JFFS3 design issues

Artem B. Bityuckiy
dedekind@infradead.org

Version 0.25 (draft)

October 24, 2005

Abstract

JFFS2, the Journalling Flash File System version 2, is widely used in the embedded systems world. It was designed for relatively small flash chips and has serious problems when it is used on large flash devices. Unfortunately, these scalability problems are deep inside the design of the file system, and cannot be solved without full redesign.

This document describes JFFS3 – a new flash file system which is designed to be scalable.

Contents

1 JFFS2 overview

JFFS2, the Journalling Flash File System version 2 [1] is widely used in the embedded systems world. JFFS2 was originally designed for small NOR flashes (< 32MB) and the first device with JFFS2 file system was a small bar-code scanner. Later, when NAND flashes became widely used, NAND support was added to JFFS2. The first NAND flashes were also small enough, but grew in size very quickly and are currently much larger then 32MB (e.g., Samsung produces 2GB NAND flashes [4]).

JFFS2 has *log-structured* design, which basically means, that the whole file system may be regarded as one large log. Any file system modification (i.e., file change, directory creation, changing file's attributes, etc) is appended to the log. The log is the only data structure on the flash media. Modifications are encapsulated into small data structures called *nodes*.

So, JFFS2 is roughly a log, the log consists of nodes, each node contains a file system modification. And this is basically all JFFS2 file system is. It is very simple from the physical layout's standpoint. For more information about the design of JFFS2 and about the log-structured design, look at [1], [2], and [3].

It is not the goal of this chapter to delve into details of JFFS2 but still but it is still wanted to provide enough information to make it clear why JFFS2 has scalability problems and why JFFS3 is needed. To keep this chapter simple, terms *index* or *indexing information* are used.

The index is a crucial part of any file system as it is used to keep track of everything that is stored in the file system. For example, the index may help to quickly locate the addresses of physical blocks which correspond to the specified file at the specified offset, or it helps to quickly find all the directory entries in a specified directory and so on.

For example, in case of ext2, the inode table, the bitmap and the set of direct, indirect, doubly indirect and triply indirect pointers may be considered the index. In case of the FAT file system, the File Allocation Table may be considered as the index, etc.

In traditional file systems the index is usually kept and maintained on the media, but unfortunately, this is not the case for JFFS2. In JFFS2, *the index is maintained in RAM*, not on the flash media. And this is the root of all the JFFS2 scalability problems.

Of course, having the index in RAM JFFS2 achieves extremely high file system throughput, just because it does not need to update the index on flash after something has been changed in the file system. And this works very well for relatively small flashes, for which JFFS2 was originally designed. But as soon as one tries to use JFFS2 on large flashes (starting from about 128MB), many problems come up.

At first, it is obvious, that JFFS2 needs to build the index in RAM when it mounts the file system. For this reason, it needs to scan the whole partition in order to locate all the nodes which are present there. So, the larger is JFFS2 partition, the more nodes it has, the longer it takes to mount it.

The second, it is evidently that the index consumes some RAM. And the larger is the JFFS2 file system, the more nodes it has, the more memory is consumed.

To put it differently, if S is the size of the JFFS3 flash partition [1],

- JFFS2 mount time scales as $O(S)$ (linearly);
- JFFS2 memory consumption scales as $O(S)$ (linearly).

[1] Note, all the symbols used in this document are referred in the section 7

So, it may be stood that JFFS2 *does not scale*. But in spite of the scalability problems, JFFS2 has many advantages:

- very economical flash usage – data usually take as much flash space as it actually need, without wasting a lot space as in case of traditional file systems for block devices;
- admitting of "on-flight" compression which allows to fit a big deal of data to the flash; note, there are few file systems which support compression;
- very file system write throughput (no need to update any on-flash indexing information as it simply does not exist there);
- unclean reboot robustness;
- good enough wear-leveling.

It is also worth noting here, that there is a patch which is usually referred to as the "*summary patch*", that was implemented by Ferenc Havasi and was recently committed to the JFFS2 CVS. This patch speed up the JFFS2 mount greatly, especially in case of NAND flashes. What the patch basically does is that it puts a small "*summary*" node at the end of each flash erasable block. This node, roughly speaking, contains the copy of headers of all the nodes in this eraseblocks. So, when JFFS2 mounts the file system, it needs to glance to the end of each eraseblock and read the summary node. This results in that JFFS2 only needs to read one or few NAND pages from the end of each eraseblock. Instead, when there is no summary, JFFS2 reads almost all the NAND pages of the eraseblock, because the node headers are spread more or less evenly over the eraseblock.

