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 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 [McKusi84] 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 metadata 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. First approach we shall call the *extent approach* and it requires that all blocks are read from an extent at once and after changes are made all blocks are stored to an extent at once. Last approach we shall call *block approach.* It is prevalent in modern filesystems. Space is allocated conservatively, one block at a time eventually leading to fragmentation of metadata.

Block approach seems efficient at first glance because changing one detail should require only one block to be written to disk. This kind of reasoning is flawed because we do not account for future operations. We should consider amortizing performance over entire lifespan of a file, not assume that any one operation must be carried out as fast as possible independently of other further operations.

It could be argued that first approach, where all metadata is always loaded and stored in one sweep is better in every usage scenario. To show that, we need to recognize that magnetic hard disks have quite skewed 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 on average. 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 so far 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 [OSTEP] directories put entries into data blocks, files put data locations into indirect blocks, and so on. In modern designs [Rodeh12] [ZFS] copy-on-write B-trees are replacing indirect blocks but data is still being divided into small chunks. The issue being described here has not gone away.

To show that extent 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 some metadata blocks are accessed, either read or written. We also assume that cached blocks are forgotten between cycles due to assumed long time intervals between cycles. Blocks are assumed to be 4K size. Suffixes like K M refer to thousands and millions of bytes.

Reading or writing N blocks would take (assuming random and sequential pattern, referring to extent and block approach respectively):

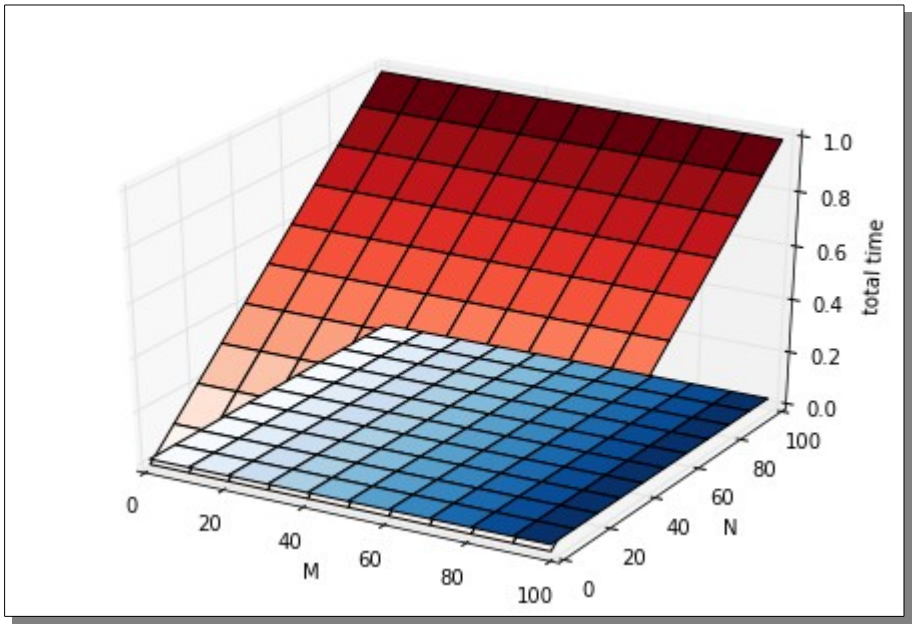
$$R_1(N) = 0.010 + N \cdot 4\,K/120\,M = 0.010 + N \cdot 0.00003$$
$$R_N(N) = N \cdot (0.010 + 4\,K/120\,M) = N \cdot 0.01003$$

We will conceive 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):

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

Notice that extent approach causes more blocks to be copied from/to disk than what is actually accessed by a process. This is intentional.

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.

