# 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.

## 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 but 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 2 seeks or more would be slower than reading or writing 1.2 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. In traditional designs, most operations can be done by processing one block at a time. Computers of the past often had little memory and caching complete sets of metadata was not feasible. Also changing metadata often ended up in writing just one block, which might have been seen as a compelling reason to use this approach. Furthermore hard disks guarantee that any given block is written atomically. All these reasons contributed to a decision that data structures should be broken down into blocks.

To fully describe a single file, we need an amount of data that almost always takes space of more than one block. Traditionally, one main block (called inode) would hold most important data, several blocks would point to where actual content is located, some blocks would keep directory entries in case of a directory, and so on. For a large 1GB file for example, assuming 8KB blocks and 64 bit pointers, at least 128 blocks would be needed. Certain operations such as copying or deleting 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 divided into individual blocks. Last situation is prevalent in modern filesystems. Space is allocated conservatively, one block at a time leading to fragmentation of metadata.

This approach seems efficient at first glance because changing one detail should require only one block to be written. This kind of reasoning is flawed because we do not consider the impact on further operations into account. We should consider amortizing performance over entire lifespan of a file.

It could be argued that first approach, where all metadata is always loaded and stored in one sweep 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 to process as small amount of them. It seems that the opposite approach of processing bigger chunks never gained traction. In traditional designs, directories put

groups of entries into a linked list of blocks, files put data locations into indirect blocks, and so on. In modern designs like Btrfs, cow btrees are replacing linked lists but data is still being divided into small chunks.

To show first approach is better we will consider total time spent on all operations throughout a lifetime of a file. Only metadata is counted towards the time spent reading or writing. Parent directory metadata is also excluded. At time zero file metadata has no copy on disk. We let a certain number of cycles of the file being opened, changed several times and then closed. After each open, half of metadata blocks are read (in last approach) or all metadata blocks are read (in first approach) by design. Cached blocks are forgotten between cycles due to assumed long time intervals.

It is hard to estimate how many blocks would be needed to lookup a random entry but if metadata was broken down into a linked list then one random lookup would require going through half the list. The more lookups the more of the list we would have to go through. On the other hand, indirect addressing blocks form a tree and not a list. Ultimately half the list is a pure assumption.

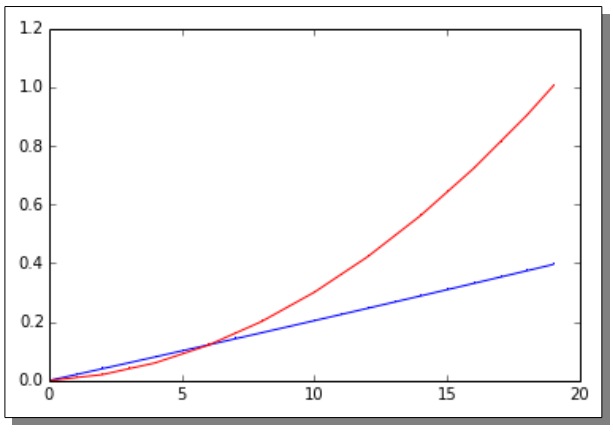
Each time file is opened we make a certain number of changes to metadata, where each change can be reflected in changing just one block on disk. These changes will be applied by overwriting their corresponding blocks on disk in place.

Reading or writing N blocks would take (first and last approach respectively):

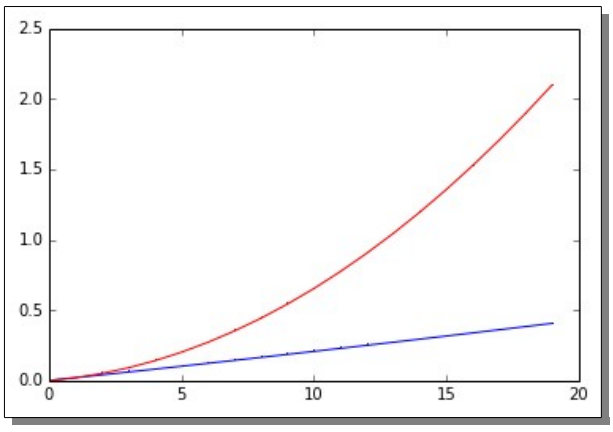
$$R_1(N) = 0.010 + N \cdot 4\,K/120\,M = 0.010 + N \cdot 0{,}00003$$
$$R_N(N) = N \cdot (0.010 + 4\,K/120\,M) = N \cdot 0.01003$$

We will run sort of simulations with two independent variables, N is number of changes in each cycle and M is number of cycles. After each cycle, the number of blocks will grow by N. Therefore total time would be:

$$T_1(N, M) = \sum_{i=0}^{M-1} R_1(N \cdot i) + R_1((N+1) \cdot i)$$
$$T_N(N, M) = \sum_{i=0}^{M-1} R_N(N \cdot i \div 2) + R_N(N)$$




M=1




M=2

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 the time is shaved off of a future read operation.

Going back to processing metadata of a 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 due to coalescing. 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.

For a long time filesystems depended on this feature to ensure ongoing consistency, in particular the Soft Updates method developed by McKusick and Ganger. Later on journaling became the way of ensuring consistency (as Steven Tweedie talks about Ext3) and today checksums are also becoming popular (with Btrfs and ZFS paving the way). As computational power and memory are now in abundance we see a shift in approaches to filesystem design.