Although the patch helps a lot, it is still a not scalable solution and it only relaxes the coefficient of the JFFS2 mount time liner dependency. Let alone that it does not lessen the memory consumption.

2 JFFS3 Requirements

The following are the main user level requirements JFFS3 have to meet.

R01 JFFS3 memory consumption must not depend on the size of the JFFS3 partition, the number of inodes in the file system, size of files, directories, and the like. Of course, JFFS3 must be able to use the advantage of the available RAM, but only for different kinds of *caches* which may be freed any time in case of memory pressure.

R02 JFFS3 have to provide very fast file system mount without the need to scan the whole flash partition.

R03 JFFS3 have to provide good flash wear-levelling.

R04 JFFS3 must guarantee that unclean reboots cannot cause any file system corruption.

R05 JFFS3 must provide good enough performance.

R06 Unlike JFFS2, JFFS3 must implement write-behind caching for better performance.

R07 JFFS3 must gracefully deal with different kinds of data corruptions, flash bits flipping, bad blocks which may appear dynamically, etc.

R08 In case of serious corruptions it should be possible to reconstruct the any data which were not damaged by means external tools like `ckfs.jffs3`.

R09 All the JFFS3 characteristics ought to vary not faster then $log(S)$, where S is the size of the JFFS3 partition. JFFS2-like linear dependencies are not acceptable.

R10 JFFS3 must support extended attributes.

R11 JFFS3 must support Access Control Lists feature (ACLs).

R12 JFFS3 have to support on-flight compression.

3 Introduction to JFFS3

The main idea how to fix in JFFS2 to make it scalable is *to move the index from RAM to flash*. Unfortunately, this requires complete JFFS2 redesign and re-implementation and the design of JFFS3 is largely different to the design of JFFS2. This section discusses the base JFFS3 design ideas without any detailed description.

3.1 Indexing problem

There is a large difference between block devices and flash devices in how they allow to update the contents of a sector. Block devices admit of so-called "*in-place updates*", i.e. the update may be written straight to the sector. Flash devices do not allow this unless the whole eraseblock has been erased before.

Obviously, it is unacceptable to erase the whole eraseblock each time a sector is updated. Instead, so-called "*out-of-place updates*" technique is usually used. This simply means, that no attempts to update the sector in-place is made, but instead, the update is written to some other sector and the contents of the previous sector is afterwords regarded as garbage.

This "out-of-place writes" property of flash devices assumes that JFFS3 also has log-structured design as in JFFS3 any update is written out-of-place. And it seems that this is natural for any flash file system to have log-structured design.

It is interesting to notice that in log-structured file systems for block devices (like the one described in [2]) not any update is "out-of-place". There are always some fixed-position sectors present. These sectors usually refer the file system index admitting of quick file system mount and they are updated in-place in these file systems.

But flash devices have limited number of erase cycles for each eraseblock and it is impossible to guaranty good wear-levelling if some eraseblocks are reserved for similar purposes. So, it is important that *in JFFS3 there are no in-place updates* as good wear-levelling is one of the main requirements to JFFS3 (see section 2).

The "out-of-place updates" property makes it difficult to maintain the index on the flash media. Figure 1 demonstrates why.

Suppose the index is kept and maintained on flash and it consists of 4 parts A, B, C, and D which refer each other: A refers B and C, B refers D, and C refers D. This means, that A contains the physical flash address of B and C and so on.

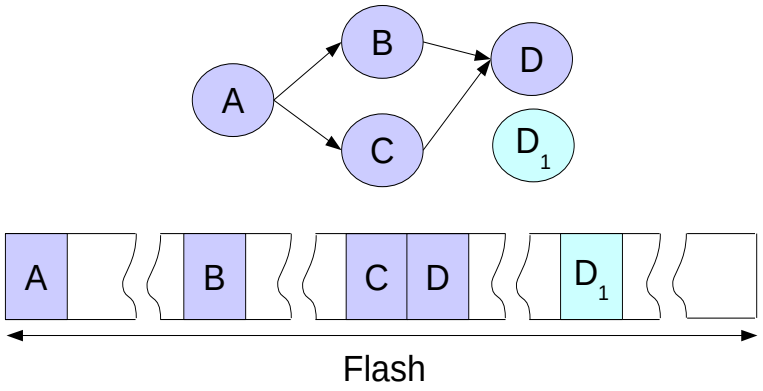

Figure 1: JFFS3 indexing problem example.

