the buffering permits prefetching, which is important for improving read performance. Finally, implementing a log-structured file system would be much harder in a system that cannot buffer segments in memory.

7.2.2. Better communication between server and controller needed

The link between the server and the RAID-II controller is too slow compared to the speed of the rest of the system. Because the link has high latency and low bandwidth, metadata operations are fairly expensive if they have to access metadata from RAID-II. For good performance, metadata must be cached in file server memory to minimize transfers across the link. If the server could access controller memory at close to kernel memory speeds, the file system design could be greatly simplified. The current link speed is not a good match for the speed of the rest of the system, as it is almost an order of magnitude slower than the disk and network transfer speeds.

7.2.3. More buffer memory needed

The memory capacity on the RAID-II controller is too small. To provide an effective cache, the memory should be considerably larger. Even as buffers, the memory cannot support a large number of parallel streams. A substantial amount of controller memory is needed for low-level RAID operations. The result is that the amount of memory available for the file system to use is limited. With a much larger amount of memory, a file system cache might be a useful addition to the Sawmill file system.

7.2.4. More powerful server CPU needed

The current file server CPU is too slow. As shown in Chapter 6, the CPU became a performance bottleneck in many cases. Although the RAID-II architecture greatly reduces the load on the file server's memory system and reduces the CPU load from copying, the CPU performance is still important, especially since the data rates are much higher with RAID-II. The main loads on the CPU are handling incoming requests, providing the file system abstractions, performing striping computations for RAID, and handling the low-level disk interrupts.

Besides increasing the CPU performance, more processing could be off-loaded to the controller, leaving less load for the server. In particular, there should be more support on the RAID-II for disk array operations. Currently, the file server must issue each disk operation individually and take an interrupt for each operation. This puts a heavy load on the server just to keep the disks busy. With lower-level support for this, the server could issue a single disk array operation and take a single interrupt when it completes. This would greatly improve the ability of the system to scale to larger and faster disk arrays.

7.2.5. Server should be on the fast network

In the current RAID-II system, the file server is not directly connected to the fast network. The fast network is connected only to the RAID-II controller and to clients. Thus, the server must access RAID data through the VME link. The result of this is that the server's access to data is slower than the clients' access, and the server must use an entirely different data path, as discussed in Section 7.2.2.

7.3. Future Work

Probably the most important unanswered issue with Sawmill is the cost of cleaning in a log-structured file system, which was described in Section 5.3. Cleaning was not implemented in Sawmill, so performance measurements are not available. Previous work [Ros92] indicated that overall cleaning costs would be low. However, [SBMS93] and [SSB$^+$95] found cleaning costs to be high for some workloads, such as transaction processing. Costs were high particularly in environments with largely full disks, and cleaning could potentially cause service interruption. The impact of cleaning on the performance of log-structured file systems continues to be a topic requiring additional investigation. In particular, the effect of cleaning on performance is different for a RAID than for an individual disk because of the small-write parity problem of RAID devices.

Another major issue is how to improve the performance of small reads. Since small reads are limited by disk positioning time, unlike other operations, their performance will not improve as fast with changes in technology. Thus, techniques such as prefetching are likely to play an increasing role in improving read performance.

Finally, log structured file systems provide opportunities for improving large read performance by data reorganization, as explained in Section 5.3.3. After random writes, data may not be stored sequentially on disk, resulting in lower performance for later sequential reads. However, by reorganizing data on disk so the data is sequential, the reorganized data can be more efficient for reads. Thus, reorganization on Sawmill could improve sequential read performance. This reorganization could either take place during idle times in the system, or it could be integrated with cleaning so cleaned data is written back in a better pattern for reads.

7.4. Concluding thoughts

As processor speeds continue to increase faster than disk speeds, disk arrays will continue to increase in popularity. With the spread of local area networks and client-server architectures in the personal computing world, the need for high-bandwidth data storage will become even more widespread. The likely consequence is increased use of RAID disk arrays. File system technology must keep up with these changes. Since log-structured file systems such as Sawmill make good use of RAID disk arrays and keep small writes from limiting performance, research into log-structured file systems is important for the future.

