Sawmill: A High-Bandwidth Logging File System

Ken Shirriff
John Ousterhout
University of California, Berkeley

Abstract

This paper describes the implementation of Sawmill, a network file system using the RAID-II storage system. Sawmill takes advantage of the direct data path in RAID-II between the disks and the network, which bypasses the file server CPU. The key ideas in the implementation of Sawmill are combining logging (LFS) with RAID to obtain fast small writes, using new log layout techniques to improve bandwidth, and pipelining through the controller memory to reduce latency. The file system can currently read data at 21 MB/s and write data at 15 MB/s, close to the raw disk array bandwidth, while running on a relatively slow Sun-4. Performance measurements show that LFS improved performance of a stream of small writes by over a order of magnitude compared to writing directly to the RAID, and this improvement would be even larger with a faster CPU. Sawmill demonstrates that by using a storage system with a direct data path, a file system can provide data at bandwidths much higher than the file server itself could handle. However, processor speed is still an important factor, especially when handling many small requests in parallel.

1. Introduction

An I/O gap has arisen between the data demands of processors and the data rates 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.

One common solution to the I/O bottleneck is the disk array, where several disks provide data in parallel. This can provide much higher performance than a single disk, while still providing relatively inexpensive storage. By using a RAID (Redundant Array of Inexpensive Disks), disk arrays can be made reliable despite disk failures.

Even though disk arrays have the potential of supplying high-bandwidth data inexpensively, there are difficulties in making this data available to client machines across a network. For example, the RAID group at Berkeley built a prototype system called RAID-I, using a Sun-4/280 workstation connected to an array of 28 disks [CK91]. Unfortunately, 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.

To avoid the file server bottleneck, 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.

Another problem with a RAID disk array is that small random writes are very expensive due to parity computation, which is used for reliability. One solution is a log-structured file system (LFS) [Ros92], which writes everything to a sequential log so there are only large sequential writes. Thus, a log-structured file system can greatly improve performance of small writes.

This paper 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. Sawmill is designed to operate as a file server on a high-bandwidth network, providing file service to network clients.

Sawmill solves two problems: how to make use of a direct data path and how to combine a log-structured file system with a disk array. The key ideas in Sawmill are using streaming instead of caching, performing log layout on the fly, minimizing per-block overheads, and moving metadata over a separate control path.

The remainder of the paper is organized as follows. Section 2 gives some background on RAID storage, the RAID-II storage system, and log-structured file systems. Section 3 describes in more detail the implementation of the Sawmill file system and the techniques it uses to obtain high bandwidth. Section 4 contains performance measurements of Sawmill. Section 5 discusses related work, Section 6 discusses future work, and Section 7 concludes the paper.

2. Background

There are three main concepts combined in the Sawmill file system: the RAID disk array, hardware support for a fast data path, and the log-structured file system (LFS). This section gives background information on these ideas.

2.1 RAID

A RAID (Redundant Array of Inexpensive Disks) combines two ideas: parallelism and redundancy [PGK88]. A RAID uses multiple disks in parallel to provide much higher bandwidth than a single disk. A RAID can also perform multiple operations in parallel. Redundancy in the form of parity is used to maintain reliability. By storing parity, data can be fully recovered after a single disk failure.

In a RAID, data is striped across multiple disks. We use a RAID-5 architecture, which stripes data as blocks. With N disks, a group of N-1 data blocks is striped across N-1 disks. A parity block is computed by exclusive-oring these N-1 blocks and the parity block is stored on the remaining disk. Parity is distributed; successive parity blocks are stored on different disks to avoid the bottleneck of a dedicated parity disk. Each individual block is called a *stripe unit*, and a collection of N-1 data blocks and a parity block is called a *parity stripe*.

For peak efficiency, a full parity stripe should be written at once. In this case, the data and the parity can be written out in parallel. If only part of the parity stripe is modified, parity must be recomputed from the data already on disk. This overhead is especially costly for small writes. Figure 1 illustrates how parity is recomputed after a small write: the old data and parity must be read and then the new data and the recomputed parity are written, resulting in 4 disk operations. Thus, small writes are relatively expensive with a RAID, and writes of a parity stripe are most efficient.

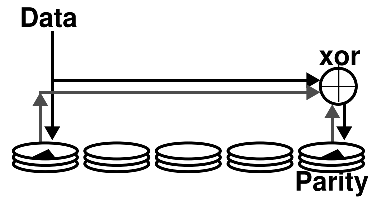

Figure 1. Read, modify, write. When a small part of the stripe is written to disk, the new parity is computed from the old data, the old parity, and the new data. Thus, four disk operations are required.