Suppose D should be updated. Since it is updated out-of-place, the newer version D_1 is written to some other place. But there are B and C which still refer D and they ought to be updated as well. And when they are updated out-of-place, A still refers the old B and C, and so on. Thus, it is not that trivial to store indexing information on flash.

3.2 Wandering trees

To address the above problem it is possible to use *wandering trees*. Figure 2 demonstrates how do wandering trees work.

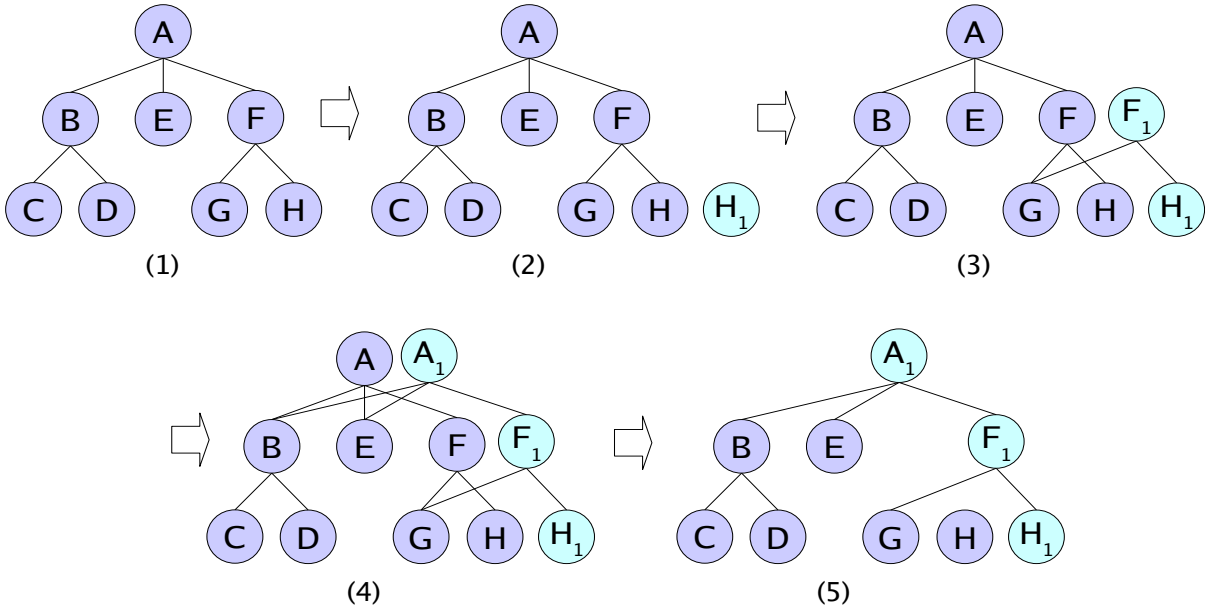

Figure 2: Wandering tree example.

1. Suppose that the index is a tree and it is stored and maintained on the flash media. The tree consists of nodes A, B, C, D, E, F, G, and H. Suppose node H should be updated.

2. At first, the updated version H_1 is written. Obviously, F still refers H.

3. Now the corresponding links in node F are changed and node F_1 is written to flash. F_1 refers H_1. But as F_1 is also written out-of-place, A still refers the old node F.

4. Finally, the new root node A_1 is written ant it refers F_1.

5. Nodes A, F, H are now treated as garbage and the updated tree contains nodes A_1, B, C, D, E, F_1, G, and H_1.

So, wandering trees is the base idea of how the indexing information is going to be maintained on the flash media in JFFS3. And it stands to reason that any tree may be called "wandering tree" if any update in the tree requires updating parent nodes up to the root. For example, it makes sense to talk about wandering Red-Black trees or wandering B^+-trees and so forth.

3.3 B$^+$-trees

JFFS3 uses B^+-trees and this subsection makes a short introduction to B^+-trees. There is a plenty of books where one may find more information.

