Avionics Systems

AVS-DOC-TN-0000
PAFFS Documentation

Issue: 0.1
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 Aeronautics Flash FS" and aims to use a minimum RAM footprint with the ability to manage multiple flashes. Most 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 1.1.4. 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 applicable to PAFFS. Areas and address mapping -> Section 1.1.1

1.1 Fundamental structures

1.1.1 Areas

Overview To separate between different Types of Data. Is the logical combination of two or more erase blocks.

- 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 an easy garbage collector (see fig. 1.1)
- AreaMap held in flash (but cached in RAM) translates between logical area n° and physical area n°
- AreaMap also keeps record of corresponding types and usage statistics for garbage collection

- `AreaStatus` can be one of `closed`, `active` and `empty`.

Table 1.1: Area types

Type	Description
Superblock	One superblock area is automatically 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 1.1.3
Index	The index areas will contain only tree nodes (including inodes). See Chapter 1.1.4.
Data	Data areas contain data chunks of files, directories and soft-links referenced by the index.
Journal	The single journal area (some when) will contain uncommitted changes to the index.

Area types

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 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 1.1.3), but is cached in RAM.
This area map grows linear in size with area count which is by itself depended by 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 $(3+2+16+32+32+32+32)/8 * (2048 * 1024 * 1024 * 1024)/(2048 * 1024)/2 = 19529728/2 = 9764864$ Byte $= 9,31$ MB RAM. Increasing the size of an area to 8 erase blocks would result in 2,3 MB usage, which would suit the requirements better.

Area summary The area map also contains a pointer to an area summary, which is an array of the per-page information if it was `free`, `used` or `dirty`. This area summary completely in RAM if the area is active, but is written to the first page in an area when it is getting closed. It is being used as information where to find the next free page and for the garbage collector to copy only valid chunks. The reference may only be valid if area status is active.

[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, $\approx 1 * 10^6$ blocks.

1.1.2 Garbage collection

Functionality The garbage collection needs at least one free area to copy valid data to. At first, one or more of the *closed* and therefore usually full areas are inspected (as their `AreaSummary`s are not cached) 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 physical areas swap their logical numbers. The new area is marked *active*, and the `AreaSummary` is held in RAM. If the previous way is not sufficient, *active* and therefore non-empty areas are chosen. This would only be done at the very end of space because it is not as efficient.