2.2 The RAID-II storage architecture

The Berkeley RAID project found that a high-bandwidth disk array could easily saturate the memory bandwidth of a typical workstation file server [CK91]. The RAID-I prototype used a Sun-4/280 workstation connected to an array of 28 disks. The bandwidth available through the system was very low, only 2.3 MB/s, mainly due to the low bandwidth of the Sun 4 file server's memory system and backplane.

To avoid the file server bottleneck, the follow-on RAID-II project built a storage system with a new hardware architecture designed to support disk array bandwidths. This hardware provides a high-bandwidth data path between the disks and the network, bypassing the file server. This results in separation of the control and data paths: 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 illustrates the hardware configuration of RAID-II. The controller provides a fast 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, and the disk interfaces. Although RAID-II was designed to support 40 MB/s, the maximum bandwidth is currently about 24 MB/s. Performance is currently limited by the disk controller and the VME link to the controller; each of the four disk controllers can handle about 6 MB/s. We have used a HIPPI network and an Ultranet to connect to clients. Further details of RAID-II are given in [DSH+94].

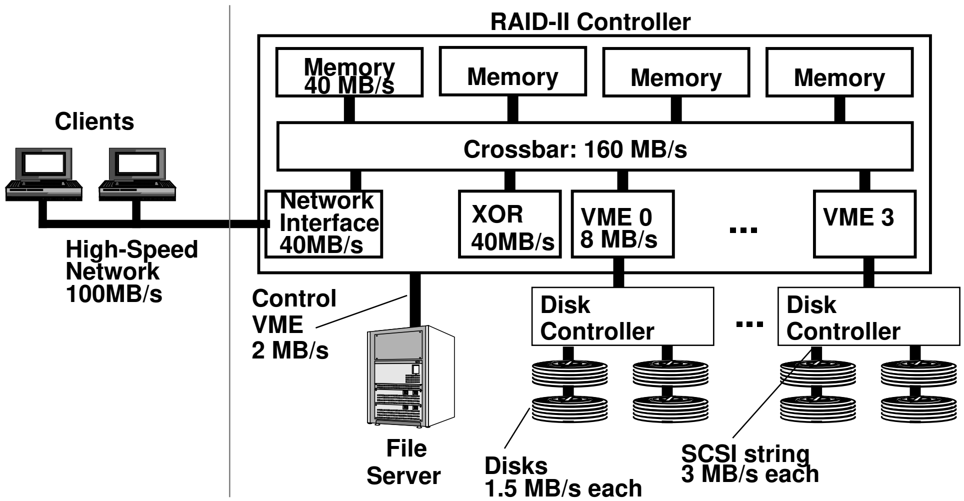

Figure 2. The RAID-II storage architecture. The RAID-II storage system (on the right) provides high-bandwidth file access to the network clients (on the left). The RAID-II controller provides a fast data path between the disks and the network. The controller is built around a fast crossbar that connects the disks, memory, and the network. The file server, running the Sawmill file system, controls the storage system but stays out of the data path. The current configuration uses 16 disks on four controllers. These disk are treated as a single RAID stripe group.

2.3 Log-structured file systems

The third idea used by the Sawmill file system is the log-structured file system (LFS). A log-structured file system [Ros92] writes data only to a sequential log. (A file block does not have a fixed position on disk, but instead its position changes every time the block is rewritten.) The log is written to disk in large units, called *segments*, on the order of 1 MB in length. As the log fills the disk, old segments are read in to free up the stale data; this process is called cleaning.

Because it writes only to the log, an LFS does not perform any random writes, but only large sequential log updates. This is an advantage with any disk system because it avoids the seek time and rotational latency of a random write. However, there is an even more significant advantage with a RAID due to the high parity overhead of small writes to RAID. By using an LFS, random writes can use the full sequential write bandwidth of the disk array because they are grouped together into a log write. The LFS segment size must be a multiple of the parity stripe size for maximum performance.

3. Implementation

This section describes the implementation of the Sawmill file system. Section 3.1 gives an overview of how Sawmill processes client requests. Section 3.2 describes how Sawmill uses the controller memory. Section 3.3 explains how read performance is improved. Section 3.4 describes the new log layout techniques. Section 3.5 describes the handling of metadata.