The inexact definition of B-tree may be formulated as a balanced search tree where each node may have many children. The *branching factor* or the *fanout* defines the maximal number of node's children. While B-trees may contain both useful data and *keys* and *links* in non-leaf nodes, B^+-trees are B-trees which store data only in leaf nodes, while non-leaf nodes contain only keys and links.

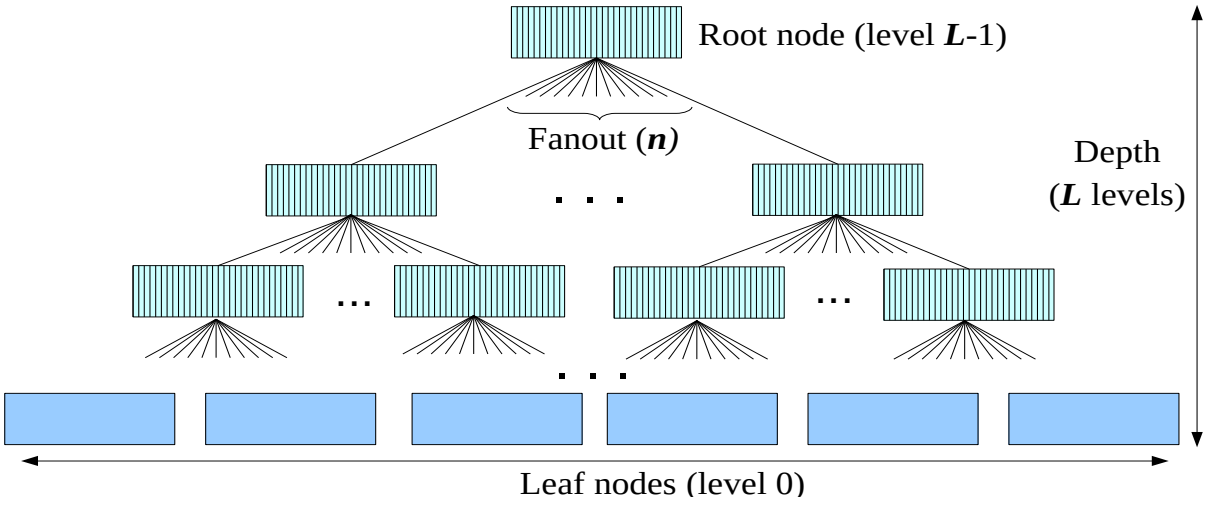

Figure 3: B^+-tree example.

Figure 3 demonstrates a B^+-tree with branching factor n and the number of level L. Note, that in JFFS3 levels are numbered starting from *leaf* nodes (level 0) and ending at the *root* node (level $L-1$).

Leaf nodes in the B^+-tree contain data which are indexed by keys. Non-leaf nodes do not contain data, but contain only the indexing information, namely, *keys* and *links*.

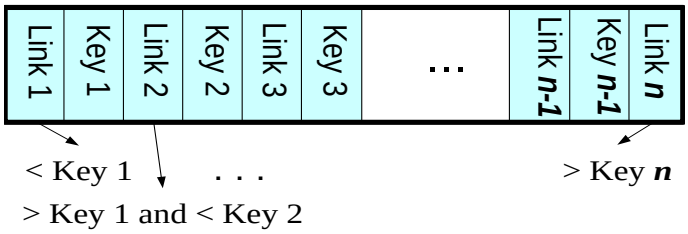

Figure 4: The structure of a non-leaf node in B^+-tree.

Figure 4 depicts the structure of a non-leaf node. There are n links and $n-1$ keys in the node. Links may point to either leaf nodes or other non-leaf nodes. In the former case, the leaf node will contain data which corresponds to the key which follows the link. In the latter case, the pointed non-leaf node (and the whole subtree with the root in this non-leaf node) will contain more keys in range $(Key\ 1, Key\ 2]$.

Keys are sorted in the ascending order in non-leaf nodes, so it is not that difficult to lookup data corresponding to any key. Furthermore, the tree is balanced, so the the number of lookup steps does not depend on the key.

When objects are inserted or removed from the tree, re-balancing may be needed. The tree is re-balanced by means of splitting nodes or merging them and there is a simple enough algorithm exists. Please, refer to Donald Knuth's books for more information about re-balancing B^+-trees.

B^+-trees are widely used when working with block devices (e.g., hard drives). Indeed, these devices have a fixed input/output unit size (usually referred to as a *sector*) and it is natural to use B^+-trees with node size multiple to the size of the sector in order to store information on such devices.

