Experiences with Implementing a Log-Structured File System[1]

Trishul Chilimbi (chilimbi@cs.wisc.edu)
Jussi Myllymaki (jussi@cs.wisc.edu)
Yoav Weiss (yoav@cs.wisc.edu)

Computer Sciences Department, University of Wisconsin-Madison

In this paper, we identify a performance bottleneck prevalent in current file system designs. With typical disk I/O usage patterns, only 5-10 percent of the data transfer rate of the disk can be utilized due to time spent in excessive disk head movement. A file system design that eliminates to a large extent the need to move the disk head when writing is presented. This file system design is called a log-structured file system, having similarities to a logging scheme usually found in database management systems.

We first describe the underlying scenario and the problems involved in more detail, after which we will outline our design and implementation. We used a trace-driven simulation to measure the performance of our implementation. Our results and findings show that this file system scheme can indeed utilize a very high percentage of the disk I/O bandwidth.

1 Introduction

1.1 The Disk I/O Bandwidth Problem

Current technology trends indicate that although CPU speeds are increasing by an order of magnitude every two years and main memory sizes are doubling every two years as well, the speeds of disk drives are barely improving at all. The main problem with disk drives is that the mechanical properties of the disks (speed at which read and write heads move) are unlikely to improve in the near future. While this makes track-to-track seek times remain the same over time, the I/O transfer rates of disks are increasing due to more dense information storage on the disk surface.

A software system that makes frequent read or write head movements (seeks) when accessing small files on the disk will experience the bottleneck described above [1]. The number of disk transfers needed to complete a transaction is fairly small (e.g. two disk accesses to read a page from a file), but the amount of time spent seeking will dominate the execution time of the whole transaction. Therefore, the data transfer rate of the application is not nearly as high as the transfer rate of the disk. For example, UNIX file systems typically can use only 5-10% of the disk bandwidth [2].

Solutions to overcome the problem include large caches for both reading and writing, caches with battery backup, cache logging, and log-structured file systems. Using large caches satisfies most of the read requests without disk access, but a good portion of the requests are write operations which means there is still a lot of data that must be written to the disk at random locations at a later time.

Log-structured file systems (LFS) try to get rid of the cause of the problem: the seeks. It maintains

---

[1]This project was done for the courses CS736 (Advanced Operating Systems) (Jussi, Yoav) and CS764 (Advanced Topics in DBMS) (Trishul, Jussi, Yoav). This is the final project document. December 17, 1992.

a large cache, but instead of updating files in-place on disk (at random locations), it keeps a log of the write operations. Appending the log information to a log file is a sequential write operation, so the system avoids seeking the disk and can fully use the transfer rate of the disk.

1.2 Log-Structured File Systems (LFS) in Related Work

A log-structured file system writes all modifications to disk sequentially in a log file, thereby avoiding the time-consuming disk seek operations. It is mainly used for speeding up the disk I/O, but is also used to provide faster crash recovery. For example, the Sprite LFS implemented at UC Berkeley can use 70-100% of the disk transfer rate, depending on the overhead consumed in reclaiming deleted file space [2]. For write operations, the performance advantage over conventional UNIX file systems is of an order of magnitude. For reads, they perform roughly the same.

The log is the only structure on disk, but it contains indexing information so that files can be read back from the log efficiently. From an indexing point of view, the structure of the log is very similar to that of a UNIX file system. There is a superblock and a list (table) of index structures (inodes). Each inode contains addresses of data pages of the corresponding file. However, in a UNIX file system, the location of these index structures is fixed. In an LFS, the index structures are floating: they are appended to the log like other file modifications [3][4].

Collectively, the index structures are called the map file. Because the map file doesn't have a fixed position in the log, the LFS must keep track of where the parts (blocks) of the map file are located. In addition to serving other purposes, the superblock describes the position of the map file blocks. Therefore, the super block is the starting point for all file references. It may be cached in main memory for fast availability.

The scheme described above makes all writes sequential while retaining the ability to perform random retrieval.

2 Internal Organization of an LFS

2.1 Structure of an LFS

The structure of an LFS is described by a superblock which is similar to a UNIX Fast File System (FFS, [1]) superblock in that it contains the file system parameters like segment size, number of segments, data block size, etc. The disk is statically partitioned into fixed size segments, typically half a megabyte each. The logical ordering of these segments creates a single, circular log.