3.1 Handling a typical request

Clients communicate with Sawmill over a high-speed network. In the current implementation, client applications are linked with a small library to use the Sawmill file system. This library converts file system operations into socket operations. For instance, to open a file, the client calls a library routine, which opens a socket connection to the Sawmill file server. The client then sends the open parameters through the socket and receives the status from the file server. It would also be straightforward to provide access to data via NFS; however, performance would be limited by NFS protocol overhead.

For a read, the client library routine sends the read parameters over the socket and then receives the data. On the server side, the read request is received into controller memory and copied through the RAID-II controller into the file server. The Sawmill file system determines where the data is stored on disk. The file server sends commands to the RAID-II controller to read data from disk into controller memory. 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. Note that the file server only processes requests and handles control messages, while the RAID-II controller does the actual data movement.

Writes to Sawmill are handled in a similar fashion. The client sends the write parameters over the socket, followed by the data. The file server receives the write request, determines where in controller memory to receive the data, and informs the RAID-II controller to start receiving data. 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. Again, the data is transferred from network to disk without passing through the file server's memory.

Sawmill currently uses the same network to receive requests and to transfer data. However, it would be straightforward to use separate control and data networks. This would typically be used if the high-speed network had high latency, for instance, to set up a connection.

3.2 Controller memory

The RAID-II controller board has a 32 MB memory buffer. This memory is used by the RAID striping driver and the Sawmill file system. The RAID striping driver uses memory for buffering, 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. This section describes the use of controller memory for reads. Section 3.4 will describe the use of controller memory for layout of write data in the log.

In the original Sawmill file system design, we planned to implement a standard file system block cache in controller memory. However, we decided against this for two reasons. First, 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 with new blocks. Second, with RAID-II, the memory available for a cache would be about 20 MB; we expect this is 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. In any case, client caching could provide most of the potential benefits of a server data cache.

Currently, 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. As data blocks are read from disk to memory, previous blocks are being sent from memory to the network. We are currently adding prefetching to Sawmill to obtain this benefit for small operations.

3.3 Batching of reads

For efficiency, data should be read with a single disk operation whenever possible. Unfortunately, a sequential request might not be stored sequentially in the log. Fortunately, there will often be locality of writes, so the data may be stored in the same segment, even if the blocks are not stored sequentially. In this case, it would be more efficient to read the entire 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.

To group reads, the file system loops over each block, checking its position on disk and seeing whether it is part of the same segment. When it collects as many blocks as possible, it can read them all at once. 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.

3.4 On-the-fly 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. Sawmill uses a new method of performing log layout, called on-the-fly layout, to get more efficiency. In a standard LFS implementation [Ros92][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 contrast, Sawmill assigns a position in the log to each data block when the write request comes in. The file system informs the controller of this position, 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 such as inodes. Figure 3 illustrates the two techniques.

There are several advantages of doing layout when writes are received rather than as a backend process. First, layout for a large write can be done as a single operation, instead of many per-block operations, thereby minimizing the processing overhead. Second, blocks do not have to be moved to and from the cache, but are immediately put in the proper location. The RAID striping driver requires the segment written to disk to be stored contiguously in memory. Thus, a copy operation would be required, which would significantly reduces performance. (An alternative would be a device driver that accepts a scatter-

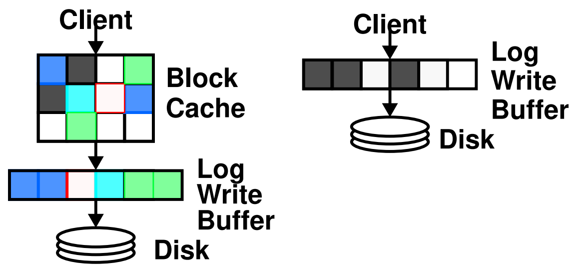

Figure 3. Comparison of log layout techniques. On the left, the layout uses a cache as in traditional log-structured file systems. On the right, the layout is done on the fly, directly into a write buffer.

gather array so layout could be done by moving pointers instead of 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.

Because it doesn't use a cache, on-the-fly layout has a few potential disadvantages. First, data cannot 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. However, if blocks are placed in the log immediately, they will be written in the random order in which they were received. This will decrease performance if they are later read sequentially. The second disadvantage occurs if data blocks are modified or deleted. With a cache, only the live blocks will be written to disk. However, with on-the-fly layout, the dead blocks will have a position in the log and will go to disk, resulting in unnecessary disk traffic. We expect that these disadvantages will not be important in practice. Because of the high data rates, blocks are likely to go to disk before they would be modified or deleted. Also, if there were a cache on the clients, the client cache could do the processing and provide the benefits.