3.4 Indexing in JFFS3

The way how JFFS3 stores and indexes the file system is similar to the approach used by the *Reiser4* file system (see [5]). All the file system objects (inodes, files, directory entries, extended attributes, etc) are kept in one large B^+-tree. Effectively, the whole JFFS3 file system may be regarded as one large B^+-tree. This tree is further referred to just as "*the tree*".

Every object which is stored in the tree has a *key*, and the object is found in the tree by this key. To make it clearer what object keys are, the following is an example of how they may look like:

- file data key: {`inode number`, `offset`};
- directory entry key: {`parent directory inode number`, `direntry name hash`};
- extended attribute key: {`target inode number`, `xattr name hash`} and the like.

The following are terms which are used in JFFS3 to refer nodes of different levels in the tree:

- nodes of level 0 are *leaf nodes*;

- nodes of level 1 are *twig nodes*;
- nodes which are not the root, not leaf, and not twig are *branch nodes*;
- no-leaf nodes (i.e., the root, branch and twig) are *indexing nodes*.

Note, the same terminology (except indexing nodes) is used in the Reiser4 file system [5].

Non-leaf nodes are called "indexing nodes" because they contain only indexing information, nothing else. No file system data is kept in the indexing nodes. Indexing nodes have fixed size which is equivalent to the flash *sector* size.

It is important to note that somewhat unusual terminology is used in this document. The smallest input/output unit of the flash chip is called *sector*. Since JFFS3 mainly orients to NAND flashes, the sector is mostly the NAND page and is either 512 bytes or 2 Kilobytes. For other flash types the sector may be different. If flash's minimal input/output unit is very small (like one bit in case of NOR flash) there should be a layer which emulates larger sectors (say, 512 bytes).

In opposite to indexing nodes, leaf nodes have flexible size, just like nodes in JFFS2. So, roughly speaking, JFFS3 file system may be considered as JFFS2 file system (leaf nodes) plus indexing information (figure 5).

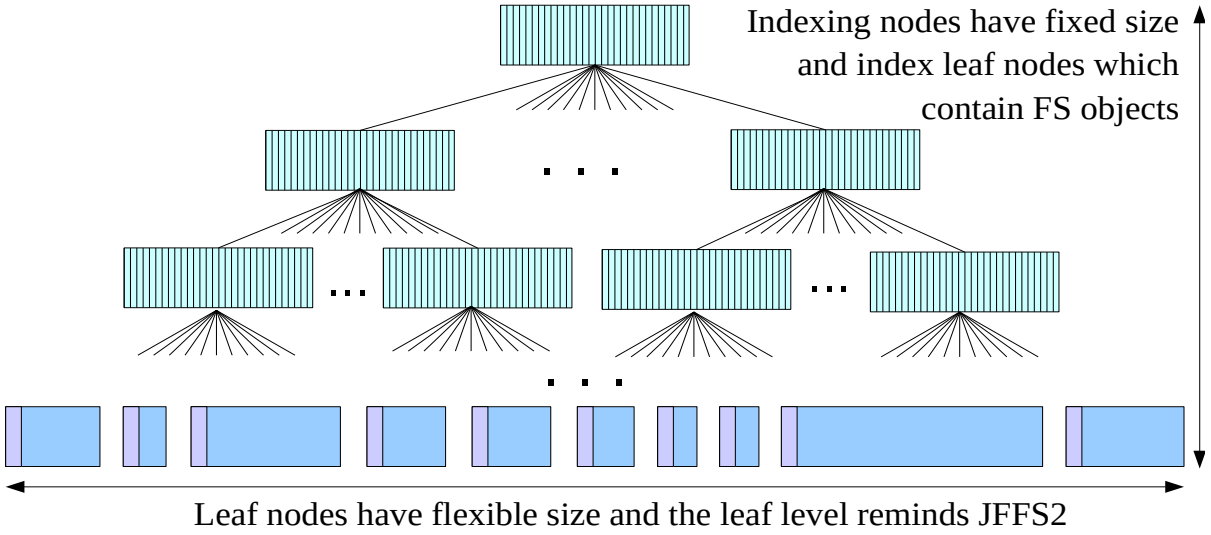

Figure 5: The JFFS3 tree.

Similarly to JFFS2, leaf nodes consist of *header* and *data*. The header describes the node data and contains information like the key of the node, the length, and the like. Node data contains some file system data, for example directory entry, file's contents, etc.