LFS collects dirty pages and builds up a segment in memory. When this in-memory segment fills up it is written sequentially to disk to the next available clean segment. Thus in contrast to FFS the inodes are not at fixed locations on disk. LFS maintains a data structure called the imap (inode map) to locate inodes.

A physical segment contains one or more partial segments. Partial segments arise due to fsync or sync system calls. A given segment is always written sequentially from beginning to end, however the log is threaded on a segment by segment basis.

LFS follows a no-overwrite policy in that dirty blocks are always written to the current end of the log. Thus modified blocks are at different locations than the original blocks. To reclaim space resulting from dead blocks LFS has a segment cleaner that performs garbage collection by reading in a segment, identifying the live and dead blocks in the segment, discarding the dead blocks and appending the live blocks to the end of the log. To enable the segment cleaner to identify the live blocks, each segment has at its head a structure called the segment summary block which has information on all the datablocks and inodes stored in that segment. In addition LFS maintains a structure called the segment usage table that indicates the number of live bytes in each segment and the last modified time (age of the segment). The imap and the segment usage table are combined into a structure called the ifile. At checkpoint time the ifile is written to disk providing a stable point from which recovery can be started in case of a system crash.

The physical layout of LFS is shown in Fig. 1 (Appendix A).

2.2 Segment Cleaning

LFS performs segment cleaning to reclaim space resulting from dead blocks. It is a three-phase process consisting of reading a number of segments into memory, identifying the live data in these segments, and writing the live data back to the end of the log modifying the inodes and indirect blocks along the way.

To decide which segment to clean LFS follows a cost-benefit cleaning policy. The cost of cleaning a segment is 1 + u (where u is the utilization of the segment) as first the segment has to be read in and then the live bytes have to be written out. The benefit is the amount of free space obtained by cleaning the segment which is 1 - u. Moreover, free space in a cold segment (long lived data) is more valuable than free space in a hot segment (short lived data) as for a hot segment the data will likely die quickly and the free space will rapidly reaccumulate. Thus the system might as well delay cleaning for a while and let more of the blocks in the hot segment die. To estimate the stability of data in a segment the age is used under the assumption that the older the data in the segment, the longer it is likely to remain unchanged.

Thus:

$$\frac{\text{benefit}}{\text{cost}} = \frac{(1 - u) * \text{age}}{1 + u}$$

To clean a segment LFS must be able to identify the live data blocks in that segment. In addition it must be able to identify the file to which each block belongs and the position of the block within the file in order to update the file's inode to point to the new location of the block. To do this LFS uses the segment summary block which holds the inode number and the logical block number for every datablock in that segment. Using this, the segment cleaner can distinguish live data blocks from those that have been deleted or overwritten.

The segment cleaner starts cleaning when the number of clean blocks in the system falls below a parameterized Low Water Mark and continues cleaning until the number of clean blocks has risen above a parameterized High Water Mark.

3 Crash Recovery

When a system crash occurs the last few disk operations may have left the file system in an inconsistent state. In LFS the last disk operations are easy to determine: they are at the end of the log. LFS uses a two-pronged approach to recovery: checkpoints which define consistent states of the file system and roll-forward which is used to recover information written since the last checkpoint.

3.1 Checkpoints

To create a checkpoint LFS first writes out all the dirty pages to disk along with the ifile (which is written to a fixed location on disk) and then writes a checkpoint region to a special fixed position on disk. The checkpoint region contains a pointer to the last segment written and the current time. During reboot LFS reads the checkpoint region and uses the information there along with the ifile to initialize the file system. In order to handle a crash during a checkpoint there are actually two checkpoint regions and checkpoint operations alternate between them. During reboot LFS reads in both checkpoint regions and uses the one with the most recent time. Checkpoints are performed periodically as specified by a parameterized time interval.

