MDLFS: Exploiting Information to Improve Performance

CS736 Project

Timothy E. Denehy
Nilofer F. Motiwala
{tedenehy, motiwal}@cs.wisc.edu

December 18, 2000

Abstract

In this work, we address the separation between file systems and storage systems. A file system that is aware of its underlying storage structure may be able to take advantage of its extra knowledge. MDLFS is a disk aware file system that tracks the performance of its storage components. This dynamic information is used to make informed decisions about the allocation of data to each of several disks. The performance of MDLFS using a variety of placement strategies is compared to FFS and LFS on top of a RAID storage system. Our test results show that discriminating among disks using performance information can lead to better throughput.

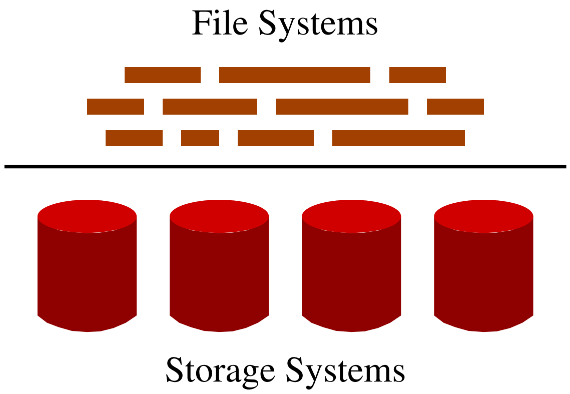

Figure 1: Existing System View

1 Problem and Motivation

There currently exists a separation between file systems and their underlying storage systems. This situation is depicted in Figure 1. The advantage of this separated structure is its simplicity. The storage system can provide a basic interface and implement its operations in a variety of different ways. The file system can make use of this basic interface and rely on the storage system to service its requests.

The problem with this view, however, is that it leaves file systems with incomplete knowledge about the storage systems on which they rely. This forces file systems to make assumptions that may or may not be true. For instance, most file systems assume that they are operating on top of a single disk. Many modern storage systems, most notably RAID systems [8], incorporate several disks. File systems also tend to assume that their disk provides consistent performance from all of its regions. Modern disks, however, are often implemented using multiple zones. The longer, outer tracks of the disk contain more sectors than the shorter, inner tracks. Since the disk maintains a constant rate of rotation, a higher transfer rate is achieved from the outer tracks [6]. There can also be some variance in performance between individual disks in storage systems with multiple devices.

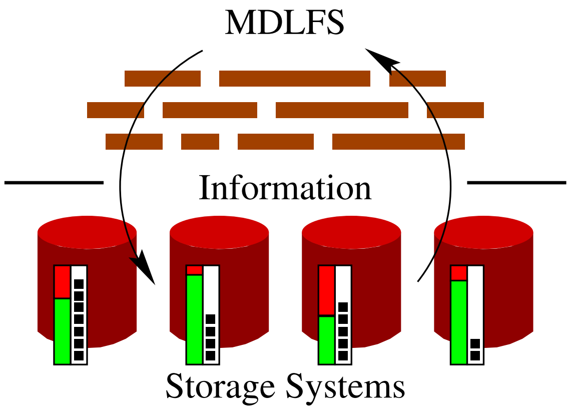

Figure 2: MDLFS System View

2 Approach

Given the existing separation between file systems and storage systems, and the discussion of how this gap can lead to erroneous assumptions on behalf of the file system, let us talk about our approach to solve this problem.

As shown in Figure 2, we decided to bridge the existing gap with an information channel between the file system and storage system. Thus, the file system is now aware of the multiple underlying storage devices. Further, as mentioned earlier, not all storage devices are equivalent in performance. In order to distinguish among devices based on their performance we decided to monitor performance statistics through the information channel. Our embodiment of this approach is a multiple disk log-structured file system (MDLFS).

Our hypothesis is that having knowledge about multiple disks and the ability to differentiate among them based on their attributes will improve the overall performance of the file system.