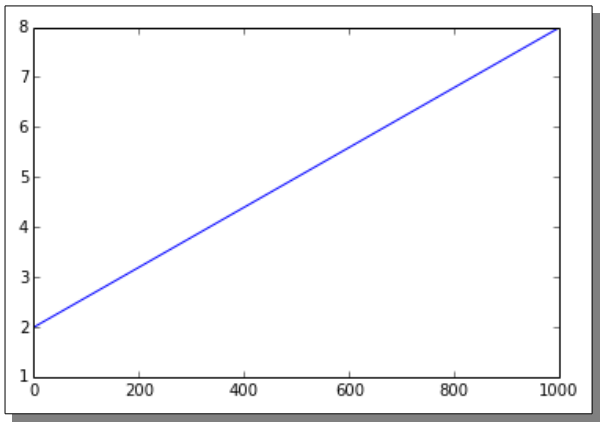
$$T_N < T_1$$
$$B \cdot R_N(N) < B \cdot 2 \cdot R_1(M)$$
$$0.01003 \cdot N < 2 \cdot (0.01 + 0.00003 \cdot M)$$

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

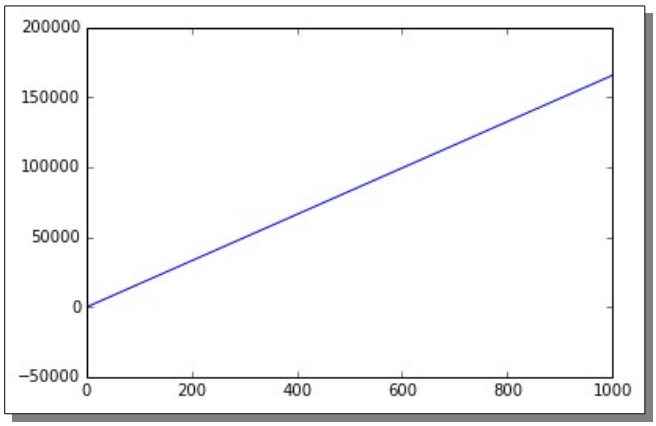
These last two inequalities show limits on how much metadata could be accessed during one cycle before block approach starts to loose advantage. Asymptotically, less than 1 in 166 metadata blocks could be accessed. For smallest files the limit is 2 blocks.

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?

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

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.

Additionally extent approach has the benefit that one checksum is enough to verify integrity of all metadata while block approach would require either as many checksums as there are blocks or reading additional blocks and checksumming them together.

Finally it should be admitted that the theoretical model above assumes most naive implementation of block approach. It would be possible to allocate blocks individually but as close to each other as situation allows. Orlov allocator is the mechanism meant to achieve exactly that. Model above assumes blocks are allocated uniformly from entire disk. However this would make the block approach only so much better in comparison and it is doubtful that entire argument would become invalid.

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 [Rodeh12] [ZFS]. This trend might be easily explained by common expectation that filesystems should be able to handle billions of files. 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 the whole 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) or is faster when multiple changes are accumulated. After transition 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 skewed 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 on average. 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 the same theoretical model can be applied.

Fsync is not efficient. Files stored on computers were important since about the time when files started being stored on computers. Programs started using techniques that would ensure reasonable state after computer was interrupted in it's work which did and still does occur quite often. POSIX standard implies an approach that is being used to this day. Windows system uses an identical approach despite not recognizing POSIX. The well established approach replaces a file using following template:

- Open a new file that will replace some existing file
- Fsync directory above
- Write content to new file
- Fsync new file
- Close new file
- Rename new file to replace existing file

This approach has been the expected way of achieving atomic changes to files and caused a lot of pain when people were depending on different guarantees than that of fsync. For example, ext3 implemented fsync in a naive way that flushed all files to disk and not just the file of interest, which caused it to be slower than perhaps it should have been while at same time implemented non-POSIX behavior of ordering data writes before metadata writes which made following rename safe. These two facts made fsync both slow and unnecessary. As result people started skipping on fsync. Later on ordering behavior was disabled to increase performance and non-standard code started corrupting files. This story has been described in LWN post POSIX vs Reality.