Leaf and indexing nodes are physically separated, which means that there are eraseblocks with only indexing nodes and with only leaf nodes. But of course, this does not mean that the whole flash partition is divided on two parts, this only means that the indexing and leaf nodes are not in one eraseblock. Figure 6 illustrates this.

Eraseblocks which contain only indexing nodes are called *indexing eraseblocks* and those with leaf nodes are called *leaf eraseblocks*.

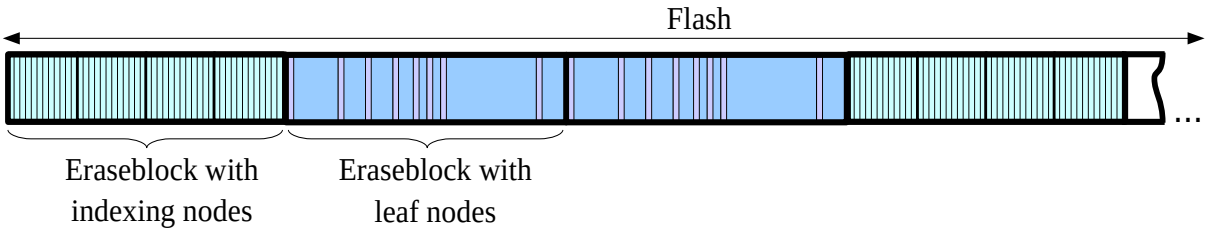

Figure 6: Leaf and indexing nodes separation illustration.

The depth of the tree depends on how many objects are kept on the file system. The more files, directories, etc are present on the file system, the deeper is the tree. Fortunately, the number of tree levels grows very slowly with the growing number of file system objects and the tree lookup scales as $O(log_n S)$ (logarithmically).

The following are the advantages of the JFFS3 indexing approach.

- Many different key assignment schemes may be used and this gives a flexibility in how objects are sorted in the tree. Thus, one may optimize JFFS3 for specific workloads by means of changing the format of the keys.
- The leaf nodes may be compressed, so JFFS3 admits on on-flight compression.
- In case of corruptions of the indexing information it is possible to re-create it by means of scanning leaf nodes' headers.
- There is a clear separation between data and indexing information. This implies that the indexing information and data separately may be cached separately, without overlapping in the same cache lines. This leads to better cache usage and is described very well in the Reiser4 paper [5].

3.5 The Journal

The JFFS3 tree is both B^+-tree and wandering tree. Any file system change implies that a new node is written to the flash media, which in turn, assumes that a number of indexing nodes must be updated. Namely, the whole path of indexing nodes up to the root node should be updated (see section 3.2).

Evidently, it is very expensive to update several indexing nodes on each file system change and *the journal* provides a mechanism to avoid this.

The journal consists of a set of eraseblocks (the *journal eraseblocks*) which do not have a fixed location on flash and are not contiguous on flash. Any flash eraseblock may be used as an journal eraseblock.

When something is changed in the JFFS3 file system, the corresponding leaf node is written to the journal, but the corresponding indexing nodes are not updated. Instead, JFFS3 keeps track of file system changes in RAM in a data structure called *the journal tree* (see figure 7).

When something is read from the file system, JFFS3 first glimpses at the in-RAM journal tree to figure out if the needed data is in the journal. If the data are there, the journal is read, otherwise JFFS3 performs the usual tree lookup (see figure 8).