3 Methodology

To test our hypothesis, we decided to create and test a disk aware file system (MDLFS). For purposes of this project, we extended an existing implementation of a log-structured file system [9]. A log-structured file system attempts to improve write performance by writing all data and metadata in large segments. The disk is viewed as an infinite log, and the new segments are appended to the end. We chose to use LFS for two main reasons. First, LFS was designed with write intensive workloads in mind. One of our sub-goals for this project was to see how much write throughput could be improved by making the file system disk aware. Further, the task of placing the next segment offers a well defined decision point for selecting which disk to use given the current state of all disks.

4 Implementation

MDLFS was implemented by extending the NetBSD version of LFS, which is based on the original version of LFS for BSD Unix [11]. The LFS source code was replicated and modified to create a separate and distinct MDLFS file system within the kernel. This allowed both LFS and MDLFS to exist within the same system, which was beneficial for verifying correct operation and later testing the two file systems.

4.1 Accessing Multiple Disks

To integrate more than one disk, the majority of the file system code was presented with a logical address space that spans all of the disks in the system. The system compensates for the multiple, separate devices at a lower level. This approach was least intrusive to the existing file system source code, and it allowed for a less cumbersome implementation.

Several changes were made to allow the file system to access multiple disks. First, the superblock was augmented with the necessary attributes for each disk in the file system. The values added included the device names and the first and last logical blocks that reside on each disk. This change forced corollary modifications to newfs and the file system mount routines. The newfs program is responsible for the initial layout of the file system on disk. It was modified to accept multiple device parameters that are used to create the logical address space. Each disk is assigned a portion of this address space, and the details are stored in the initial superblock. The MDLFS mount routines use these details to prepare the file system at mount time. The device names are used to find an associated vnode. These vnodes are inspected for previous mounts, and if available they are marked as mounted for the new instance of the file system. These mount routines also initialize some MDLFS accounting information.

The BSD versions of Unix use the vfs and vnode layers to interact with their file systems [10] [15]. The vfs layer defines a set of operations that are related to an entire file system, and the vnode layer consists of a set of operations for specific files within a file system. These versions of Unix also use buffers to coordinate input and output operations. These buffers contain fields to define the specifics of the operations and encapsulate the associated data.

Within the vnode layer, the vop_strategy() method controls the file system's access to disk operations. A new mdlfs_strategy() method was implemented to intercept these operations and make the necessary modifications to access multiple disks. This routine takes the buffer in question and remaps its logical block number to its corresponding physical block number on a specific device. This has the effect of redirecting the input or output operation to the correct disk in the multiple disk system.

4.2 Monitoring Disk Performance

Several mechanisms were added to MDLFS to allow it to track the performance of each disk. In order to time the length of disk operations, a timing structure was added to the buffers discussed previously. When a write operation is started in mdlfs_strategy(), the current time is determined and recorded in the corresponding buffer. Upon completion of the operation, the input and output subsystem invokes the mdlfs_callback() routine. This method consults the current time to determine the total time for the operation on the buffer. This total time is divided by the number of kilobytes accessed in the operation, and the resulting throughput is stored to represent the disk's current performance. The fact that the mdlfs_strategy() and mdlfs_callback() routines bound each operation is also used to track the number of outstanding write requests on each device. This MDLFS implementation only monitored disk performance during write operations, but it may also be beneficial to incorporate similar monitoring for read operations.

5 Making Informed Decisions

MDLFS uses the dynamic performance and state information it records for each disk to decide where to place the segment that will be written next. Several different strategies for this choice were implemented and tested:

1. Choose the disk with the highest throughput (Write Time)

2. Choose the disk with the least number of outstanding requests (Write Requests)

3. Choose the disk with the smallest workload (Workload)

Each disk's workload is defined as the product of its throughput and its number of outstanding requests.

One drawback of this MDLFS implementation is the small time lapse between choosing the location of the next segment and actually writing to that segment. In order to allow for easier cleaning and recovery, the segments are chained together in the order they are written. To provide this functionality, the location of segment $n + 1$ is chosen just before the first partial of segment n is written. When segment $n + 1$ is written, the performance data that was used to choose its location is somewhat stale, but we believe the data is still representative of the disk's performance. This problem could be alleviated by introducing a logical segment to physical segment mapping [16], but this would require significant changes to the rest of the file system.