3.2 Roll Forward

During roll forward LFS uses the information in the segment summary block to recover recently written file data. When a segment summary block indicates the presence of a new inode LFS updates the inode map it read from the checkpoint. This automatically incorporates the file's new data blocks into the recovered file system. If data blocks are discovered for a file without a new copy of the file's inode then the roll forward code assumes the new version of the file on disk is incomplete and ignores the new data blocks. The information in the segment usage table is also updated to reflect the live data left after roll forwarding.

4 Simulation and Performance Measurement

4.1 Tools and Methods

The underlying limitation that influenced the design of our experiments was the fact that our LFS has been built on top of the UNIX Fast File System. The results of the experiments directly depend on the testing procedures we designed, and there is no way we could compare our results to those that FFS would produce. The above limitation lead us to design an experiment which will answer two questions. The first is what is the best internal configuration of our LFS and the second is how it measures up with the FFS (rough estimate only). The measuring parameter we chose is the disk bandwidth. The disk bandwidth directly depends on the number of reads, writes, and seeks. Therefore, the equation we used for computing the disk bandwidth is:

$$\frac{\mathrm{Rcost}(\mathrm{user}) + \mathrm{Wcost}(\mathrm{user})}{\begin{array}{l}\mathrm{Rcost}(\mathrm{user}) + \mathrm{Wcost}(\mathrm{user}) + \mathrm{Scost}(\mathrm{user}) \\ +\mathrm{Rcost}(\mathrm{admin}) + \mathrm{Wcost}(\mathrm{admin}) + \mathrm{Scost}(\mathrm{admin})\end{array}}$$

where Rcost(user), Wcost(user), and Scost(user) are the time costs spent in reading (ie., doing read

I/O), writing, and seeking user data, respectively. Similarly, Rcost(admin), Wcost(admin), and Scost(admin) denote the time costs spent in reading, writing, and seeking administrative (e.g. segment cleaning) data, respectively.

Our experiments simulate some typical database transactions which include random reads and writes from and to different files, random number of joins using an index, frequent accesses of catalog files, sequential scans, etc.

4.2 Simulation parameters

There are several factors in our LFS that can affect the results a great deal. The three major factors we decided to test are: A. segment size, B. data block size, and C. number of buffer pool page frames. We varied each factor to have four different values which means that there are 64 combinations of those (see table).

Category	**Description**	**Values**			
		1	2	3	4
A	# of frames	32	24	16	8
B	block size	4K	3K	2K	1K
C	segment size	512K	384K	256K	128K

In order to reduce the number of experiments and still have some meaningful statistical results we used the "Latin Square" design [5]. This design reduces the number of experiments from 64 to 16. If the values of factor A are identified with the rows of a two-way table and the levels of B with the columns of the table, then the defining characteristic of a Latin square design is that every level of factor C appears exactly once in each row and once in each column. Our Latin square design looks like:

		A			
	C	1	2	3	4
	1	3	2	4	1
B	2	1	4	3	2
	3	2	3	1	4
	4	4	1	2	3

The value of k (the level of factor C) is completely determined by the values of i (factor A) and j (factor B). To emphasize this, we use $X_{ij(k)}$ to denote the observed value when the three factors are at levels i, j, and k respectively, with k taking on only one value for each i, j pair. The model is then

$$X_{ij(k)} = \mu + \alpha_i + \beta_j + \delta_k + \varepsilon_{ij(k)}$$
$$i = 1,...,N;\ j = 1,...,N;\ k = 1,...,N$$

where the $\varepsilon_{ij(k)}$'s are independent and normally distributed with mean zero and variance σ^2, and $\Sigma\alpha_i = \Sigma\beta_j = \Sigma\delta_k = 0$. In order to increase the validity of our experiments we repeated each run five times with a different seed for the random number generator and used their averages.

4.3 Simulation Results

The results of the simulation are summarized in the following table (blank spaces represent extreme results which were ignored).