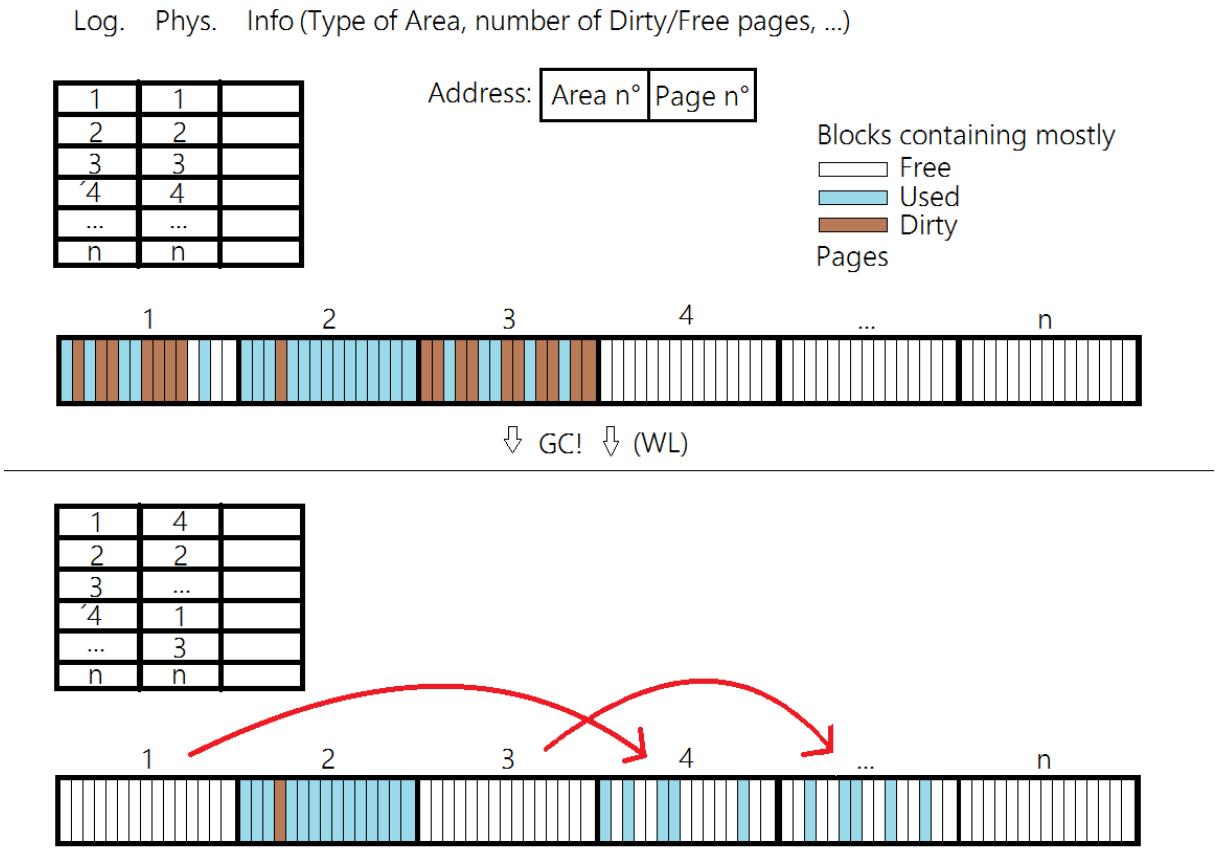


Figure 1.1: Basic process of garbage collection with areas

1.1.3 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) enormously in contrast to other places being 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.

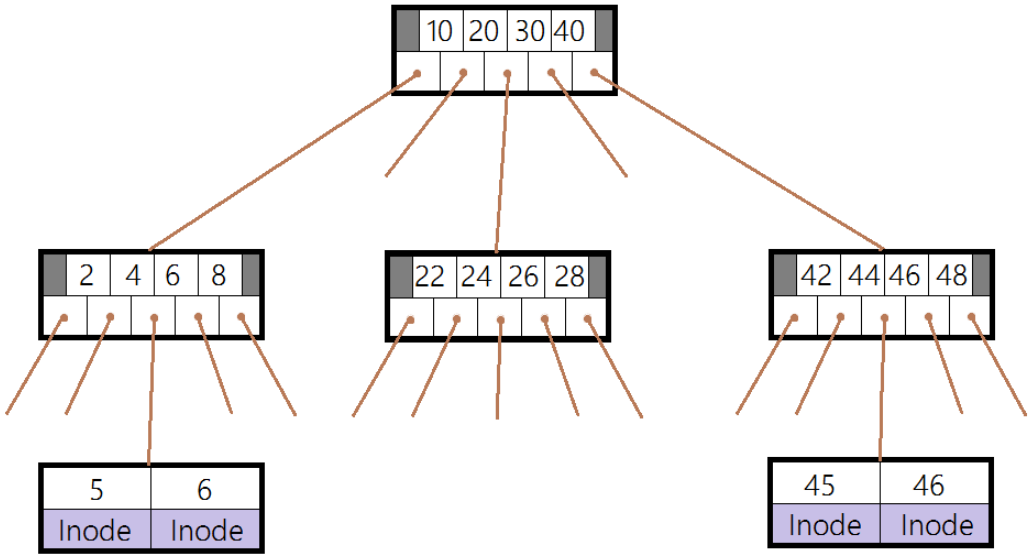
1.1.4 Tree Indexing

Overview B+Tree is a variation of the standard binary search tree with more keys per node. Another difference is having all actual data only in leaves. This increases space for keys and therefore the fanout. Variable size, minimum tree height + 1[3]. 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 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 main memory space is running low. Only now all nodes marked as dirty are committed to flash and freed in main memory. This caching can save 98% of flash operations and speeds up access as well (see [RD2]).

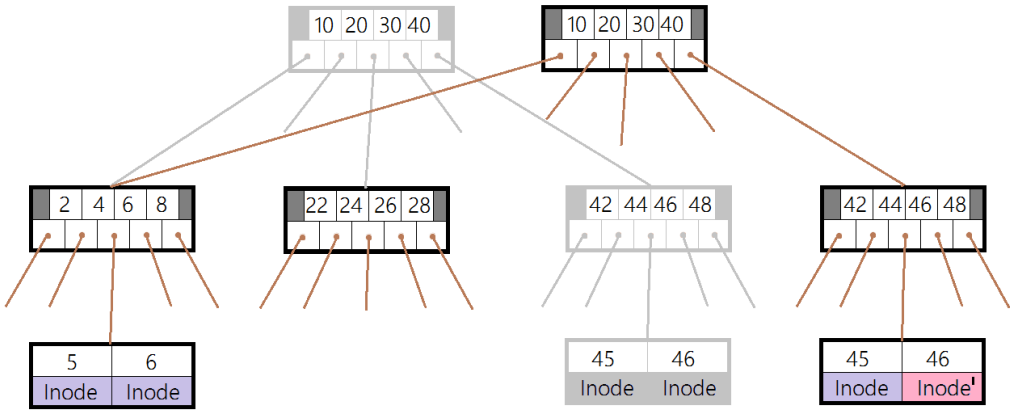
1.1.5 Inodes

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).

[3] Check if true.



(a) Before



(b) After

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

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.

optionEndOfDocument

- END OF DOCUMENT -