6 Performance Measurements

Testing of our MDLFS implementation was done on a machine with two disks, a SCSI (Quantum XP34300) and an IDE (Western Digital Caviar WD100EB). Under our configuration, there was a wide variance in the performance of the two disks; the IDE was much faster than the SCSI. The degree of this variance is reflected in our preliminary results.

We used the Iozone [3] benchmark for measuring the performance of our implementation and to compare against existing file systems (FFS and LFS). All benchmarking was done with the following parameters: a file size of 100 MB and a record length of 16 KB. We initially experimented with a record length of 4 KB. This led to some uncharacteristic results in the case of random writes for FFS and LFS. Because the record length (4 KB) was smaller than the block size (8 KB), the blocks had to be read before they could be written, and this caused FFS to perform significantly better. Thus, to avoid this situation, we increased the record length to 16 KB.

The operations that Iozone performed were: sequential write, sequential re-write, sequential read, sequential re-read, random write, and random read. In the discussion below we will compare how FFS, LFS, and MDLFS performed on these operations.

All tests were run without the cleaner, and the results presented are the averages over three trials. The RAID tests were run using a software RAID (NetBSD raidframe).

6.1 Single Disk Performance

The first set of tests involved running the Iozone benchmark for each of the file systems on a single disk (on both IDE and SCSI). The Write Time metric was used for testing MDLFS (note, however, that the choice of metric is of no significance since there is only one disk). In this case, all of the file systems had a consistent and correct view of the underlying storage system. The reason for conducting this comparison is to quantify the overhead that MDLFS incurs in decision making. Figure 3 shows the results for these tests.

We expected MDLFS to perform as well as LFS,

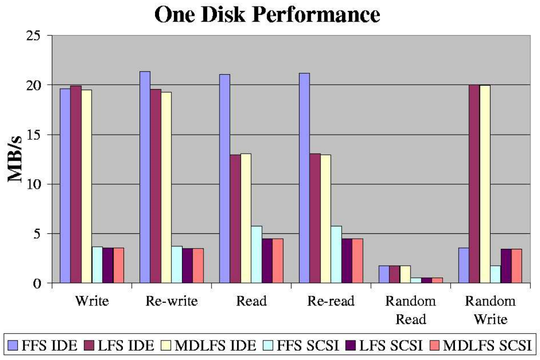

Figure 3: One Disk Performance

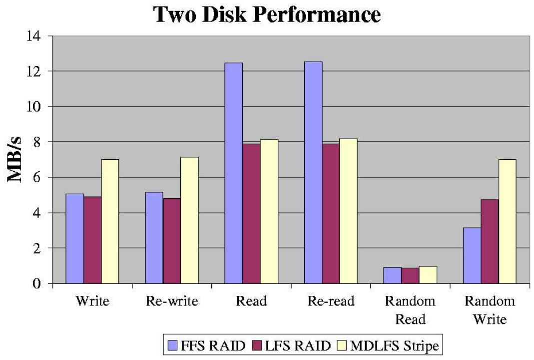

Figure 4: Striping Performance

and in both cases our expectations were met. There is a slight degradation in the case of MDLFS writes as compared to LFS. This is attributed to the overhead involved in maintaining timing information on all write buffers and other bookkeeping information regarding performance.

The superiority of FFS in read performance as compared to LFS is explained by the fact that the writes were followed by re-writes. Thus, the LFS data layout was not as sequential as that of FFS at the end of the re-write phase of the test. In the case of a write followed immediately by a read, we expect LFS to perform equivalently to FFS.

The other important aspect highlighted by this graph is the wide variance between the two disks. The IDE disk has a write throughput of 20 MB/sec, whereas the SCSI disk has a write throughput of only 3 MB/sec. This significant disparity plays an important role in explaining the performance gain of MDLFS over FFS and LFS in subsequent tests.