Test No.	Combination	Bandwidth					Average
		1	2	3	4	5	
1	A1; B1; C3	54.6	58.0		86.6	84.7	71.0
2	A1; B2; C2	90.0	89.0		86.6	84.4	87.5
3	A1; B3; C4	32.1	28.2	22.1	43.2		31.4
4	A1; B4; C1	89.7	88.5	91.8	86.2	84.1	88.1
5	A2; B1; C1	90.2	89.1	92.0	86.6	84.7	88.5
6	A2; B2; C4	41.6	36.0	31.6	51.9		40.3
7	A2; B3; C3	33.8	41.8		86.3	84.2	61.5
8	A2; B4; C2	89.7	88.5		86.2	84.1	87.1
9	A3; B1; C2	90.2	89.1		86.6	84.7	87.6
10	A3; B2; C3	17.4	16.3	13.5			15.7
11	A3; B3; C1	89.7	88.7	92.0	86.3	84.2	88.2
12	A3; B4; C4	8.2	7.9	6.2	11.9		8.5
13	A4; B1; C4	16.4	17.9	16.5	16.3	21.2	17.7
14	A4; B2; C1	90.0	89.0	91.8	86.6	84.4	88.4
15	A4; B3; C2	89.7	88.7		86.3	84.2	87.2
16	A4; B4; C3	6.6	6.1	6.0			6.3

The statistical test that is performed is a multifactor analysis of variance (ANOVA) where for each factor the test statistic *f*(factor *i*) is Mean Squares(factor *i*) divided by Mean Squares(*error*). The null hypothesis for all factors is that there is no significant differences in the results, i.e., a different value of each category will **not** influence the expected result. The test statistic for each category is then compared with a corresponding critical value from the F distribution and if it is higher (for different significance levels) the null hypothesis is rejected. The following table summarizes these results with 1% and 5% significance levels.

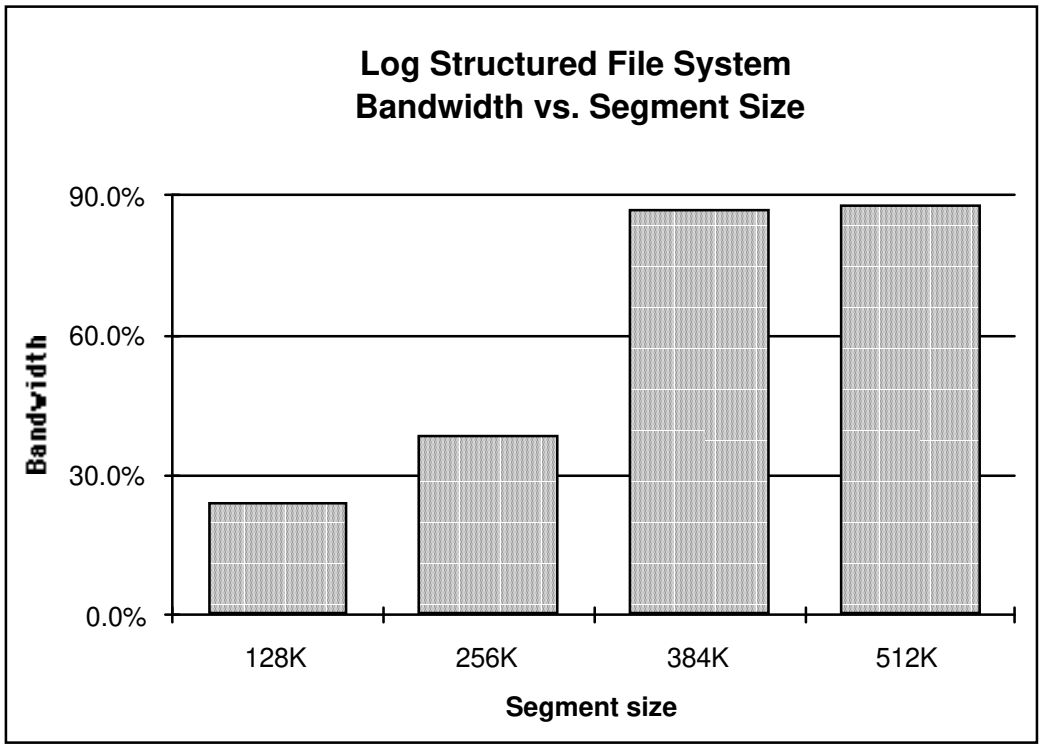
	Test Statistic	F(5%) 4.76	F(1%) 9.78
f(A):	2.47	accept	accept
f(B):	1.62	accept	accept
f(C):	21.28	reject	reject

As can be seen from the table, the only category whose null hypothesis was rejected is category C, namely, the number of segments.