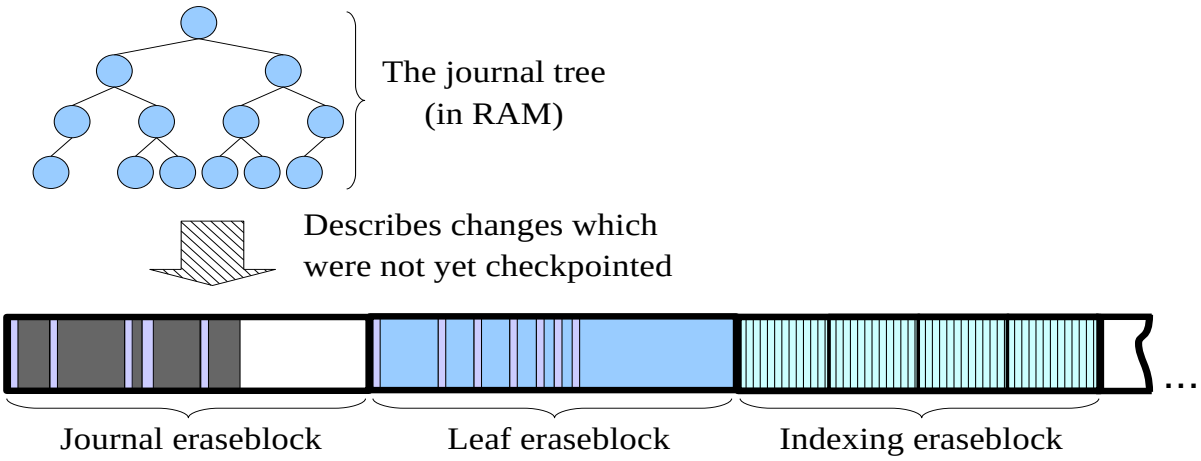

Figure 7: The JFFS3 journal.

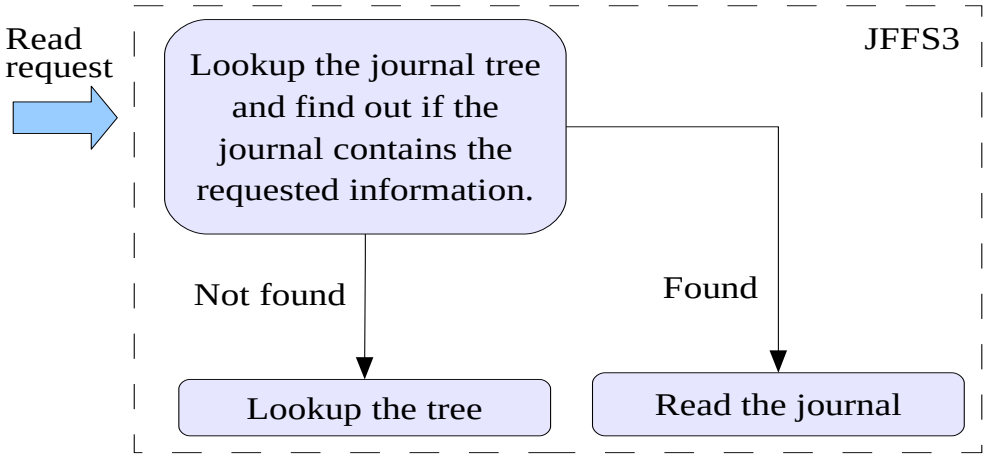

Figure 8: The read request processing in JFFS3.

The journal is *checkpointed* when it is full or in some other appropriate for JFFS3 time. This means, that the indexing nodes corresponding to the journal changes are updated and written to the flash. The checkpointed journal eraseblocks are then treated as leaf eraseblocks and new journal eraseblocks are picked by JFFS3 using the common JFFS3 wear-levelling algorithm.

The journal makes it possible to postpone indexing information updates to later and potentially more appropriate time. It also allows to merge many indexing nodes updates and lessen the amount of flash write operations.

When JFFS3 file system is being mounted, the journal should be read, "replayed" and the journal tree should be built. So, the larger is the journal the longer it may take to mount JFFS3. From the other hand, the larger is the journal, the more writes may be deferred and the better performance may be achieved. By the other words, there is a trade-off between the mount time and the performance and one may vary them by means of changing the size of the journal.

3.6 The superblock

The JFFS3 *superblock* is the data structure that describes the file system as a whole (i.e., the offset of the root node, the journal eraseblocks, etc). When the file system is being mounted, it first finds and reads the JFFS3 superblock.

In case of traditional file systems the superblock usually resides at a fixed position on the disk and may be found very quickly. Conversely, due to the "out-of-place write" flash property it is impossible to assign a fixed position for the JFFS3 superblock. Things are getting even more complex because of the need to provide good wear-levelling – it is incorrect to just reserve several erasable blocks for the superblock unless it is guaranteed that these eraseblocks will not be worn out earlier then the other eraseblocks.

We have the following two requirements that ought to be met in JFFS3:

- JFFS3 must be able to quickly find the superblock;
- the superblock management techniques must not spoil the overall flash wear levelling.