6.2 Striping Performance

The next test was conducted to see how the MDLFS striping strategy compared to RAID striping. FFS and LFS were run on RAID level 0 (only striping, no redundancy, no parity). To test on equal grounds, RAID was configured with a stripe size of 1 MB to correspond to the stripe unit of MDLFS (in MDLFS, the size of the segment was 1 MB).

Since the comparison was on equal grounds and we were not employing any performance knowledge in the decision making process, we expected MDLFS to do as well as LFS. However, as can be seen in Figure 4, MDLFS performed better. Our hypothesis to explain this behavior is that in the case of FFS and LFS, there is some overhead in transferring control to RAID. In MDLFS, the multiple device information is integrated into the core of the system and hence it incurs less overhead in making placement decisions.

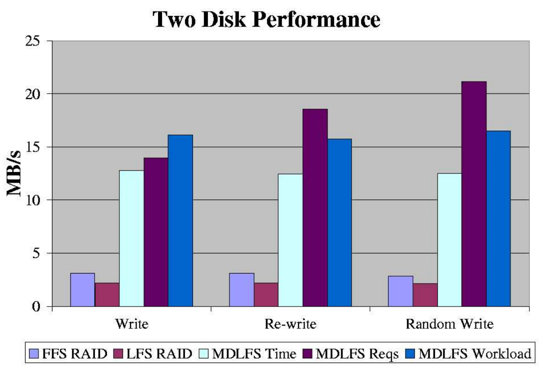

Figure 5: Multiple Disk Write Performance

6.3 Multiple Disk Performance

Figure 5 offers a comparison among the various MDLFS decision making strategies as well as FFS and LFS. The different strategies were outlined in section 5. The amount of data allocated to each disk by the various strategies is shown in Table 1. All three MDLFS strategies perform significantly better than FFS and LFS. (FFS and LFS were run on RAID level 0 at a stripe size of 32 blocks). From the data in Table 1, it is evident that MDLFS is biased towards the faster disk (IDE). Since most of the data is written out to the faster disk, MDLFS performs much better than FFS and LFS on RAID, which equally distributes the data between the two disks.

In the write case, the Workload strategy does the best, perhaps since its utilization of the two disks is a compromise between the other strategies. However, as the tests progress into the re-write phase, the number of requests strategy performs better. This may be due to the high number of outstanding requests on the SCSI disk after the write phase of the testing, thereby causing the IDE to be picked more often in the re-write phase. The spikes in the throughput delivered by the number of requests strategy may also be attributed to the smaller amount of data that gets written in the re-write and random write phases.

Table 1: Data Distribution for Various Strategies

Strategy	SCSI	IDE
Write Time	22 %	78 %
Write Requests	9 %	91 %
Workload	15 %	85 %

The MDLFS Write Time strategy delivers the lowest throughput among the three strategies. This is partly due to a forced reset of a device's write time. This reset was enforced after a disk had not been chosen for ten consecutive intervals in order to ensure that one slow write did not impair the disk indefinitely. If the threshold was set higher, we suspect that the Write Time strategy would perform comparably to the number of requests strategy.

The write throughput of FFS and LFS is limited by the throughput of the SCSI disk since the data is sequentially laid out across the two disks.

Figure 6 depicts the read throughput achieved on the Iozone tests. Iozone tested operations in the following order: write, re-write, read, re-read, random read and random write. Hence, the reads were affected by the preceding write and re-write operations. The data placement achieved by using the number of requests strategy delivers the highest throughput for sequential reads. Once again, the data distribution of Table 1 explains the behavior we see here. Since the number of requests strategy lays out most of the data on the IDE disk, when doing a read, we are not slowed down by the throughput of the SCSI disk. The other two strategies layout more

