Ideas and Observations
Foundations for a versatile filesystem

Arkadiusz Bulski

Abstract

Cameleonica is a prototype of a highly versatile filesystem. This paper describes a wide range of concepts that could but have not been incorporated into popular filesystem designs. Both potential benefits and reasons why these features were not popularized earlier are presented. Expectations of their performance are then justified.

Introduction

When reading scientific papers that describe historical filesystem designs from the golden times of Unix development, one might come to a conclusion that evolution of filesystem designs was very incremental in nature and much effort was put into not making too many dramatical changes at any one time. Well known example would be a paper by McKusick et al, "A Fast File System for UNIX" where they mention increasing block size from 512 to 1024 bytes that lead to doubling of throughput from 2% to 4% of disk maximum performance. They conclude from this fact that increasing block size is a good method of increasing performance. Eventually they achieved 47% performance but it was a result of several modifications, including new allocation policies. One might be tempted to conjecture that achieving maximum performance might just as well require a complete overhaul of the entire design. Proposed filesystem is based on this approach.

New ideas

B-trees are not efficient. We can see a trend within filesystem design towards using both modified in-place and copy-on-write B-trees. This trend might be easily explained by common expectation that filesystems will be able to handle billions of files, that they will be able to scale. Today popular filesystems explicitly advertise their ability to scale among main points why to choose them over the competition. B-trees are a good approach if we expect huge amounts of keys to be stored. B-trees scale asymptotically with logarithmic complexity which means even billions of entries can be stored with only a few disk accesses needed to reach any given entry. This asymptotic behavior seems to have mislead everybody. It would be justified to use B-trees if we expected billions of files to be stored, but that seems to be a false assumption. Everyday experience suggests to me that most computers hold something on the order of 100'000 files. Even up to a million entries can be stored more efficiently by other means.

Any representation that can be efficiently manipulated and stored in memory, and can be stored and loaded from disk in one sweep from an extent will be more efficient.

Consider a hybrid or rather a transitional approach. Initially all entries are loaded and stored together. Due to usage, amount of entries grows over a certain threshold. We then trigger a transition operation where we switch to a B-tree representation. During transition we relocate all entries (already in memory) to tree nodes, and store all nodes into newly allocated extent in one sweep. The threshold can be chosen low enough so storing entire extent (in one sweep) is faster than analogous B-tree operation (few disk seeks), and also the transition is considered fast enough to be acceptable. Subsequent operations are carried out on a B-tree representation which is the compared alternative. In summary, efficiency below the threshold is better, crossing the threshold is made acceptable, and efficiency above the threshold is equal.

The threshold would actually be very high. We need to recognize that magnetic hard disks have quite disproportionate performance characteristics. Flash based disks are precluded due to reasons explained elsewhere. Representative hard disk is capable of ~120 MB/s of sustained throughput and ~10 ms seek time to random location. Quick calculation shows that reading or writing an extent smaller than 1.2 MB is faster than literally one seek, on average. Therefore a B-tree operation requiring say 4 seeks would be as fast as reading or writing 3.6 MB in one sweep. A huge amount of entries can be stored in that amount of data. Actually even more entries can be stored on disk than in memory. If for example entries are sorted in order then we can store their pairwise differences instead. Smaller numbers can be encoded more compactly using varint encoding as used in Protocol Buffers.

Inodes are not efficient. Since the early days, filesystems were designed under a principle that data structures have to be broken down into fixed size blocks. This seems to have been due to the fact that hard disks require read and write operations to be carried out on 512 byte long, discrete blocks of data. Furthermore hard disks guarantee that any given block is written atomically. For a long time filesystems depended on this feature to ensure ongoing consistency. Today checksums are becoming the mainstream means of ensuring consistency but in the old days available computational power was not enough to support this approach. Tables in McKusick's paper show computational power was a bottleneck already. Let us get back to data structures. To fully describe a single file, we need an amount of data that takes space of several blocks. One main block (called inode) would hold most important data, several blocks would point to locations where actual content is being kept, some blocks would keep directory entries in a grouped linked list, and so on. For a large 1GB file for example, at least 128 blocks would be needed. Certain operations such as copying file or browsing directory would necessarily require all metadata blocks to be read. Consider now a range of possibilities, where all metadata blocks are either allocated in one continuous range (called an extent), are divided into subsets, or are spread into individual blocks. Last approach is prevalent in modern designs. This approach is efficient with respect to changing metadata, as changing any one detail at once should require only one block to be overwritten, but it is not efficient with respect to subsequent reading. It could be argued that first approach, where all metadata is always loaded and stored in one extent, is necessarily better in every possible usage scenario. To show that, we need to recognize that magnetic hard disks have quite disproportionate performance characteristics. Flash based disks are precluded due to reasons explained elsewhere. Representative hard disk is capable of ~120 MB/s of sustained throughput and ~10 ms seek time to random location. Quick calculation shows that reading or writing an extent smaller than 1.2 MB is faster than literally one seek, on average. This observation may seem counter intuitive. Filesystem designs seem to be based on the assumption that data needs to be processed in smallest chunks possible and only way to go is process as small amount of them. It seems that processing bigger chunks instead never gained traction. Intuitively, processing bigger chunks could make sense because in long term perspective, processing bigger chunks saves much time (related to latency) at the cost of less time (related to throughput), only that time is shaved off a future operation. Going back to processing metadata of any given file. If we need to read or change only one detail, then extent approach is better due to reasons described above as long as extent is smaller than 1.2 MB, which for all practical purposes can fit metadata of any file. If we need to read or change more details, then extent approach is even better in comparison. Splitting the extent could only save throughput time by causing additional seeks but we already established that processing the entire extent takes less than one seek.

HDD and SSD optimality is mutually exclusive. Filesystem must be able to process large amounts of small write operations. That is not the only scenario in which it must perform but is an important one. Let us consider the extreme case of smallest changes possible, each change being the size of a byte.

[work in progress]

Resurrected ideas