Avionics Systems

AVS-DOC-TN-0000
PAFFS Documentation

Issue: 0.2
Date: 2016-08-31

DLR

Contents

List of Figures

List of Tables

Documents

Reference Documents

[RD1] Artem B. Bityuckiy. *JFFS3 design issues*. Standard. 2005.

[RD2] Ferenc Havasi. *An Improved B+ Tree for Flash File Systems*. Standard. 2011.

[RD3] Ferenc Havasi et al. *JFFS3 plan extension*. Standard. 2006.

1 Introduction

PAFFS stands for "Protective Avionics Flash FS" and aims to use a minimum RAM footprint with the ability to manage multiple flashes. Some of the design ideas are inspired by the file system JFFS3 [RD1], which was later extended [RD3] and discontinued as predecessor of UBIFS.
The main differences to usual disk file systems are out-of-place-writing, a higher bit error rate and high deletion costs in terms of durability and speed. Keeping track of the files data chunks has to be different than just maintaining a big table on a disk, because the high frequented lookup table would be worn out earlier while other parts of flash will be nearly untouched. The standard approach of flash aware file systems (such as YAFFS) is to maintain a table-like structure in RAM and committing every chunk of new data to flash with an unique, increasing number, just like a list. This adds far more complexity to the file system as it has to scan the whole flash when mounting, but increases lifetime of the flash enormously. However, this RAM table grows linear with flash size, and thus does not scale for modern (> 2GB) flash chips. To solve this, the information has to be on flash. B+-Tree -> Section 1.1.8. Another big challenge is to reduce wear. Every change to data has to be written to another place and the old location has to be invalidated somehow. This is because the smallest unit of deletion is bigger (usually around 512 - 4086 times) than the smallest unit of a write operation. The common approach is to give every logical chunk an increasing version number. Because of the disadvantages pointed out earlier, this is not applied in PAFFS. Areas and address mapping -> Section 1.1.1

1.1 Fundamental structures

1.1.1 Areas

Overview An Area is a logical group of erase blocks. The number of contained blocks is a trade-off between RAM usage (less Areas) and garbage collection effectiveness (less overhead). Areas also separate different types of data and simplify handling of addresses after moving data. At the end of each Area, some amount of pages (usually not more than one) are reserved for an area summary (see chapter 1.1.4).

- AreaType can be one of Superblock, Index, Data, (Journal). See table 1.1.
- Address split in logical area n° and page n°, to give way for a simple garbage collector (see fig. 1.1).

- `AreaMap` held in flash (but cached in RAM) translates between `logical area n°` and `physical area n°`. It also keeps record of corresponding types and usage statistics for garbage collection.
- `AreaStatus` can be one of `closed` (probably full, no guaranteed `area summary` in cache), `active` (`area summary` in cache, free pages) and `empty` (no `AreaType` set, no used pages).

1.1.2 Area types

Table 1.1: Area types

Type	Description
Superblock	One superblock area is on the first area of flash, the rest is dynamically allocated. It contains the anchor blocks as well as jump pads and a superpage. See chapter 1.1.7
Index	The index areas will only contain tree nodes (some of them storing inodes). See chapter 1.1.8.
Data	Data areas contain data chunks of files, directory entries and softlinks referenced by the index.
Journal	The single journal area (some when) will contain uncommitted changes of the `area Summary`.

1.1.3 Area map / Addresses

Due to the nature of flash, any deletion is delayed as long as possible. But when free space runs low, a garbage collector has to delete dirty pages while keeping addresses valid. This is why addresses consist of two parts; a logical area number, and page offset inside this area. To read a chunk of data at an address, the logical area number has to be translated to a physical area via the `AreaMap`. It is stored in the superpage (see chapter 1.1.7), but is cached in RAM.
This area map grows linear in size with the number of areas which is defined as block count divided by area size.
The size of an area is a trade-off between low RAM usage (big areas) and a more efficient garbage collection (small areas). In this test environment, an area size of two erase blocks is chosen[1]. With a normal 2GB flash chip[2], this configuration would use $(3+2+16+32+32)/8*2^{21}/2 = 11141120$ Byte $= 10,625$ MB RAM. Increasing the size of an area to 8 erase blocks would result in 2,65 MB usage, which would suit the requirements better.