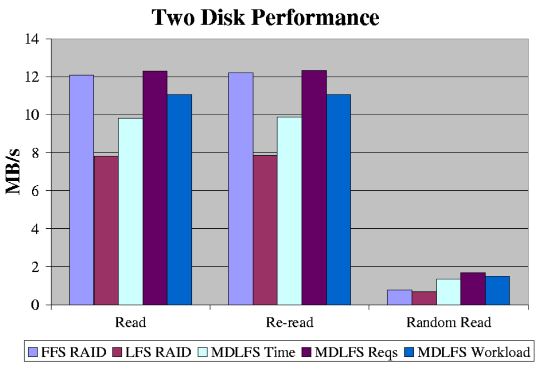

Figure 6: Multiple Disk Read Performance

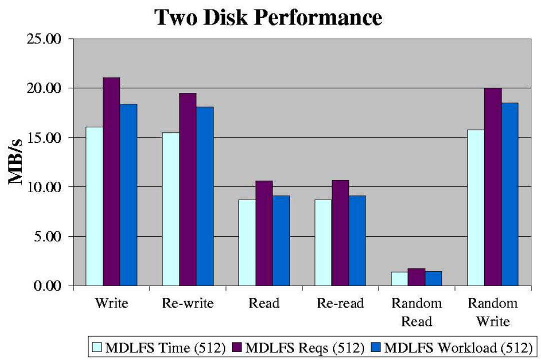

Figure 7: Multiple Disk Performance, 512 KB Segments

data on the SCSI disk and hence are affected more by the read throughput of MDLFS on the SCSI disk (4 MB/sec). FFS and LFS are also adversely affected because of the equivalent striping done by RAID. The FFS and LFS rates of 12 MB/sec and 8 MB/sec, respectively, are explained by their 6 MB/sec and 4 MB/sec throughput on the SCSI disk. Since the data is being read sequentially, the slower disk dictates the throughput.

A comparison of Figures 3 and 5 reveals an unexpected result for the write throughput of MDLFS. In Figure 5, one would expect an increase in write throughput due to the parallelization achieved in utilizing multiple disks. Hence, a reasonable write throughput would be around 23 MB/sec (adding the throughput of both disks). However, we see a decrease in throughput (12-16 MB/sec). This could be attributed to the large segment size, which may make it difficult to exploit the paraellelization among disks. To test this parameter, we ran our benchmarks with a reduced segment size of 512 KB. As can be seen in Figure 7, the write throughput improves to a maximum of 21 MB/sec in these modified tests. These results suggest that selecting a proper segment size is important for good performance.

The tests above confirm our hypotheses that having an information channel between the file system and storage system can lead to better data placement and improved performance. We must point out that these are preliminary tests and were tested on only one hardware configuration. More concrete conclusions could be made after more rigorous testing.

7 Related Work

Some recent work [5] has explored the use of adaptive methods to improve the performance of log-structured file systems. The goal of this research is to design a self-tuning system that monitors hardware and workload characteristics and adjusts the behavior of the system accordingly. The authors examine write performance by varying the segment size, using adaptive cleaning methods that choose between cleaning entire segments and simply plugging existing holes, and using cached data to reduce the cost of writes. They also explore read performance by using the existing cleaning mechanisms to reorganize data for more efficient reads. This reorganization is based on dynamic tracking of file access patterns. The use of these adaptive methods helps to alleviate some of the performance problems previously associated with log-structured file systems.

Spiralog [4] [16] is a remote, log-structured file system. Its goal is to provide a high-performance, highly available file system with an integrated backup mechanism. Spiralog combines a log-structured on-disk format with B-tree technology and write-ahead logging to provide stability and recoverability. It uses a mapping layer between the file address space and the log address space, as well as a translation between logical segments and physical segments on disk. The Spiralog cleaner uses a set of heuristic policies based on a finite state representation of segments. This approach allows the cleaner to estimate the utility of cleaning a certain segment, and then base its final decision on the current policy.

Another approach to improving performance is striping data across multiple disks [1]. Chen et al have explored maximizing performance in a striped array. They examine rules to determine an optimal striping unit based on a given range of workloads, request sizes, and hardware parameters.

Zebra [2] is a network file system that provides high availability and high performance. The file system data is striped across its servers, thereby increasing the file access bandwidth. Striping in Zebra is log based; data written by each client is gathered in a log which is then striped across the servers.

The adaptive disk striping method [13] aims at improving performance by applying a fuzzy logic rule base for striping data in an adaptive and redundant manner. The method involves dynamically changing the stripe unit based on the load level, the level of concurrency desired within and across applications, and on file access patterns.