3.5 Metadata movement

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. With a normal storage system architecture, the file system just accesses metadata as it needs it. However, with RAID-II any required metadata must be explicitly copied between file server memory and the RAID-II controller. Since moving data over this path is relatively slow, metadata is cached in file server memory. As a result, the Sawmill file system must maintain consistency among metadata in file server memory, metadata in controller memory, and metadata stored on disk.

4. Performance measurements

This section describes performance measurements of the Sawmill file system. Section 4.1 describes measurements of a single request stream with requests that are handled sequentially. Section 4.2 describes measurements of multiple requests processed in parallel: these numbers indicate the additional performance that a disk array can obtain by handling concurrent requests. Performance results are given for the raw disk array, the disk array configured as a RAID, and the disk array running Sawmill.

These performance measurements used RAID-II with 16 disks. The stripe unit was 64 KB, yielding a parity stripe size of 960 KB. In other words, writes of 960 KB transfer 64 KB data blocks to 15 disks and a 64 KB parity block to 1 disk in parallel. The LFS segment size was also 960 KB. The measurements in this section are of random requests of various fixed sizes. For these measurements, the Sawmill file server was a Sun-4/280, which is relatively slow (about 9 SPEC89 integer SPECmarks).

Because we don't have a client that can handle the data rates of RAID-II yet, the following performance tests did not transmit data across the network, but only transferred the data between the disks and controller memory. Our current client can only transfer 3 to 4 MB/s over the network. However, the measured server CPU load to handle this network traffic was 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 performance numbers in this section should be a reasonable indication of the actual network performance with a fast client.

There are several key results from the performance 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 performance close to that of the raw disk. Because of LFS, Sawmill can handle small writes an order of magnitude faster than the raw disk. However, the server CPU is a performance bottleneck for small writes and for multiple request streams. Small write performance with Sawmill would improve dramatically with a faster CPU.

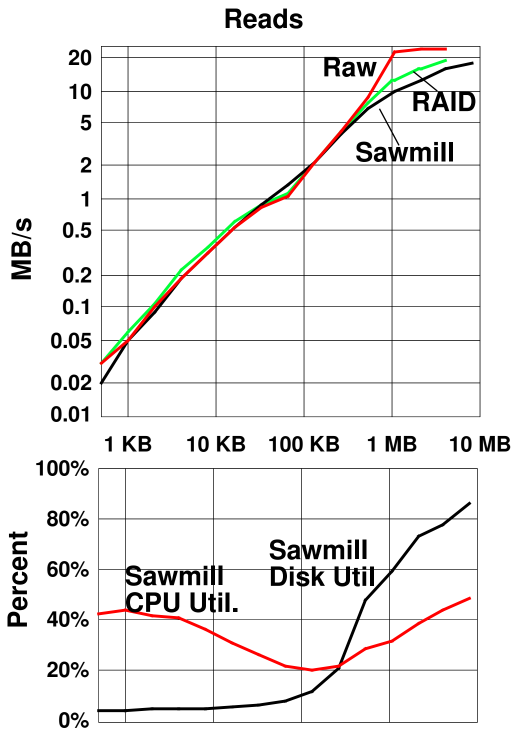

Figure 4. Read performance for a single stream of random requests as a function of the request size. The top graph shows bandwidth through the raw disk array, the disk array treated as a RAID, and the Sawmill file system. The lower graph shows CPU utilization and disk utilization through Sawmill.