4.4 Analysis of Simulation Results

Our tests produced two interesting results, one expected and the other somewhat surprising. The expected result is that bandwidth is dependent on segment size (see graph). As the segment size increases the bandwidth increases too. The second result is that bandwidth is not dependent on the

number of buffer page frames. In order to understand why, one needs to understand what influences the bandwidth. If the segment size is big the locality increases and therefore the number of seeks reduces regardless of number of page frames. On the other hand, when the segment size is small the locality decreases which causes an increase in the number of seeks and therefore the bandwidth decreases.



5 Conclusions

After implementing a log-structured file system and measuring its performance, we are quite convinced that the scheme can be effectively used to gain a disk I/O performance improvement of an order of magnitude, or even more, compared to conventional file systems. An LFS will be a viable alternative as long as the major restricting characteristic of a disk (it being a mechanical device) is limiting its accessing speed. This advantage is further enforced by the current trends of increasing core memory sizes and faster CPUs. In ten years, disks have presumably been replaced with other forms of mass memory, and, as a result, file system schemes will be totally reshaped.

6 Conclusions of the Project

6.1 Initial Plans

Our initial plan included designing, implementing and measuring the performance of a log-structured file system. The ultimate goal was to offer a user application the same kind of file-handling interface as is available in the UNIX operating system, but with a performance advantage. The semantics of the file-handling routines were to be the same as in UNIX.

In particular, our initial plan included the following:

- providing a file system with flat file name space (no directories)
- supporting large files
- offering file operations: create, open, close, read, write, lseek, unlink
- a segment cleaner (a garbage collector)
- taking checkpoints
- using caching to maximize throughput of disk I/O
- measuring the performance of the file system

In addition, our initial plan included the following optional features:

- a tree-structured file name space
- crash recovery using checkpoints

6.2 Actual Outcome

As a retrospect, we are quite satisfied with what we were able to accomplish. All features in the initial plan list were implemented, and a reasonable amount of measurement was conducted. Due to time limitations and guidance from our instructors, we did not concentrate on the optional features of the initial plan. Some shortcuts we taken (e.g. by implementing the co-operation of the LFS and the segment cleaner as a function call instead of two separate processes, and not supporting a multi-user environment) to allow us to concentrate on what we thought was essential.

Our implementation consists of roughly 4,000 lines of C code which includes library routines for file-handling (lfs_creat, lfs_open, lfs_close, lfs_unlink, lfs_read, lfs_write, lfs_lseek) and LFS maintenance (lfs_format, lfs_initialize, lfs_shutdown, lfs_clean_segment, lfs_checkpoint, lfs_recover, lfs_getdirentry), as well as a few utility programs (lformat, lcp lrm, lls) and our test programs.

6.3 What We Learned

We studied the disk I/O bandwidth problem and how one particular scheme tries to solve that problem. We learned the scheme by actually implementing a core part of it. We learned how a file system is constructed internally and how it is embedded in an operating system. Simulation and performance measurement results further taught us that the LFS scheme is a viable alternative, at least until the mechanical limitations of disks have been overcome by introducing some other (non-mechanical) form of mass storage.

References

[1] Ousterhout, J., and Douglis, F. "Beating the I/O Bottleneck: A Case for Log-Structured File Systems", ACM SIGOPS, January 1989, pp. 11-28.

[2] Rosenblum, M., and Ousterhout, J.K. "The Design and Implementation of a Log-Structured File System", ACM Transactions on Computer Systems, 10(1), February 1992, pp. 26-52.

[3] Seltzer, M., Bostic, K., McKusick, M.K., and Staelin, C. "An Implementation of a Log-Structured File System for UNIX", in preparation for the Winter -93 Usenix conference.

[4] Seltzer, M., and Stobraker, M. "Transaction Support in Read-Optimized and Write-Optimized File Systems", Proceedings of the 16th International Conference on Very Large Data Bases, Brisbane, Australia, 1990, pp. 174-185.

[5] Devore, Jay, L. "Probability and Statistics for Engineering and Sciences". Second Edition. California: Brooks/Cole, 1987.