[1] Which should be the minimum size, because the anchor area has to be two blocks in size.
[2] 2048 Byte user data per page, 1024 pages per block, $\approx 2^21$ blocks.

1.1.4 Area summary

Overview To distinguish between `free`, `used` and `dirty` pages, a per-area list is held in cache. The Information is used by the garbage collection copying only used pages to a new area, by the function looking for free pages for writing data or index chunks, and by runtime sanity checks. The Caching techniques are described in chapter 1.1.6. If the cache runs low or an unmount is requested, a packed area summary is written to the last pages of an area. Due to the nature of flash, this can only be done once until the area is garbage collected and therefore committable again.

Functionality For saving space, the information of `free`, `used` and `dirty` pages is truncated to only hold the information `used` or `other` for each page. The missing knowledge can be reattained by sweeping the whole area upon unpacking. If a page is empty, it is considered `free`; `dirty` if not.

1.1.5 Garbage collection

Overview The garbage collection needs at least one free area to copy valid data to. At first, one or more of the `closed` areas are inspected (as the `active` areas are still in use), sorted by texttdirty pages and eventually chosen for garbage collection. All valid pages are copied from the old area to the new area in their same relative positions. After that, the old area is erased, and the `AreaMap` is updated so that the logical areas swap their physical positions. The new area is marked *active*, and the `AreaSummary` is deleted from Flash, thus giving way for a new commit of an area summary. If the previous way is not sufficient, the `garbage buffer` is given up for storing the more information. This procedure can not be reverted, and is reserved for `index` type only to enable a safe unmount without information loss.

Functionality Besides of looking for the most dirty blocks, the garbage collection also keeps the committed area summaries in mind. Depending on the mode (just looking for free space or trying to delete a committed area summary), it prefers or exclusively looks at areas with a committed area summary. This procedure helps the area summary cache freeing old entries.

1.1.6 Area Summary Cache

Overview The area summary caches read and write accesses to the area summaries of any area. Due to the limitation of RAM, the filesystem can't keep the status of every page in every area accessible. Every accessed area summary is loaded from cache (if previously committed to area) or created, until cache runs full. To free cache entries, the area containing the most dirty pages is committed to flash and deleted in cache. The minimum number of available cache entries is three, as two are needed for the `active` areas of