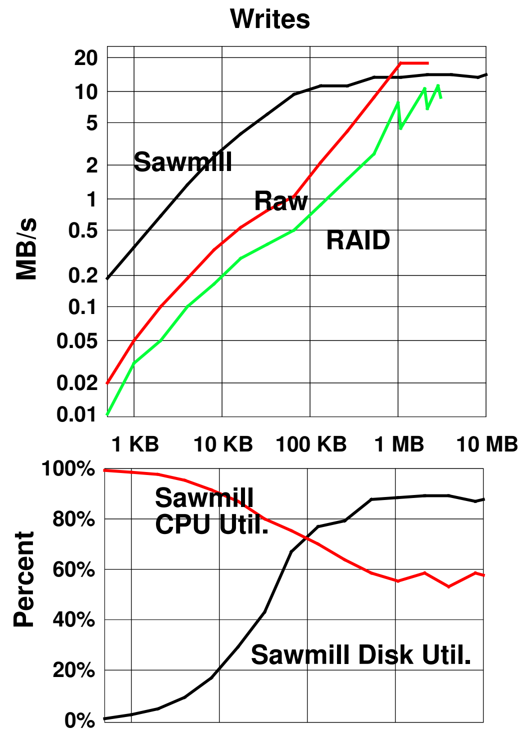

Figure 5. Write performance for a single request stream. As in Figure 4, the top graph shows performance of the raw disk array, the RAID, and Sawmill. The lower graph shows CPU and disk utilization with Sawmill. Note that because of LFS, Sawmill performance is much better than writing directly to the disks. Also note that small requests are totally CPU limited. This is due to LFS: for small writes, almost all the time is spent grouping them together. CPU utilization remains high, even for large writes.

Also, Sawmill can only support about 3 or 4 concurrent streams of small requests before the CPU becomes a bottleneck.

4.1 Single request stream

This section describes the performance of the raw disk array, the disk array treated as a RAID, and the Sawmill file system, when receiving a single stream of requests. This indicates the performance available to a single application. Figure 4 shows read requests and Figure 5 shows write requests.

4.1.1 Raw disk array performance

To understand the performance, the first item to examine is the disk array. The array contains IBM 0661 disks [IBM91]. These disks have average rotational latency of 7 ms and average seek time of 12 ms. Each disk can read and write at about 1.6 MB/s. The disk array was configured with 16 disks on four controllers. A single SCSI string was able to read at 3.14 MB/s with two disks. With two strings of two disks per controller, each controller can provide 6.24 MB/s. For the measurements in this section, the disk array was treated as independent disks, with no parity computation. To make the measurements comparable to the RAID and Sawmill measurements, requests were broken into 64 KB blocks and made to the appropriate number of disks: requests smaller than 64 KB used a single disk and larger requests used multiple disks.

The “Raw” lines in Figures 4 and 5 show the performance of the raw disk array as a function of request size. Performance was highly dependent on request size. Peak read performance was 24.5 MB/s and peak write performance was 18.4 MB/s. For small requests, seek time dominated.

With the raw disk array, the processing overhead is minimal, so the performance is almost entirely dependent on the disk speed. The time to complete a request to a disk can be modeled approximately as:

20 ms + disk_request_size / peak_bandwidth

where the measured peak bandwidth is 1.57 MB/s for reads and 1.16 MB/s for writes. By dividing the request size by this time, the bandwidth for a given request size can be modeled. Writes are slower because of missed revolutions; the disks have a track buffer that minimizes missed revolution costs for reads.

4.1.2 RAID performance

In the previous section, the disk array was viewed as independent disks. In this section, the disks are 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.

The "RAID" lines in Figures 4 and 5 show the read and write performance of a single stream to the RAID device. Read performance is generally the same as for the raw disk. For large accesses there is a small performance loss, mainly from processing overhead and increased latency due to the RAID striping driver. RAID writes take at least twice as long as writes to the raw disk due to the read-modify-write parity update. Note the spikes in write performance at multiples of 960 KB; these sizes are multiples of the parity stripe size and thus don't require a parity read-modify-write.

These measurements show that for good write performance, the file system must perform operations in multiples of the parity stripe size.

4.1.3 Sawmill performance

The Sawmill file system, by using the techniques described earlier, was able to obtain peak bandwidth of about 80% of the raw disk bandwidth. For the measurements in this section, random reads and writes of various sizes were made to a single 30 MB file.

For a single request stream, Sawmill read performance is close to the raw disk performance. There is a 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 performance is 21 MB/s, slightly below the raw disk bandwidth. For individual small random reads, as shown in Figure 4, 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.

Figure 4 shows the CPU and disk usage for Sawmill read requests. The disk usage shows the percentage of time the disks are in use, averaged across all 16 disks (e.g. 8 disks in use 50% of the time is 25% utilization). Because the measurements in Figure 4 use only a single stream of requests, small requests only use one disk at a time and cannot exceed 6.25% utilization. Note that CPU load is fairly high in Figure 4, but drops 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.

Figure 5 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 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 cost of writes that modify less than a file block; in this case an additional read would be required to fetch the unmodified data.

Figure 5 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, performance is about 20 times that of the RAID. This performance would be much better 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 performance of 2 MB/s, about 200 times the performance of writing directly to the RAID.