Another example is a property called *safe new file* that makes fsync on parent directory unnecessary. SQLite in particular allows to take advantage of this behavior although it is optional and depended on by default. This behavior is common in modern filesystem but it not required by POSIX and applications cannot rely on his property being met. Refer to ??? for modern filesystem semantics that are common but not standard, same article

dissects what SQLite requires and can take advantage of.

Whatever improvements to filesystems will be made in the future it seems clear that the applications will remain to use the approach established by POSIX and filesystems will have to cooperate. Inventing new APIs that break existing code will not gain traction, no matter how fancy they seem.

Below we shall consider a new solution to this problem that will keep files correct weather the POSIX approach or ordering behavior based approach was taken while performance is approaching no use of fsync at all.

Consider a filesystem where all operations are both atomic and ordered. Above that fsync calls are implemented as no-op.

We can see that template code remains to keep files consistent. This is due to the fact that application developers (usually) use fsync not to persist data immediately but to persist data before metadata, which is exactly what ordering means. Linux man pages define fsync as a means to persist data immediately but surprisingly this is not what POSIX strictly requires but rather just a mainstream interpretation. POSIX first defines fsync (in vague terms) as flushing buffers to a device but then explains in the notes, and explicitly, that if a filesystem can guarantee safety of files in a different way then that also counts as a valid implementation of fsync. Flushing buffers is an obvious way to do it but not the only way. Excerpt from POSIX documentation:

> *(DESCRIPTION) The fsync() function shall request that all data for the open file descriptor named by fildes is to be transferred to the storage device associated with the file described by fildes. The nature of the transfer is implementation-defined. The fsync() function shall not return until the system has completed that action or until an error is detected.*
>
> *(RATIONALE) […] It is explicitly intended that a null implementation is permitted. This could be valid in the case where the system cannot assure non-volatile storage under any circumstances or when the system is highly fault-tolerant and the functionality is not required. In the middle ground between these extremes, fsync() might or might not actually cause data to be written where it is safe from a power failure.*

Persisting ordered operations used to be implemented though ordered persistence, that is blocks were written synchronously effectively putting a write barrier between each write. Although this approach guarantees ordering there is another approach.

Consider a copy on write scheme where first sector contains address of last persisted block and address of root inode, further sectors contain a continuous stream of data and metadata similar to what a log-structured filesystem would contain.

Each operation puts more blocks into the stream in a sequential but asynchronous way. Specifically metadata blocks are put after data. When the filesystem is sure that blocks up to a given point were persisted it issues an atomic but also asynchronous write of the first block, moving the address of last used block. Recovery is not strictly necessary because first sector reliably tells us how much data was surely stored.

Compactness can be achieved by storing intents instead of full structures. For example, in log structured filesystem a single file write could end up storing the data buffer, indirect blocks, file inode, parent directory data and inode, and so on up to root inode.

Instead only the data buffer and an intent describing to which file this data belongs to would be enough. Inode would have to be stored eventually but not before write was persisted. Recovery would have to involve doing a so called *roll forward*, scanning the log from last known end forward, looking for intents and verifying checksums. File write intent would have to be incorporated into earlier inode during recovery. Complexity is added to recovery code but performance gain may well be worth it.

Bibliography

McKusi84: Marshall Kirk Mckusick, William N. Joy, Samuel J. Leffler, Robert S. Fabry, A Fast File System for UNIX, 1984
Twee00: Steven Tweedie, EXT3, Journaling Filesystem, 2000, http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html
OSTEP: Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau, Operating Systems: Three Easy Pieces,
Rodeh12: Ohad Rodeh, BTRFS: The Linux B-tree Filesystem, 2012
ZFS: Jeff Bonwick, Bill Moore, ZFS: The Last Word in File Systems, ,
Agraw07: Nitin Agrawal, William J. Bolosky, John R. Douceur, Jacob R. Lorch, A Five-Year Study of File-System Metadata, 2007

Please know that all cited documents are saved in project's repository and you do not have to search the web to read them. This includes cited web pages.