Summary Of “The Design and Implementation of a Log-Structured File System”

Robert Habrman
Linköpings Universitet
Linköping, Sweden
robha301@student.liu.se

ABSTRACT

This is a summary of the article “The Design and Implementation of a Log-Structured File System” that is written by Mendel Rosenblum and John K. Ousterhout in year 1991. The article is mainly about a new technique to improve the disk storage management and certain aspects of it. A technique that was new in the 1990’s is/was called a “log structured file system”. The main idea of this technique is to buffer a sequence of information in a file cache and then in one disk operation write all the information to the disks and by that trying to to improve the write performance.

INTRODUCTION

The subject of this paper is a new technique called a “log structured file system”. This technique is proposed because the trend that we have seen is that disk access times are improving in a certain speed but not even close to the pace that the CPU-speed development is increasing. This difference will most likely get bigger if the trend keeps going and that will make applications more disk bound. The technique will reduce the difference between these two parts by to increasing disk access times.

METHOD

Sprite LFS is a prototype log-structured file system that they have used in several simulations in the article. Sprite LFS is said to outperform the 1991’s Unix file system regarding small-file writes and is as good as or better regarding large writes to disk. A log structured file system converts many random synchronous writes from the RAM to the disk (traditional file systems) to fewer and larger sequential transfers from the file cash to disk which can utilize a larger part of the disks raw bandwidth.

So the idea of log structured system is that we are going to write data sequentially in a structure called the log to the disk, and never overwrite so called live data. Since the system don’t overwrite data we have to rewrite data to update data that is on the disk. We will discuss how the system manage that later in the summary.

Since the disk consist out of segments that contain blocks representing the files, the files are going to be split throughout the segments/blocks sequentially on the disk. Every segment is always written from its beginning block to its last. To rewrite a segment, it needs be completely empty.

In order to keep track of all the blocks and to what file they belong to the system has something called inodes. For each file in the system there is a inode that contains the files attributes such as owner, permission, type, disk address(pointers) of the blocks, etc. Whenever the system is going to update a block on the disk the system does not erase the old data (right away), it simply writes a new block, likewise for the inodes. The

Every segment also contains a segment summery block that keeps track of which file each block belongs to and also the position that the block has within file. To determine whether a block is live or not it checks with the files inode.

As I mentioned earlier the system buffers a sequence of information that is going to be written to the disk. To be able to reduce the risk of loosing data that is in the buffer in the RAM (it has not been written to the disk yet) the system can not have a to big buffer. But since the disk is spinning and we want to eliminate any kind of seek (for free space) we want to buffer enough information to write sequentially to the disk. So we buffer enough information in the file cache to avoid to seek on the disk for available space.

Since the system does not overwrite any data we have to write a new inode every time there is a update of a file or a new file written to the disk from the buffer. And in that case all the inodes of the files wont have any fixed location and that can be a problem if we want to read the file and can’t find the inode that belongs to it. In order to keep track of the inodes the system has something called inode map. A inode map contain pointers to all the inodes on the disks. And since the system write sequentially to the disks as it spins and it doesn’t seek for information on the disk the inode map will also be needed to be written to the disk

every time an update or a new inode is written from the buffer. The inode map will always be written to the disk last from the buffer since all the changes written to the disk essentially means that the inode map should be updated. To keep track of the inode map blocks that are stored on the disk there is a fixed checkpoint region on each disk.

Until now we have only seen how the system writes data to the disk, but what happens after it runs out of memory. To avoid running out of memory the Sprite LFS system have something called segment cleaning. Since we do no want the disk to be fragmented the segment cleaner compresses live data (blocks still having pointers pointing at them from inodes) from fragmented segments. The data that isn't live can be overwritten. The Sprite LFS system starts to clean segments when the number of clean segments reaches a fixed bottom-number and cleans until it reaches a fixed top-value. In order for the segment cleaner to be able to choose the most suitable segment to clean it uses a segment usage table that records data about each segment. The table keeps track of the the number of live bytes in a segment and the last time a block has been modified in that segment.

In order to reduce the risk of data loss when the system crashes they also implemented two important aspects in the Sprite LFS system. One called checkpoints and one called roll-forward.

The checkpoint is written to the disk after a specific amount time (since the last checkpoint was written) and to a fixed position when the system structure is in a "good" condition. The checkpoint itself contains necessary information for the system to reboot such as blocks of the inode map, the segment usage table, current time and a pointer to the last written segment. A checkpoint write time of 30 seconds had been implemented in the Sprite LFS.

Since a checkpoint is written in a specific time it can only contain data for just that specific moment. To prevent any kind of data loss between the checkpoint and the crash the Sprite LFS contain Roll-Forwad.

One of the simulations was made in order to compare different cleaning policies. A file system simulator was built. The simulator overwrites one of the files on the disk with new data in each new step. By using the simulator and different patterns together the Sprite LFS they discovered that "better" grouping and locality resulted in worse performance compared to a system without locality.

RESULTS

All of the different features that was briefly mentioned in this summary were implemented in the Sprite LFS system except the roll-forward method that didn't get installed in the production system. The Sprite LFS system was operational in the year 1990. In the article they also mention that there was no difference between the Sprite LFS and Unix system in the 1990 for an "everyday" user. The reason was that the machines at that time were not fast enough to be disk bound. Initially the Sprite LFS was developed to manage workloads of many small files but it turned out that the file system works for bigger files as well. The log structure file system was developed to use the disks capacity better than other file systems and by that reduce the difference of future development degree between faster processors and disc access time.

DISCUSSION

Reading an article 25 years later after it has been released makes me wonder if they released an even better version after the year they wrote this article. They also proposed solutions to make Sprite LFS faster by doing much of the segment cleaning by night so that the disk could have maximized its number of free segments, was this also implemented. I believed that the result regarding the Sprite LFS system was fair since they provide the reader with a lot of simulations and various tests which shows both the advantages and the disadvantages that the system has.

I also wonder if these kind of systems are implemented in the systems today and if they have been further worked on and developed even more. In the article they talk much about simple disks, and I wonder if this method is compatible for SSD disks. They also mentioned that the roll-forward method was not installed in the production system by the time this article was written, why didn't they install it, was there some complication with the method.

Some questions regarding the article:

Did this method contribute into a new way of thinking regarding different file systems?

What is the optimal time of the checkpoint interval?

How big should the file cash/buffer be in order to prevent to much data to be loss in case of a crash and in order to maximize performance?

REFERENCES

Mendel Rosenblum and John K. Ousterhout- "The Design and Implementation of a Log-Structured File System"