MDLFS extends the log-structured file system design by adding the ability to adapt to multiple devices with varying performance characteristics. MDLFS also attempts to improve throughput by exploiting parallel I/O. Unlike these previous systems, however, MDLFS discriminates among the disks that make up its underlying storage. It uses performance information associated with each disk to make better placement decisions. Disks with better performance are given preference to improve the overall throughput of the system.

8 Future Work

Several enhancements could be explored to improve MDLFS in the future. Currently, all of our decisions are based only on write operations. It would be beneficial to also track read operations, thereby providing a more complete picture of the current workload on each disk.

This implementation of MDLFS did not perform any cleaning. The cleaner could be enhanced to provide some load balancing between the disks. Also, by using information about file access patterns, the cleaner could reorganize data in a more strategic manner.

Finally, data could be replicated across the disks to further exploit parallelism. For example, replicated data could provide the opportunity to direct a read request to a disk with a smaller workload.

9 Conclusions

Existing file systems have little knowledge of their underlying storage structure. This simplified view could lead to false assumptions and the loss of the opportunity to exploit the advantages of multiple disks. In this work, we extended a log-structured file system to take advantage of multiple disks.

MDLFS is a disk aware file system that tracks information about the performance of its storage system components. This knowledge gives MDLFS the ability to make informed decisions about the placement of data.

The performance of MDLFS has been compared to LFS and FFS on RAID. In our preliminary tests, the RAID systems place data equivalently across the disks without regard for their performance characteristics. MDLFS, on the other hand, is able to place data in a more strategic manner to overcome the variance in its disks.

Based on our results, we believe that making file systems aware of storage systems could lead to increased performance.

References

[1] Peter M. Chen and David A. Patterson. Maximizing Performance in a Striped Disk Array. *ISCA*, 1990, pp. 322-331.

[2] John H. Hartman and John K. Ousterhout. Zebra: A Striped Network File System. *Proceedings of the USENIX File Systems Workshop*, Ann Arbor, Michigan, May 1992, pp. 71-78.

[3] Iozone Filesystem Benchmark. December 2000. <http://www.iozone.org>

[4] James E. Johnson and William A. Laing. Overview of the Spiralog File System. *Digital Technical Journal of Digital Equipment Corporation*, 8(2):5-14, October 1996.

[5] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the Performance of Log-structured File Systems with Adaptive Methods. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[6] R. V. Meter. Observing the Effects of Multi-Zone Disks. *Proceedings of the 1997 USENIX Conference*, January 1997.

[7] John K. Ousterhout and Frederick Douglis. Beating the I/O Bottleneck: A Case for Log-Structured File Systems. Technical Report USB/CSD 88/467, University of California, Berkeley, October 30, 1988.

[8] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, Chicago, May 1988.

[9] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *Proceedings of the 13th Symposium on Operating System Principles*, October 1991, pp. 1-15.

[10] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. *USENIX Conference Proceedings*, June 1985, pp. 119-130.

[11] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An Implementation of a Log-Structured File System for UNIX. *Proceedings of the Winter 1993 USENIX Conference*, San Diego, CA, January 1993, pp. 307-326.

[12] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. File System Logging versus Clustering: A Performance Comparison. *Proceedings of the 1995 USENIX Technical*

Conference, New Orleans, LA, January 1995, pp. 249-264.

[13] H. Simitci and D. A. Reed. Adaptive Disk Striping for Parallel Input/Output. *Proceedings of the Seventh Goddard Conference on Mass Storage Systems and Technologies*, March 1999.

[14] Ken Shirriff and John Ousterhout. Sawmill: A High-Bandwidth Logging File System. *Summer USENIX '94*, Boston, MA, June 1994, pp. 125-136.

[15] B. Welch. A comparison of the Vnode and Sprite file system architectures. *Proceedings of the USENIX File Systems Workshop*, 1992, pp. 2944.

[16] Christopher Whitaker, J. Stuart Bayley, and Rod D. W. Widdowson. Design of the Server for the Spiralog File System. *Digital Technical Journal of Digital Equipment Corporation*, 8(2):15-31, October 1996.