Outline of Log-Structured File Systems

Erik Sven Vasconcelos Jansson
erija578@student.liu.se
Linköping, Sweden

– Review Copy –

ABSTRACT
With the advent of increasingly more powerful processing power and faster disk space accesses, disks have instead focused on making storage capacity larger and cheaper. This is unfortunate for file systems, which primary task is to manage these disks. In the article[2] by Rosenblum and Ousterhout a novel way to tackle this problem (and others) is presented: by laying out data sequentially and writing data in *segments* one can (potentially) achieve better bandwidth usage than previous contemporary file systems. Because of its *log*-like layout, *crash recovery* is also faster. This paper is a summary of that article, which presents *log-structured file systems (LSF)*.

MOTIVATION
Designing and implementing an entirely new file system from scratch was not something the original authors did out of the blue, they had clear reasons, based on several observations:

- **Technology:** processor and memory performance were evolving at a nearly exponential rate, while the disk was not. Since the main purpose of a file system is to manage this device, a solution to minimize the zenith latency (caused by re-positioning the read/write head) was needed.
- **Workload:** most stored files in a standard file system are small; a costly waste was happening by transfering these tiny amounts by themselves. This was caused largly due to the constant seeking through data structures.
- **File systems:** the *Unix FFS*[1] was effective at laying certain data out sequentially (decreasing seek time), however, it still spread out data structures on the disk, which meant multiple writes. Also, *crash recovery* took painstakingly long on other file systems, a complete traversal of the entire disk was often required (done with *fsck* in *Unix*).

DESIGN
To solve the above issues, Rosenblum and Ousterhout developed the LFS. To minimize latency caused by the mechanical read/write head, all data (including data structures) are written in a *sequential* manner, minimizing mechanical repositioning. Because small file writes are more common, a *write buffer* is employed, its task is to wait until a *segment* is full before doing a sequential write (allowing the LFS to write several small files in one go). Finally, *crash recovery* is solved by taking advantage of the *log*-like (ordered series of operations) layout on disk, a *checkpoint region* stores the last consistent state and queued operations to be performed. This section explores the design issues and solutions of a LFS.

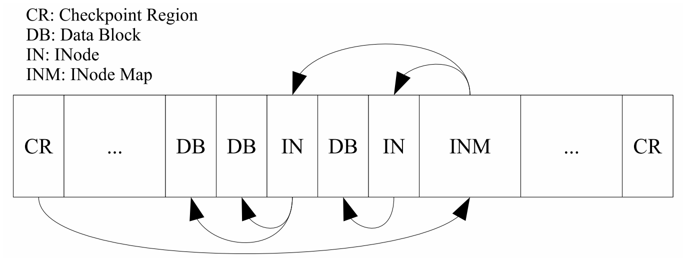

Figure 1. How the log-like structure conceptually looks like.

Reading and Locating Data Structures
Since all data structures are laid out sequentially, how does a LFS locate these structures to read/write data? The solution is to use a *checkpoint region*, which is the only data structure (specific to the LFS) that has a fixed address. This region is used to locate another data structure, the *inode map*; who's task is to transform the given *inode index* to the actual *inode*'s address. Thereafter, the inode will point to the actual *data blocks* (just like *Unix*). While this might seem slow, the inode map is usually cached in main memory (giving performance equal to the Unix FFS). See figure 1 above.

Writing and Free-space Management
As mentioned in the design introduction, data is written in segments and stored in a fast write buffer until full, thereafter it is written to the first free segment on disk. After a while, no free segments are going to be available. At this point, several segments should have been overwritten or deleted, the file system can then either *thread* or *copy* the segments to free space. Threading is optimal in a segment-by-segment basis, copying the segment to memory is perfect for defragmenting it when writing it back to disk (this is the primary way to free-up memory). This is called *segment cleaning*.

Cleaning and it's Policies
The idea behind *segment cleaning* is quite simple: read some segments into memory, identify those that have changed somehow (modified, deleted etc..) by looking at the *segment summary block*, write them back in a compressed form. This will enable the LFS to free disk space and keep the desired, tightly packed sequential structure. This however, poses several policy problems: **1)** When should this potentially expensive operation run? **2)** How many segments should it take into memory for cleaning? **3)** Which segments should be cleaned? **4)** How should the defragmented segments be written back to disk? See the *Sprite LFS* section later in this summary.