8 Bibliography

[Bak94] M. Baker. *Fast Crash Recovery in Distributed File Systems*. PhD thesis, University of California, Berkeley, CA 94720, January 1994. Technical Report UCB/CSD 94/787.

[BCGI93] L. Buck, S. Coleman, R. Garrison, and D. Isaac. Reference model for open storage systems interconnection: Mass storage system reference model version 5. IEEE Storage System Standards Working Group, October 1993.

[BEMS91] A. Bhide, E. Elnozahy, S. Morgan, and A. Siegel. A comparison of two approaches to build reliable distributed file servers. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 616–623, 1991.

[BHK$^+$91] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th Symposium on Operating System Principles*, pages 198–212, Pacific Grove, CA, October 1991.

[BN84] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[CFL94] P. Cao, E. W. Felten, and K. Li. Application-controlled file caching policies. In *Proceedings of the 1994 USENIX Summer Conference*, pages 171–182, June 1994.

[CK91] A. L. Chervenak and R. H. Katz. Performance of a disk array prototype. In *Proceedings of the 1991 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 188–197, 1991.

[CL91] L. Cabrera and D. Long. Exploiting multiple I/O streams to provide high data-rates. In *Proceedings of the 1991 USENIX Summer Conference*, pages 31–48, June 1991.

[CLD$^+$94] P. Chen, E. Lee, A. Drapeau, K. Lutz, E. Miller, S. Seshan, K. Shirriff, D. Patterson, and R. Katz. Performance and design evaluation of the RAID-II storage server. *Journal of Distributed and Parallel Databases*, 2(3):243–260, July 1994. Also appeared in International Parallel Processing Symposium 1993 Workshop on I/O.

[CLVW93] P. Cao, S. Boon Lim, S. Venkataraman, and J. Wilkes. The TickerTAIP parallel RAID architecture. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 52–63, 1993.

[CMS90] B. Collins, C. Mercier, and T. Stup. Mass-storage system advances at Los Alamos. In *Digest of Papers, Proc. Tenth IEEE Symposium on Mass Storage Systems*, pages 77–81, May 1990.

[dJKH93] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The Logical Disk: A new approach to improving file systems. In *Proceedings of the Fourteenth SOSP, Operating Systems Review*, volume 27, pages 15–28, December 1993.

[DSH+94] A. L. Drapeau, K. Shirriff, J. H. Hartman, E. L. Miller, S. Seshan, R. H. Katz, K. Lutz, D. A. Patterson, E. K. Lee, P. M. Chen, and G. A. Gibson. RAID-II: A high-bandwidth network file server. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 234–244, Chicago, IL, April 1994.

[FC87] R. Finlayson and D. Cheriton. Log files: An extended file service exploiting write-once storage. In *Proceedings of the Eleventh SOSP, Operating Systems Review*, pages 139–148, 1987.

[GA94] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. In *Proceedings of the 1994 USENIX Summer Conference*, pages 197–207, June 1994.

[Gib92] G. A. Gibson. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. ACM Distinguished Dissertations. MIT Press, 1992.

[GPS93] G. A. Gibson, R. H. Patterson, and M. Satyanarayanan. A status report on research in transparent informed prefetching. *ACM Operating System Review*, 27(2):21–34, April 1993.

[Hag87] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the Eleventh SOSP, Operating Systems Review*, pages 155–162, 1987.

[HBM+89] A. Hisgen, A. Birrell, T. Mann, M. Schroeder, and G. Swart. Availability and consistency tradeoffs in the Echo distributed file system. In *Proceedings of the Second Workshop on Workstation Operating Systems (WWOS-II)*, pages 49–54, September 1989.

[HCF+90] C. Hogan, L. Cassell, J. Foglesong, J. Kordas, M. Nemanic, and G. Richmond. The Livermore Distributed Storage System: Requirements and overview. In *Digest of Papers, Proc. Tenth IEEE Symposium on Mass Storage Systems*, pages 6–17, May 1990.

