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.

[work in progress]

New ideas

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 [Twee00]. All these reasons contributed to a decision that data structures had to 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 1 GB file for example, assuming 4 KB blocks and 64 bit pointers, at least 512 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 most of the lifetime of a file. Only metadata is counted towards the time spent on read and write operations. Parent directory metadata is also excluded. At time zero we assume that all file content and metadata is already stored on disk. We let a certain number of cycles of the file being opened, accessed several times and then closed. After each opening either some (in block approach) or all (in extent approach) metadata blocks are accessed, either read or written, no matter which. It does not matter which because metadata needs to be read before it can be changed. This is a simple fact that if data is written in place then we need to know where to put it and if data is copy-on-written then we need to deallocate previous data. Either way, we need to read metadata before we can write it. Also cached blocks are forgotten between cycles due to assumed long time intervals.

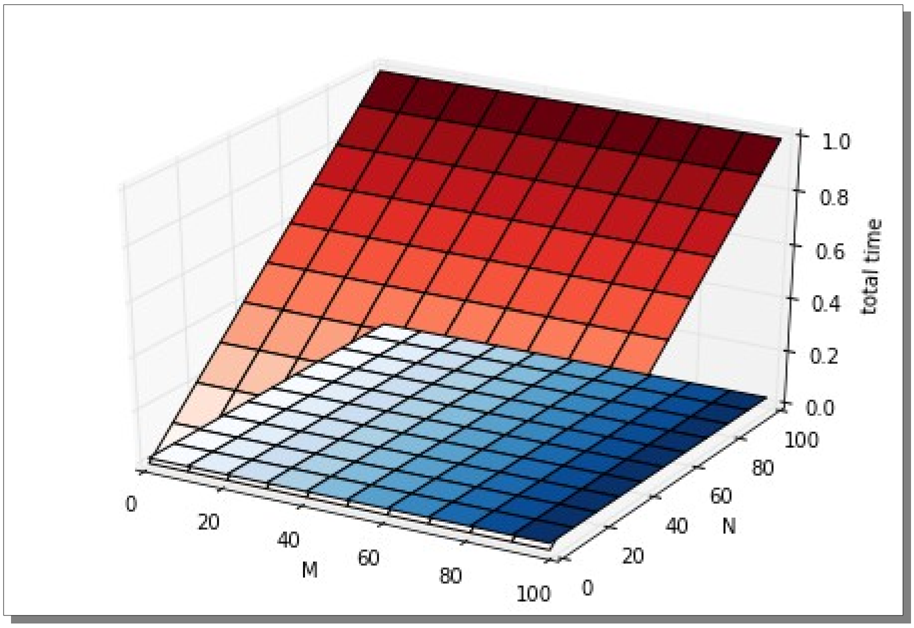
Reading or writing N blocks would take (extent and block approach respectively):

$$R_1(N) = 0.010 + N \cdot 4\,K/120\,M = 0.010 + N \cdot 0.00003$$
$$R_N(N) = N \cdot (0.010 + 4\,K/120\,M) = N \cdot 0.01003$$

We will run a simulation with three independent variables, N is number of blocks accessed in each cycle, M is number of blocks total, and B is number of cycles. Therefore total time would be (extent and block approach respectively):

$$T_1(N, M, B) = \sum_{i=1}^{B} R_1(M) + R_1(M)$$
$$T_N(N, M, B) = \sum_{i=1}^{B} R_N(N)$$

Plots show total time of extent approach (blue) and block approach (red) for least and most accessed, least and most sizable files.