Figure 5 can be used to estimate the write performance of a file system based on the Unix FFS [MJLF84] running on RAID-II and compare it with Sawmill. If writes go to disk individually, the Unix file system would have performance similar to that of the RAID (assuming no CPU overhead for the Unix file system and ignoring the seeks to write metadata in the Unix file system), so Sawmill would be a factor of 20 faster for small writes. A Unix file system with clustering [MK91] would shift performance along the RAID line, since the writes would be in larger units.

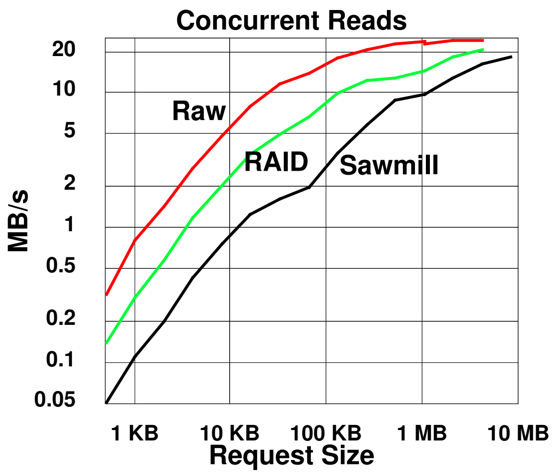

Figure 6. Read performance with multiple concurrent requests. The lines indicate the raw disk array as 16 independent disks, the disk array treated as a RAID, and the Sawmill file system. Concurrency of RAID and Sawmill are limited by the CPU load.

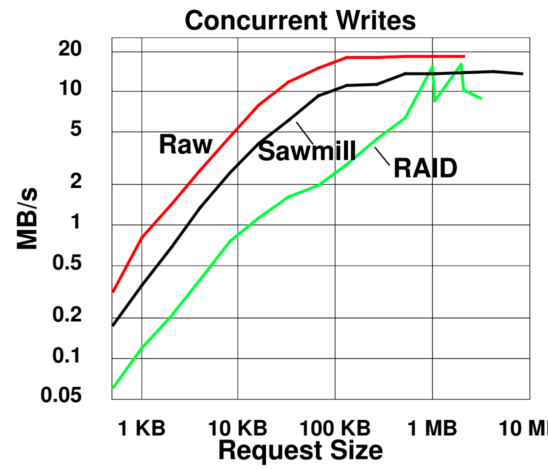

Figure 7. Write performance of the raw disk array, RAID, and Sawmill with multiple concurrent requests. The spikes in RAID write performance at 960 KB and 1.92 MB illustrate the write performance possible when a full stripe is written at once; other writes require an expensive parity update. Note that because of LFS, the performance of Sawmill exceeds performance of the RAID.

However, Sawmill performance would still be much better because Sawmill writes are grouped into full parity stripes. 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 (which will be discussed in Section 6.1), but the performance benefit of logging is likely to greatly exceed this cost.

4.2 Multiple stream performance

This section describes the performance of the system when multiple requests are processed in parallel. This indicates the performance if, for example, multiple applications or multiple clients were using the system. Figure 6 shows read performance with concurrency and Figure 7 shows write performance.

A disk array can process multiple requests in parallel if they go to separate disks. This improves the overall system throughput and the total number of operations per second, but doesn't improve the performance of any particular request stream. Concurrency is a significant benefit for small requests, which only use a single disk. Potentially, 16 small reads or 8 small RAID writes could take place in parallel, improving performance by the same factor. Requests larger than the 64K stripe unit will be striped across multiple disks and become inherently parallel. Thus, multiple requests improve the performance much more for small requests than large requests.

The limiting factor to concurrency in our system is the CPU load. Unfortunately, our 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 better processor.

In Figures 6 and 7, the degree of concurrency used depended on how much concurrency the system could handle before saturating. 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.

For the raw disk array, the CPU load is almost entirely determined by the number of disk operations. Starting a disk operation, handling the interrupt, and concluding the disk operation take about 1 ms in total. Since the SCSI controller can only handle requests up to 128KB, a larger logical operation results in multiple disk operations. Due to these factors, CPU 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.

Besides bandwidth, another important measurement is the total number of I/O operations the disk array can support per second. Figure 8 shows the

Type	Read ops per sec	Write ops per sec
1 disk	50	45
16 disk array	710	670
RAID: 1 request	55	25
RAID: 8 requests	300	100
Sawmill: 1 req.	45	345
Sawmill: 4 reqs.	95	345

Figure 8. Random 4KB I/O operations. This table shows the number of I/O operations RAID-II can handle per second under various configurations. CPU load limits multiple RAID and Sawmill requests.