[HKM+88] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[HN90] W. A. Horton and B. Nelson. The Auspex NS 5000 and the Sun SPARCstation 490 in one- and two-Ethernet NFS performance comparisons. Technical Report 2, Auspex Systems Inc., May 1990.

[HO93] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. In *Proceedings of the 14th Symposium on Operating System Principles*, pages 29–43, Asheville, NC, December 1993. ACM.

[HP90] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, 1990.

[IBM91] IBM. *0661 Functional Specification Model 371, Version 0.7*, February 18, 1991.

[KLA+90] M. L. Kazar, B. W. Leverett, O. T. Anderson, V. Apostolides, B. L. Bottos, S. Chutani, C. F. Everhart, W. A. Mason, S. Tu, and E. R. Zayas. Decorum file system architectural overview. In *Proceedings of the 1990 USENIX Summer Conference*, pages 151–163, June 1990.

[Kle86] S. R. Kleiman. Vnodes: an architecture for multiple file system types in Sun UNIX. In *Proceedings of the 1986 USENIX Summer Conference*, pages 238–247, June 1986.

[Koc87] P. D. L. Koch. Disk file allocation based on the buddy system. *ACM Transactions on Computer Systems*, 5(4):352–370, November 1987.

[Kot91] D. Kotz. *Prefetching and Caching Techniques in File Systems for MIMD Multiprocessors*. PhD thesis, Duke University, April 1991. Available as technical report CS-1991-016.

[Lee93] E. K. Lee. *Performance Modeling and Analysis of Disk Arrays*. PhD thesis, University of California, Berkeley, August 1993. Technical Report UCB/CSD 93/770.

[LMKQ89] S. Leffler, M. McKusick, M. Karels, and J. Quarterman. *The design and implementation of the 4.3BSD UNIX operating system*. Addison-Wesley, 1989.

[Min93] R. G. Minnich. The AutoCacher: A file cache which operates at the NFS level. In *Proceedings of the 1993 USENIX Winter Conference*, pages 77–83, 1993.

[MJLF84] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[MK91] L. McVoy and S. Kielman. Extent-like performance from a UNIX file system. In *Proceedings of the 1991 USENIX Winter Conference*, pages 33–43, 1991.

[MM92] J. Menon and D. Mattson. Comparison of sparing alternatives for disk arrays. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 318–329, May 1992.

[MRK93] J. Menon, J. Roche, and J. Kasson. Floating parity and data disk arrays. *Journal of Parallel and Distributed Computing*, 17:129–139, 1993.

[MSC+90] J. Moran, R. Sandberg, D. Coleman, J. Kepecs, and B. Lyon. Breaking through the NFS performance barrier. In *Proceedings of the 1990 Spring European UNIX Users Group*, pages 199–206, Munich, Germany, April 1990.

[MT90] J. Merrill and E. Thanhardt. Early experience with mass storage on a Unix-based supercomputer. In *Digest of Papers, Proc. Tenth IEEE Symposium on Mass Storage Systems*, pages 117–121, May 1990.

[Nel88] M. N. Nelson. *Physical Memory Management in a Network Operating System*. PhD thesis, University of California, Berkeley, CA 94720, November 1988. Technical Report UCB/CSD 88/471.

[Nel90] B. Nelson. An overview of functional multiprocessing for network servers. Technical Report 1, Auspex Systems Inc., July 1990.

[NWO88] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[OCD+88] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[Ous90] J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Summer USENIX '90*, pages 247–256, Anaheim, CA, June 1990.

[PGK88] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD Conference*, pages 109–116, June 1988.

[PJS+94] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3: Design and implementation. In *Proceedings of the 1994 USENIX Summer Conference*, pages 137–151, 1994.

[Pos81] J. Postel. *Transmission Control Protocol - DARPA Internet program protocol specification*, RFC 793, Network Information Center, SRI International, September 1981.