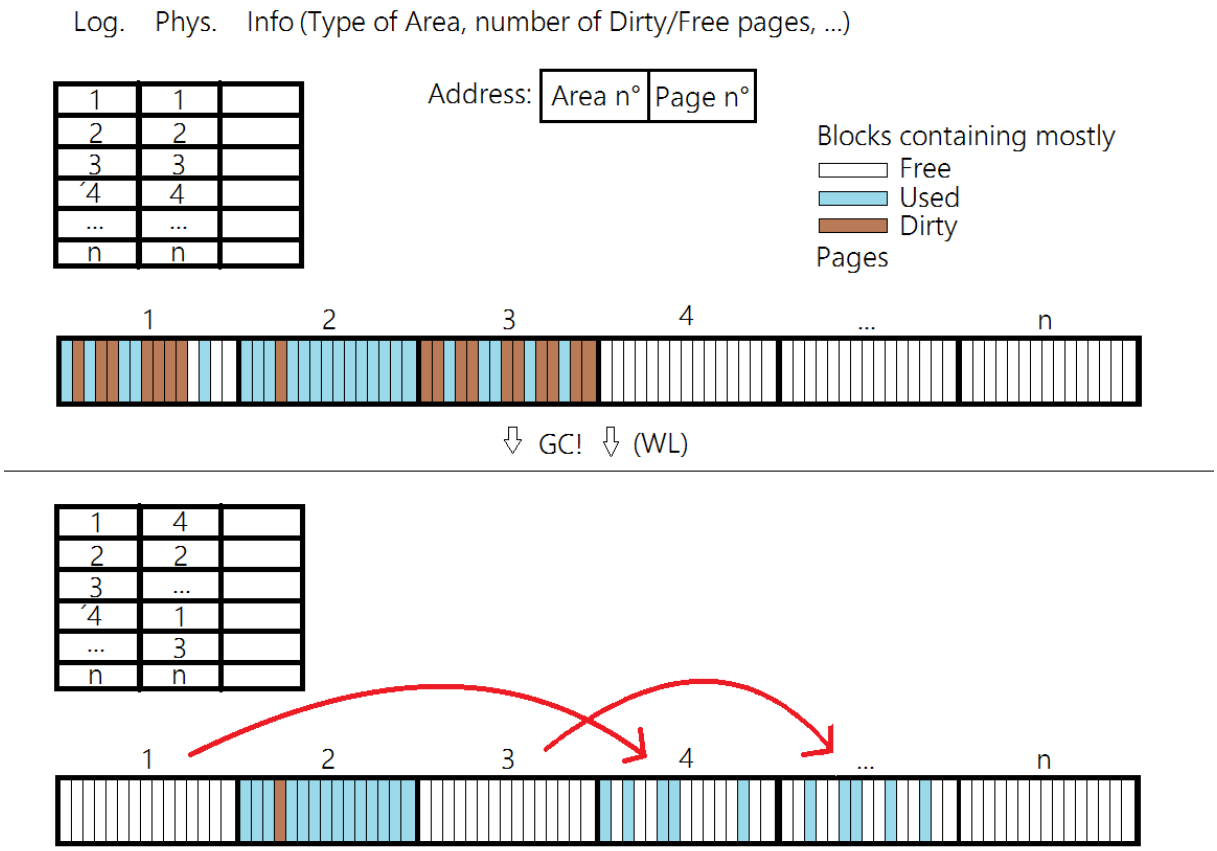


Figure 1.1: Basic process of garbage collection with areas

`data` and `index` type, and one for simultaneously invalidating obsolete data. It is possible without the third cache entry, but the performance and long-term stability is very poor. A good cache size depends on simultaneously opened files for increased speed, however 5-10 entries will be sufficient for most Applications. Depending on the number of data pages per Area, a single entry uses between 32 and 256 bytes.

Functionality There are three different caching strategies implemented. When enough cache is available, every read or write access to a new area loads its contents to cache. If no free cache entries remain, and no cache entry is uncommitted (and therefore easy to delete), two possibilities remain. If a read access is queried, the corresponding area summary is loaded in a one-shot read without loading it to cache[3]. If it is a write access, the garbage collection is called in `urgent` mode to free any cached area from its obsolete (dirty) area summary[4]. After that, a cache entry can be freed and the new one can be loaded. If garbage collection returns no committable Area (when free space is extremely low), the write access is ignored and an error is thrown. This only happens to an invalidation (former `used` page `dirty`), so the filesystem would not break, but gain unreachable pages that wont be deleted. The other transitions (from `free` to `used` and from any to `free`) are

[3]Experiments show that there is less response-time and no improvement in wear leveling if the one-shot mode is used earlier, i.e. when no free cache entries remain without considering to commit areas.
[4]This also frees all other dirty pages as well.

not possible to throw an error. The transition `free` to `used` only occurs on write actions in `active` areas which are guaranteed to have an entry in cache, the other one occurs in garbage collection freeing the whole area and thus deleting the whole cache entry. Due to the layout of any commonly used processor, the size of an array element is mostly over 8 bit. To reduce the amount of RAM used, four 2 bit elements are packed into one byte. This quadruples the number of cacheable areas.

1.1.7 Chained superblock

Overview Along with some static information, keeping an index requires having some sort of start point to find the first address of a table or the first position of a root node. The naive approach is having the first block contain this dynamic information. However this is bad practice, as it wears off the first (or first n) blocks enormously while leaving other places barely used. Another approach is to scan the whole flash for something with an unique sequential number, which is unacceptable as mount time scales linear with flash size. The following structure of a chained superblock provides wear levelling of the first blocks and enables logarithmic mount time. For a more detailed explanation see [RD1] Chapter 4.