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.

In theory, by performing multiple operations in parallel, performance should scale with the number of disks. However, the main limitation on parallelism with RAID-II is the CPU load; handling multiple requests through RAID or Sawmill saturates the CPU before all disks are in use. The RAID striping driver has more processing load than the raw disk, and the Sawmill file system has additional load. Thus, the number of parallel requests possible before the CPU saturates is reduced. The RAID measurements were saturated with 8 requests in parallel, while the Sawmill measurements were saturated with 4 requests.

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, Figure 8 shows about a 5% performance penalty; the CPU load is about 75%, so there is some delay in starting requests.

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 Figure 8 is slightly better for the RAID because seek distance was shorter for the striped requests.) Performance of a single small write stream is 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. Performance levels off around 8 request streams because the CPU becomes saturated. A faster CPU would significantly increase the total number of I/Os per second that the RAID can support.

Figure 8 also shows performance of the Sawmill file system for small requests. Comparing the read measurements to the RAID with a single request stream shows that there is a slight performance loss due to Sawmill; this is due to overhead in the file system. (The read latency is simply the reciprocal of the I/Os per second, or about 22 ms.) Due to the CPU load, reads obtained little benefit from concurrency. For writes, the number of I/Os per second is very good compared to the RAID, showing the benefit of LFS. With a faster CPU, the number of I/Os per second would be even higher, since performance is CPU limited, not disk limited.

4.3 Scalability

This section describes how the performance measurements are likely to change with improvements in technology. To summarize, performance of large requests, small writes, and concurrent operations is likely to improve with faster technology. However, small reads are limited by the performance of individual disk drives, which will not improve as quickly over time.

Because small writes are limited by CPU speed, small write performance 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 over what we measured.

Small individual random reads are limited by disk positioning time. Disk positioning times are 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 number of I/O operations per second, but won't help the performance of an individual request. Prefetching can be used to improve the performance of individual requests; if the data is fetched before it is required, the disk latency will not affect latency to handle 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.

Large reads and writes already achieve close to the raw system performance with Sawmill. Thus, large request performance will only increase 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, note that the current server is a Sun-4 and much faster CPUs are readily available.

5. Related work

There are several storage systems that provide high-performance I/O. One fundamental problem these systems must solve is how to provide the data without being limited by the file server's CPU, memory, and I/O bandwidth. There are various methods that have been used to solve these problems. For instance, mass storage systems usually use an expensive mainframe, designed to support high bandwidth, as the file server. Other systems use multiprocessors or multiple servers to provide sufficient server power.

5.1 Mass storage systems

There are several high-performance mass storage systems in use at supercomputing centers. One example is the LINCS storage system at Lawrence Livermore National Laboratory [HCF+90]; this storage system has 200 GB of disk connected to an Amdahl 5868. Another example is a system at the Los Alamos National Laboratory [CMS90], which has an IBM 3090 running the Los Alamos Common File System.

These systems solve the problem of the file server being a performance bottleneck by using a mainframe as the file server. The mainframe server is designed to have the I/O bandwidth, memory bandwidth, and CPU power necessary to provide very high data rates to clients. In addition, the server is connected to a network sufficient to handle very high data rates. The 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.

5.2 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 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, Ethernet, and a single 55 MB/s VME bus; measurements show it can supply about 400 KB/s to a single client and can saturate an Ethernet network with 1MB/s per Ethernet connection [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 would be connected to a high-performance interconnection network.

By providing multiple processors, these systems avoid bottlenecks from limited server CPU bandwidths. However, this tends to be an expensive solution, since it requires buying enough processors to provide the necessary memory bandwidth. Even so, the Auspex server still has a memory bandwidth bottleneck since it uses a single VME bus.

5.3 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 memory system of a single server. 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.

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. Zebra and Sawmill both combine LFS and RAID. The key difference is that Zebra uses multiple servers with single disks and Sawmill uses a single server with multiple disks. Swift and Zebra avoid the file server bottleneck by using multiple servers, while Sawmill avoids the bottleneck by using a data path in hardware. There is a cost trade-off: Sawmill requires a special controller, while Swift and Zebra require multiple fast servers.