[PR85] J. Postel and J. Reynolds. *File Transfer Protocol (FTP)*, Request for Comments (RFC) 959, October 1985.

[PZ91] M. Palmer and S. B. Zdonik. Fido: A cache that learns to fetch. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 255–264, 1991.

[RB89] A. L. Reddy and Prithviraj Banerjee. A study of parallel disk organizations. *Computer Architecture News*, 17(5):40–47, September 1989.

[Ros92] M. Rosenblum. *The Design and Implementation of a Log-structured File System*. PhD thesis, U.C. Berkeley, June 1992. Report UCB/CSD 92/696.

[SAS89] M. Stonebraker, P. Aoki, and M. Seltzer. Parallelism in XPRS. Technical Report UCB/ERL M89/16, UC Berkeley, February 1989.

[SBMS93] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *1993 Winter Usenix*, pages 307–326, January 1993.

[Sea91] Data storage technology trends and developments, Seagate, 1991.

[Sel93] M. Seltzer. *File System Performance and Transaction Support*. PhD thesis, University of California, Berkeley, January 1993. Technical Report UCB/ERL M93/1.

[SGH93] D. Stodolsky, G. Gibson, and M. Holland. Parity logging overcoming the small write problem in redundant disk arrays. *Computer Architecture News*, 2(2):64–75, May 1993.

[SGK+85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the 1985 USENIX Summer Conference*, pages 119–130, 1985.

[SM89] V. Srinivasan and J. C. Mogul. Spritely NFS: experiments with cache-consistency protocols. In *Proceedings of the Twelfth SOSP, Operating Systems Review*, volume 23, pages 45–57, 1989.

[Smi85] A. J. Smith. Disk cache – miss ratio analysis and design considerations. *ACM Transactions on Computer Systems*, 3(3):161–203, August 1985.

[SO92] K. Shirriff and J. Ousterhout. A trace-driven analysis of name and attribute caching in a distributed file system. In *Proceedings of the Winter 1992 USENIX Conference*, pages 315–331, January 1992.

[SS91] M. Seltzer and M. Stonebraker. Read optimized file system designs: A performance evaluation. In *Proceedings of the Seventh International Conference on Data Engineering, Obe, Japan*, pages 602–611, April 1991.

[SSB+95] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: a performance comparison. In *1995 Winter Usenix*, pages 249–264, January 1995.

[Sun89] Sun Microsystems, Inc. *NFS: Network File System Protocol Specification*, RFC 1094, Network Information Center, SRI International, March 1989.

[TD91] C. D. Tait and D. Duchamp. Detection and exploitation of file working sets. In *Proceedings of the 11th International Conference on Distributed*

Computing Systems, pages 2–9, 1991.

[Thi93] Thinking Machines Corp. *The Connection Machine system: CM-5 I/O system programming guide*, Thinking Machines Corp., Cambridge, MA 1993.

[Twe90] D. Tweten. Hiding mass storage under Unix: NASA's MSS-II architecture. In *Digest of Papers, Proc. Tenth IEEE Symposium on Mass Storage Systems*, pages 140–145, May 1990.

[Ult91] Ultra Network Technologies. *Host software porting guide*, Ultra Network Technologies, 101 Daggett Drive, San Jose, CA 1991.

[Wal83] B. Walker. The LOCUS distributed operating system. In *Proceedings of the Ninth SOSP, Operating Systems Review*, pages 49–70, November 1983.

[Wel91] B. Welch. Measured performance of caching in the Sprite network file system. *Computing Systems*, 3(4):315–342, Summer 1991. Also appears in the 3rd USENIX Symposium on Experiences with Distributed and Multiprocessor Systems.

[Wil90] D. Wilson. The Auspex NS5000 fileserver. *Unix Review*, 8(8):91–102, August 1990.

[Wil91] J. Wilkes. DataMesh - parallel storage systems for the 1990s. In *Digest of Papers, Proc. Eleventh IEEE Symposium on Mass Storage Systems*, pages 131–136, October 1991.