Functionality The first area of a flash contains two consecutive anchor blocks. The one page with the highest (i.e newest) number inside the anchor blocks is considered valid. This anchor page holds static information like file system version, number of blocks and the like as well as the address to the first jump pad located somewhere in a superblock area. These jump pads reference either a next pad or the superpage which is a page within a superblock area. This page holds frequently changed values like the address to the root node of the index tree and the area map.

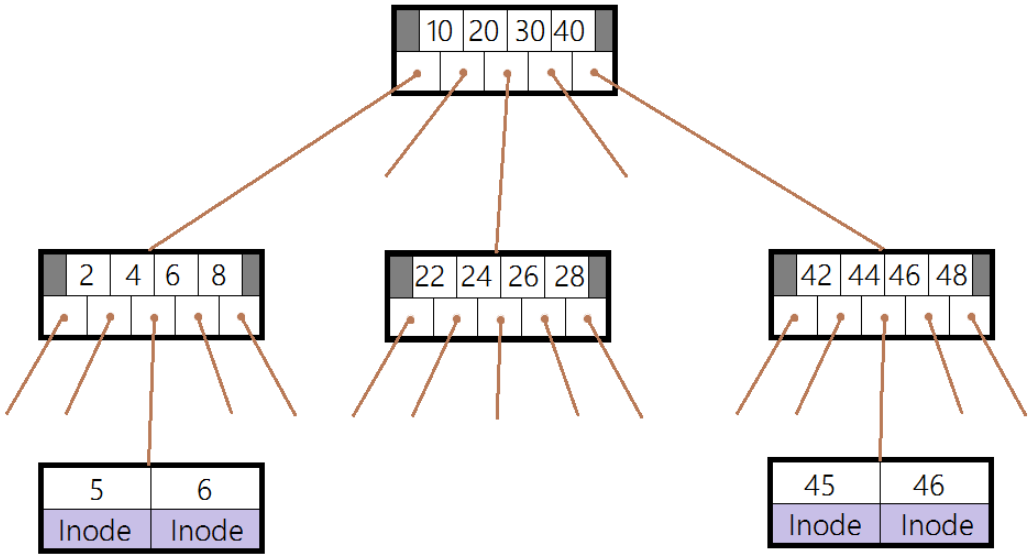
When a change of the index tree has been committed to flash, the new address of the root node has to be written to a new superpage. The address of the new superpage is then appended to the last jump pad block. If this block is full, it will be erased and the new address is written to a new jump pad block. This requires the next higher-level jump pad to append the address of the new jump pad block, and so on. When one of the anchor blocks is full, first the anchor page is written to the other block, and then the first block is deleted.

1.1.8 Tree Indexing

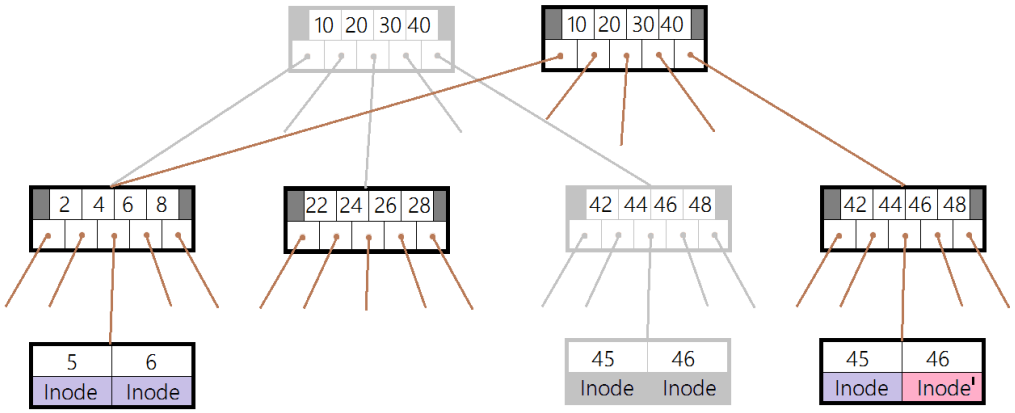
Overview B+Tree is a variation of the standard binary search tree with more than two keys per node and having all actual data only in leaves. This increases space for keys and therefore the fanout. Variable size, minimum tree height + 1[5]. Tree height is $log(n)$, and so is search, insert and delete $\mathcal{O}log(n)$. Because of *out-of-place-write*, with every change