5.4 RAID parity updates

There are several techniques to reduce the cost of updating parity after a partial stripe write. One technique is Parity Logging [SGH93]. In this technique, parity updates are written to a log. At regular intervals, the log is scanned and the parity modifications are applied to the standard RAID parity blocks. Floating Parity [MRK93] is a second technique for minimizing parity cost. In this technique, multiple parity blocks are reserved on disk. Updates can use the block closest to the current rotational position of the disk. The Logical Disk [dJKH93] implements a log-structured file system at the disk level rather than the file system level by writing all blocks sequentially to a log and maintaining the mapping between logical disk blocks and physical disk blocks. Since it writes blocks to a sequential log, it avoids random parity updates.

6. Future Work

Probably the most important unanswered issue with Sawmill is the cost of cleaning a log-structured file system. Another major issue is how to improve the performance of small reads; this was discussed earlier. A related issue is improving large read performance by data reorganization.

6.1 Cleaning cost

One of the key unanswered questions about Sawmill is the overhead due to log cleaning. Cleaning is the process of garbage-collecting the log to free up space. Cleaning is not yet operational in Sawmill, so performance measurements are not available. Previous work [Ros92] indicated that overall cleaning costs would be low. However, [SBMS93] 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 former work did not have dynamic performance measurements of the cleaner, and the latter measurements were on a largely untuned implementation.

The impact of cleaning on the performance of log-structured file systems continues to be a topic requiring additional investigation, which is beyond the scope of this paper. At this point, there is no evidence to suggest that Sawmill will not perform well for the workloads typical of office environments. Furthermore, even if cleaning costs are high, Sawmill's ability to avoid the "small-write problem" of RAID devices could well offset such overheads.

6.2 Data reorganization

Log-structured file systems are write-optimized; that is, they organize data for fast writes, even though this may disadvantage later reads. In particular, 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. Conclusions

This paper has described Sawmill, a high bandwidth file system that uses the RAID-II disk array. Sawmill uses a cost-effective file server to provide high-bandwidth access to a disk array. By taking advantage of hardware support, Sawmill provides data rates much higher than the memory bandwidth of the file server.

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, while running on a Sun-4. The log-structured file system improved performance of a small write stream by an order of magnitude over the RAID, and with a faster processor small write performance would be improved even more. The high CPU load in many of the performance 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.

In conclusion, Sawmill shows that combining a direct data path, a disk array, and a log-structured file system is a cost-effective method of providing high-bandwidth storage.

8. Acknowledgments

The authors would like to thank John Bongiovanni, Keith Bostic, and Drew Roselli for their comments on this paper. The assistance of the RAID group, in particular Ann Drapeau and Srini Seshan was instrumental in getting Sawmill running. This research was supported by the California MICRO program, ARPA grant N00600-93-C-2481, and NSF grant IRI-9116860.

9. References

[CK91] Ann L. Chervenak and Randy 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*, June 1991.

[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] Wiebran de Jonge, M. Frans Kaashoek, and Wilson 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] Ann L. Drapeau, Ken Shirriff, John H. Hartman, Ethan L. Miller, Srinivasan Seshan, Randy H. Katz, Ken Lutz, David A. Patterson, Edward K. Lee, Peter M. Chen, and Garth A. Gibson. RAID-II: A high-bandwidth network file server. In *Proceedings of the 21st International Symposium on Computer Architecture*, Chicago, IL, April 1994. To appear.

[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.

[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.

[HO93] John H. Hartman and John 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.

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

[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*, 1991.

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

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

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

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

[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.

[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.

[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.

Author Information

Ken Shirriff is a graduate student in computer science at the University of California, Berkeley. He is a member of the Sprite network operating system project. His interests include distributed operating systems, computer graphics, and fractals. He received a B.Math. degree from the University of Waterloo in 1987, a M.S. in computer science from UC Berkeley in 1990, and expects to receive his Ph.D. in 1994. He can be reached at `shirriff@cs.berkeley.edu`.

John K. Ousterhout is a Professor in the Department of Electrical Engineering and Computer Sciences at the University of California at Berkeley. His interests include operating systems, distributed systems, user interfaces, and computer-aided design. He is currently leading the development of Sprite, a network operating system for high-performance workstations, and Tcl/Tk, a programming system for graphical user interfaces. Ousterhout is a recipient of the ACM Grace Murray Hopper Award, the National Science Foundation Presidential Young Investigator Award, the National Academy of Sciences Award for Initiatives in Research, the IEEE Browder J. Thompson Award, and the UCB Distinguished Teaching Award. He received a B.S. degree in Physics from Yale University in 1975 and a Ph.D. in Computer Science from Carnegie Mellon University in 1980.