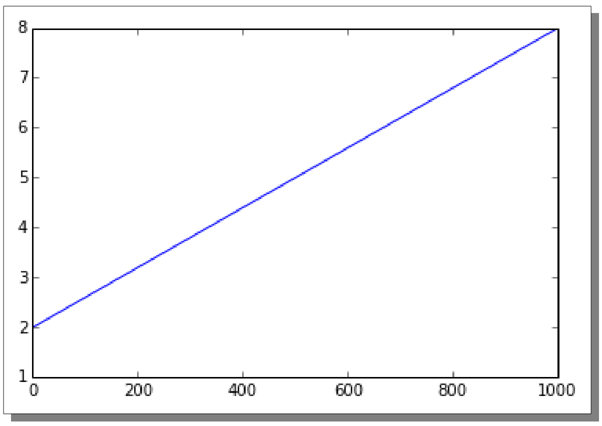
Let us look at the cases where extent approach looses in comparison. Total times can be compared through an inequality. Notice that number of cycles cancels out.

$$T_N < T_1$$
$$B \cdot R_N(N) < B \cdot 2 \cdot R_1(M)$$
$$0.01003 \cdot N < 2 \cdot (0.01 + 0.00003 \cdot M)$$

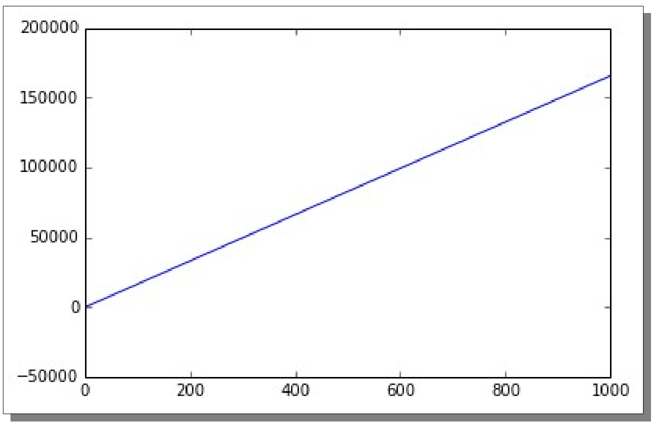
$$N < 1.994 + 0.006 \cdot M$$
$$M > 166 \cdot N - 332$$

These last two inequalities show limits on how much metadata could be accessed at one time before block approach starts to loose advantage. Asymptotically, less than 1 in 166 metadata blocks could be accessed. For extremely small files the limit is 2 blocks.

Left plot shows how many metadata blocks at most could be accessed given how many metadata blocks the file has in total. Right plot shows how many metadata blocks at least would the file need to have given how many metadata blocks are accessed. Exceeding these limits means the block approach looses advantage.



Solved for N



Solved for M

Remember when earlier it was mentioned that a 1 GB file would require at least 512 metadata blocks? According to the limits above, after opening it we could access at most 5 metadata blocks before block approach would loose advantage, and this includes the inode block. Is this really the kind of workload we are aiming to support?

When we store individual blocks we gain on transfer time related to small block size but we pay for it with seek time the next time we read said blocks. Disk characteristics make this exchange totally unfair. One seek takes as much time as transferring 1.2 MB which can for all practical purposes fit complete metadata of any imaginable file, ever.

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 should be able to handle billions of files, that they will be able to scale. Popular filesystems even 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 is a false assumption at least in general case. Most computers hold something on the order of 100'000 files. See [Agraw07] for representative statistics. That amount of dictionary entries can be stored more efficiently by other means.

Consider a hybrid or rather a transitional approach. Initially all entries are loaded from one extent, kept in memory in entirety and all the time, and occasionally stored to disk as one extent. If at some point the amount of entries grows over a certain threshold, we commence a transition to a B-tree representation, we relocate all entries (already in memory) to tree nodes, and store all tree nodes to disk in one sweep. The threshold can be chosen low enough so storing entire collection in one sweep is faster than analogous B-tree operation (few disk seeks). Subsequent operations are carried out on a B-tree representation which is the compared alternative.

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 an entire 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 entry keys are sorted then we can store their pairwise differences instead. Smaller numbers can be encoded more compactly using varint encoding as used in Protocol Buffers.

This argument is basically the same as for the inode extent. If you consider the mapping entries as metadata of some special file then exactly the same argument can be applied.

Bibliography

Agraw07: Nitin Agrawal, William J. Bolosky, John R. Douceur, Jacob R. Lorch, A Five-Year Study of File-System Metadata,
Twee00: Steven Tweedie, EXT3, Journaling Filesystem, 2000, http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html