[5] Check if true.

to a single inode, the whole path up to the root node has to be updated (see figure 1.2). To heavily decrease flash wear, a caching strategy is used. Any modification to the tree is held back in RAM as long as possible, until cache memory is running low. During cache flush all dirty nodes are committed to flash and freed in main memory. This caching can save 98% of flash operations and speed up access times as well (see [RD2]). Depending on cache flush strategy, all branch nodes are kept in cache while leaves are removed. This increases cache-hit ratio because there are less branches than leaves, but they branches more likely to be accessed again.



(a) Before



(b) After

Figure 1.2: B+-Tree after updating a single inode. The whole path has to be rewritten.

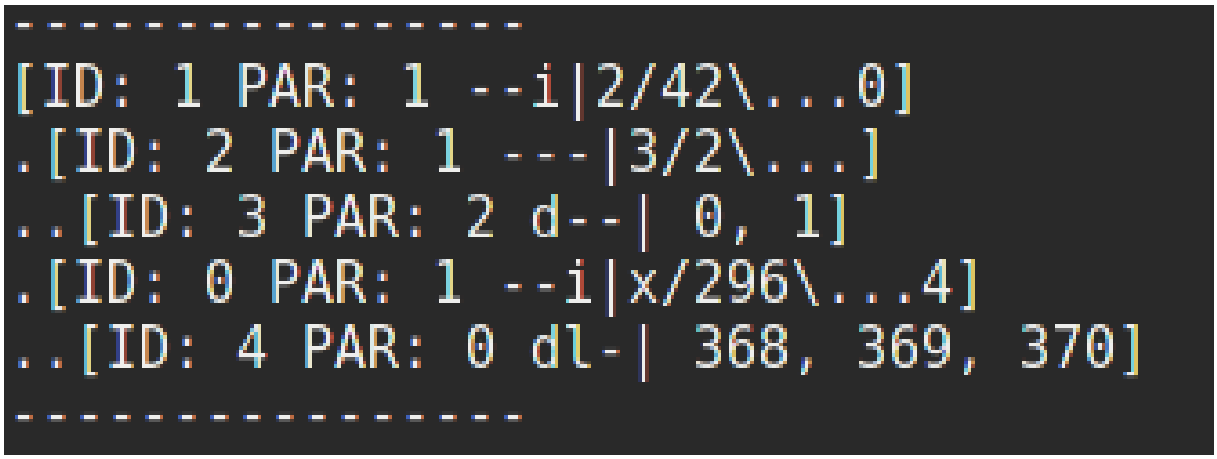


Figure 1.3: Example output of a cache content visualization. Notice the `dirty`, `locked` and `inherited lock` flags.

1.1.9 Inodes

Overview Can be file, directory and softlink. Points to data chunks in a way like `EXT3` does. See Table 1.2. It is possible to use pointer space for very small files or directories (14 * 4 Byte).

Table 1.2: Inode contents

Value	Description
`pInode_no`	Unique number of inode.
`pInode_type`	One of `file`, `directory` or `link`.
`permission`	Global read, write and execute permissions. Changeable via `chmod`.
`created`	Unix time stamp of file creation.
`modified`	Unix time stamp of last file write access.
`size`	Size of user data in bytes.
`reserved size`	Space user data is actually occupying in bytes (multiple of chunk size).
`direct pointer`	11 direct pointers to user data chunks.
`indirect pointers`	One of each first, second and third layer indirection pointers pointing to data chunks with more pointers.