Crash Recovery
In the event of a system crash, file systems must try to do damage control by either reverting to a previous consistent state (before the crash) or try to puzzle together what the system *was* going to do before the crash. This is an issue that previous file systems had trouble to resolve, a *recovery* meant a traversal of the *entire* file system! In LFS however, since the last operation is at the end of the *log*, the last consistent disk state can be found by looking at the most recent *checkpoint* (two of these are commonly stored). In order to recover as much data as possible before the crash, a LFS scans segments *after* the checkpoint, to see if a *summary block* has been created with the next, lost operation. This is called a *roll-forward*, and is accomplished by allocating these "lost" data structures based on information found within the *summary block* (not completed, queued, write operations).

IMPLEMENTATION
In addition to designing the general concepts for a LFS, the authors also implemented a version of it called *Sprite LFS* for their network operating system *Sprite*. It was used for benchmarking comparisons against *Unix FFS* in the article[2].

Sprite LFS
The aspects described in the *design* section above are of a general LFS, hence, some issues like choice of policies were left out from that section since it was mostly an implementation issue. The Sprite LFS implementation starts the cleaning procedure when the amount of clean segments in the system falls below a threshold, it then continually cleans until a certain number of segments are free. Simulations were run by the authors on these policies and concluded that the first and second policy (described above) are not very important, the third and fourth are however, critical for performance.

For these important policies, they found out that a bimodal segment distribution, where most segments are full and few are nearly empty was key to high performance. In addition, by preferring cold segments (segments that don't change a lot) as cleaning candidates rather than hot segments, *Sprite LFS* achieves better performance than the other tested alternatives. This policy, called *cost-benefit*, allows segments with low utilization to be cleaned before recently references segments. Please refer to the original paper[2] for in-depth information and several policy simulations.

Results
Several micro-benchmarks[2] were run against the *Unix FFS* (specifically, the SunOS) to measure the gains in performance under optimistic and pessimistic scenarios. In a perfect scenario where no cleaning is involved, the performance for small file writes was a staggering ten times faster. Even large file writes gained performance (though not as much, as predicted). When cleaning is happening, only 70% of the disk bandwidth is used, which is still a performance gain over the contemporary *Unix FFS*'s varying 5-10% bandwidth usage. See figure 2 for some of the articles benchmark results.

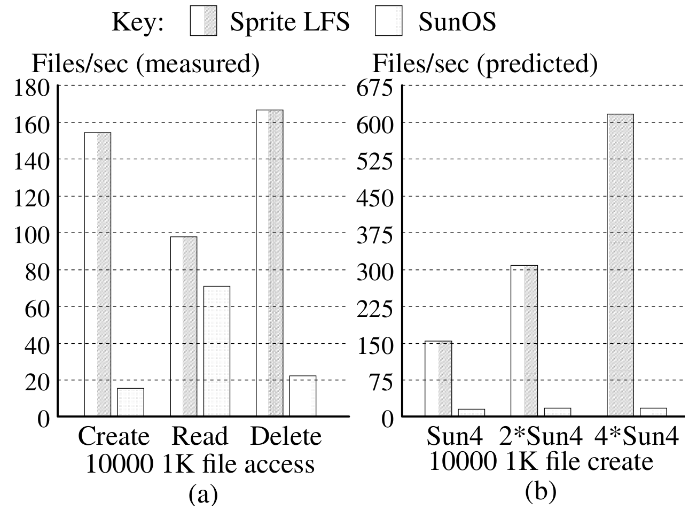

Figure 2. These describe the average small file performance for several different operations. Statistics and figure taken from the original article.

DISCUSSION
An interesting and well written article is presented by the authors Rosenblum and Ousterhout, it clearly shows the benefits (at least those shown in the micro-benchmarks) of implementing a *log-structured file system* compared to other[1] similar file systems of their time.

Impact
The controversy around the worst case cleaning costs (too hard to measure) was probably a major factor in the hesitant adoption of LFS. Still, many popular file systems have taken concepts from the article; the legacy lives on, on systems: NetApp's WAFL, Sun's ZFL and Linux BTRFS.

Questions
- With the arrival of solid-state drives, mechanical seek time is non-existent and speed has drastically been improved. Will there still be a place for a LFS (or similar) when this type of storage becomes more prevalent?
- Most modern file systems[3] implement a *journal*, which is an alternative way to handle recovery. The LFS also has a similar feature. Which one is more useful? More effective?
- Many modern HDDs employ an on-disk cache (for reading and writing), was this something that was taken into account when implementing the file system write buffer? More importantly, would it make a difference?

REFERENCES
1. McKusick, M. K., Joy, W. N., Leffer, S. J., and Fabry, R. S. *A Fast File System for UNIX*. 1984.
2. Rosenblum, M., and Ousterhout, J. K. *The Design and Implementation of a Log-Structured File System*. 1991.
3. Tweedie, S. C. *Journaling the Linux ext2fs Filesystem*. Linux Expo, 1998.