1.2 Tests

This section is under development since the filesystem is unfinished. Meanwhile look at fancy pictures:

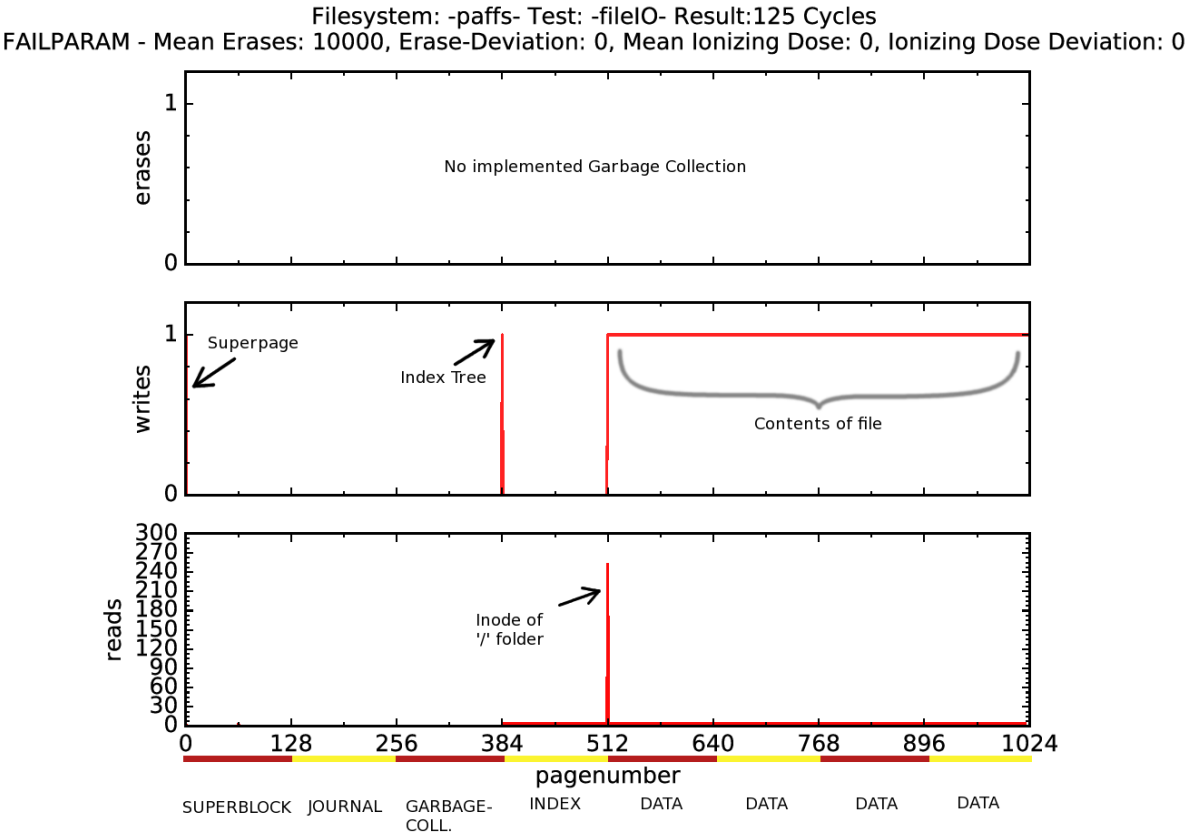


Figure 1.4: Status of simulated flash after 125 read/write cycles on a single file. Without a garbage collection, the number of rewrites is limited by flash size.
TODO: Change "contents of file" to "Dirty or used Pages of file"

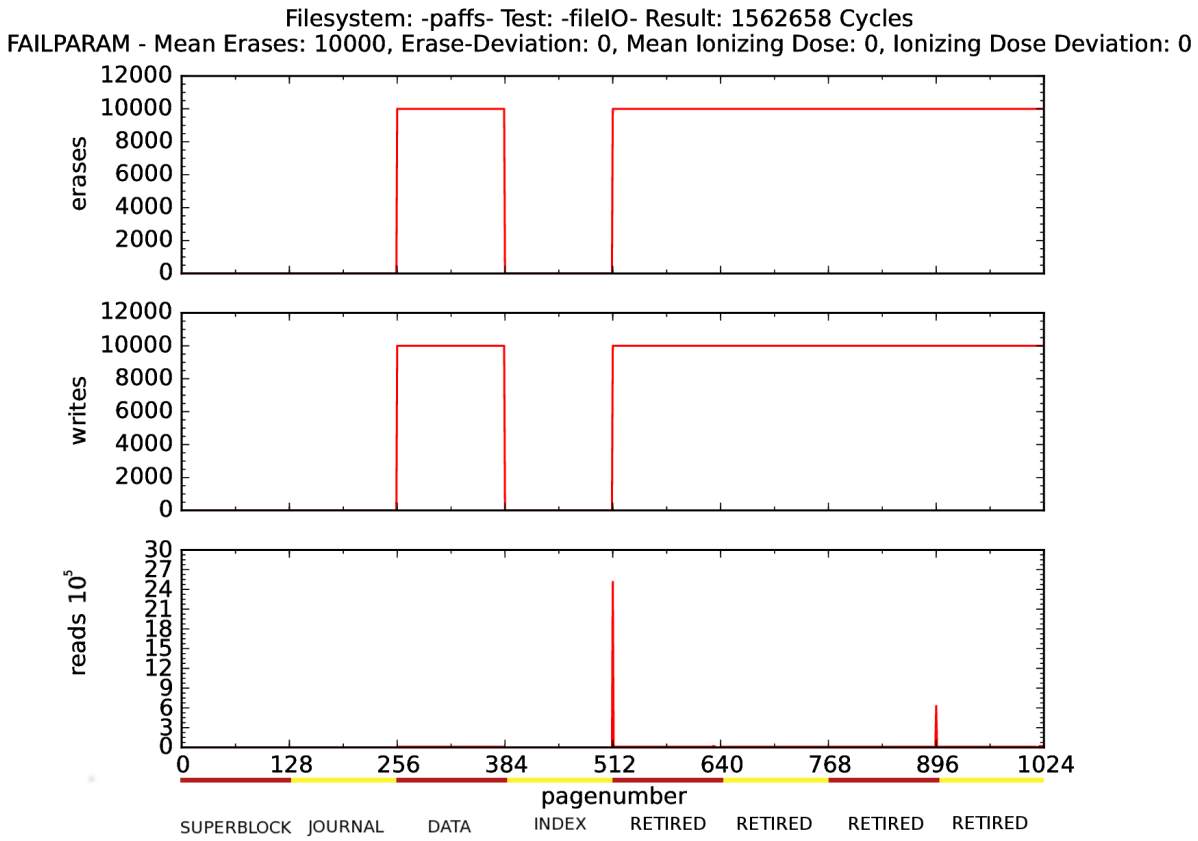


Figure 1.5: Status of simulated flash after 1562658 read/write cycles on a single file. This result is dependend on block erasure count.

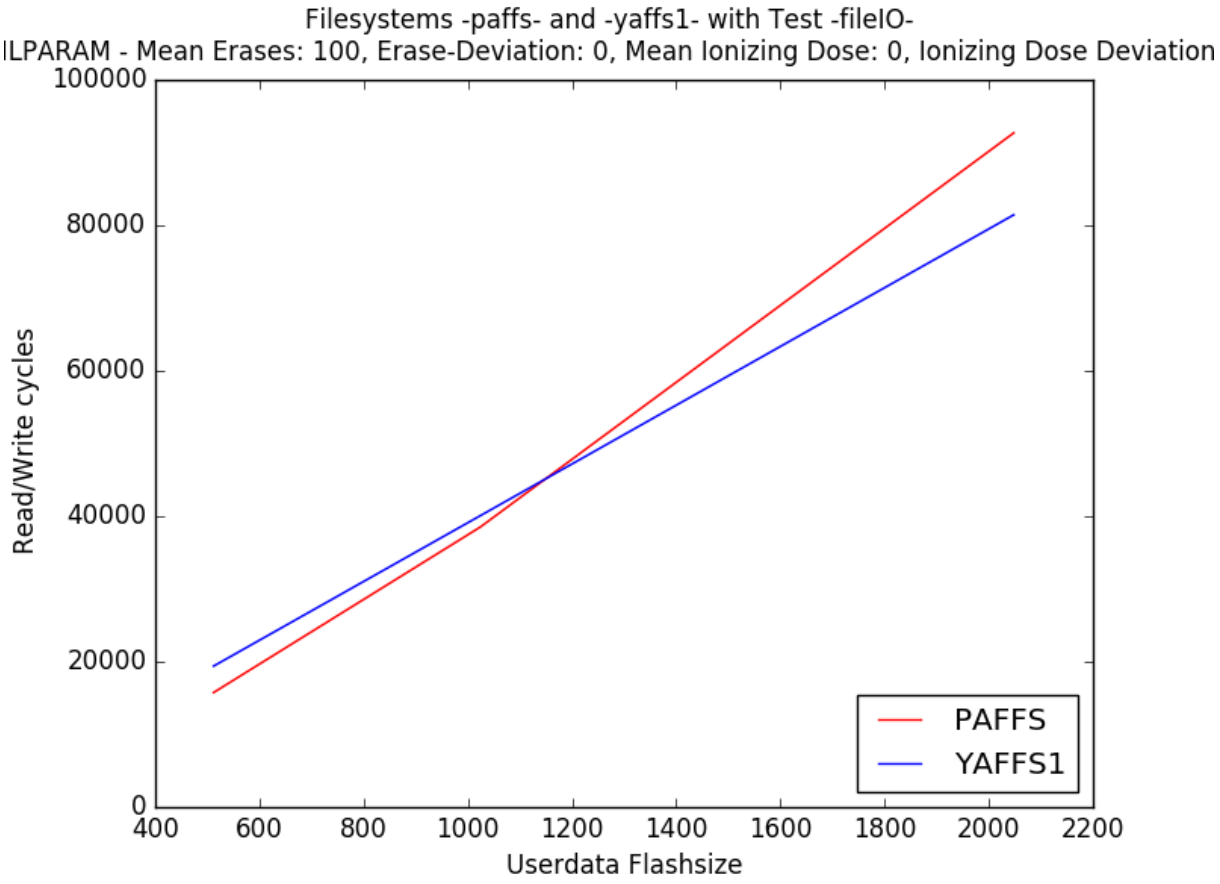


Figure 1.6: Comparison of wear levelling performance as a function of flash size between PAFFS and YAFFS1.

.1 TODOS

- file_append breaks Fs. → 1. write unittest provoking this behaviour, 2. repair this usecase.
- Add check if dirty in commit Area Summaries to enable read-only
- Implement simple ECC for correcting two biterrors.
- Read validation and error handling. If error is found, either repair it automatically or delay it until scrubbing? Scrubbing would be not very frequent, so on-the-fly would be appreciated.
- When ASCache runs full, some used pages cant be marked as dirty. Build sanity checker, scanning whole FS and marking unreachable pages as such.
- Implement dynamic number of uncommitted areas to superpage. (Maybe call it Checkpoint?)
- Merge DATA and INDEX areas to a single DATA type. Benefits: Less complexity, more efficient space usage, GC desperate mode for unmounting only. But maybe too overdone.
- TreeCache - full commit from time to time. (Maybe 'used' marker?)
- GC: Static Wear levelling, keep deletioncount in mind.
- Implement additional driver function for copying pages for optimizing.
- Replace mallocs to variable sized arrays. (Or standard container, b/c variable sized arrays are not legal in c++)
- Use Inode pointer space for very small files to speed up access and save space.
- Implement Inode and DATA Buffer to speed up everything. Note: Low priority and high effort.

optionEndOfDocument

- END OF DOCUMENT -