In the classical file systems the superblock usually contains mostly static data which is rarely updated and the superblock may have any size. In JFFS3, the superblock must be updated quite often (e.g., each time the journal is committed). This means that to lessen the amount of I/O, the JFFS3 superblock should be as small as it is possible, namely, one sector. And there is no reason to keep any static data in the superblock (e.g., the size of the file system, its version, etc). For static data, JFFS3 reserves the first eraseblock of the JFFS3 partition.

Thus, the following terms are used:

- *the static superblock* – contains only static data which are never changed by JFFS3; the static superblock resides at the *static eraseblock*; the static eraseblock is the first non-bad eraseblock of the JFFS3 partition; it is supposed that the contents of the static eraseblock may only be changed by external user-level tools;
- *the superblock* – contains only dynamic data, is changed quite often and requires special methods to deal with.

JFFS3 has rather complicated superblock management scheme which makes it possible to quickly find the superblock without any flash scanning when the file system is being mounted. This scheme provides good flash wear-levelling. Theoretically, the superblock lookup should take few milliseconds and scale as $O(log_2(S))$. For more detailed information about the superblock management scheme see section 4.1.

4 The superblock

4.1 The superblock management algorithm

To implement the superblock management scheme, JFFS3 reserves the second and the third good eraseblocks at the beginning of the flash partition (just next to the static eraseblock). These two eraseblocks are called *anchor eraseblocks*, or the *anchor area*.

Anchor eraseblocks contain references to *chain eraseblocks*. Chain eraseblocks may either refer other chain eraseblocks or the *super eraseblock* (see figure 9). Note, if there are k chain erase blocks, the anchor area will refer the the chain eraseblock 1, which will refer the chain eraseblock 2, which will refer the chain eraseblock 3 and so forth. The chain eraseblock k will refer the super eraseblock.

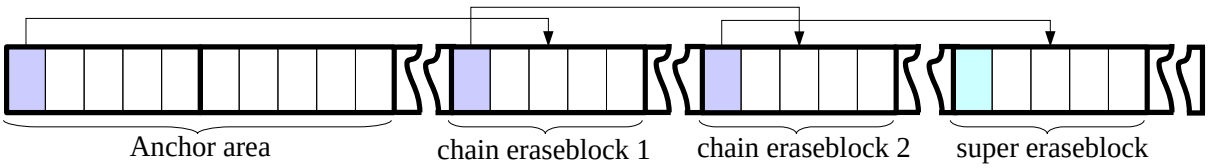

Figure 9: Eraseblocks involved to the superblock management scheme.

The super eraseblock contains the superblock which takes *one sector*. The chain eraseblocks contain references to the next chain eraseblock or to the super eraseblock.

The JFFS3 superblock management mechanism works as follows. Suppose there are k chain eraseblocks in the current superblock management scheme. The superblock updates are written to consecutive sectors of the super eraseblock. When the super eraseblock has no more empty sectors, new super eraseblock is picked, the superblock update is written to the new super eraseblock, and new reference is written to the chain eraseblock k.

Similarly, when there is no space in the chain eraseblock k, new chain eraseblock k is picked and the corresponding reference is written to chain eraseblock $k-1$, and so on. When there are no free sectors in the chain eraseblock 1, new chain eraseblock 1 is picked and the corresponding reference in the anchor area is updated.

Figure 10 presents the example of the superblock management scheme ($k = 2$).

1. Initially there are 2 chain eraseblocks (numbers 5 and 419) and the super eraseblock (number 501). There is a reference in the first sector of the anchor area which refers the chain eraseblock 1. The first sector of the chain eraseblock 1 refers the chain eraseblock 2, and the first sector of the chain eraseblock 2 refers the super eraseblock. The first sector of the super eraseblock contains the superblock.

2. After the superblock has been updated, the 2nd sector of the super eraseblock contains the valid copy of the superblock and the first sector contains garbage.

3. The superblock has been updated many times and the valid superblock is at the last sector of the super eraseblock while the other sectors of the super eraseblock contain garbage.

4. As there were no free sectors at the super eraseblock, new super eraseblock was chosen (eraseblock number 7) and the next superblock update was written to the first sector of the new super eraseblock. As the super eraseblock changed its position, the corresponding reference at the chain eraseblock 2 should have been updated. It was updated out-of-place and now the first sector of the chain eraseblock 2 is dirty while the second sector contains the valid reference to the new super eraseblock.

5. The superblock has been updated many times and the super eraseblock changed its position many times and it is currently at the eraseblock number 100. The

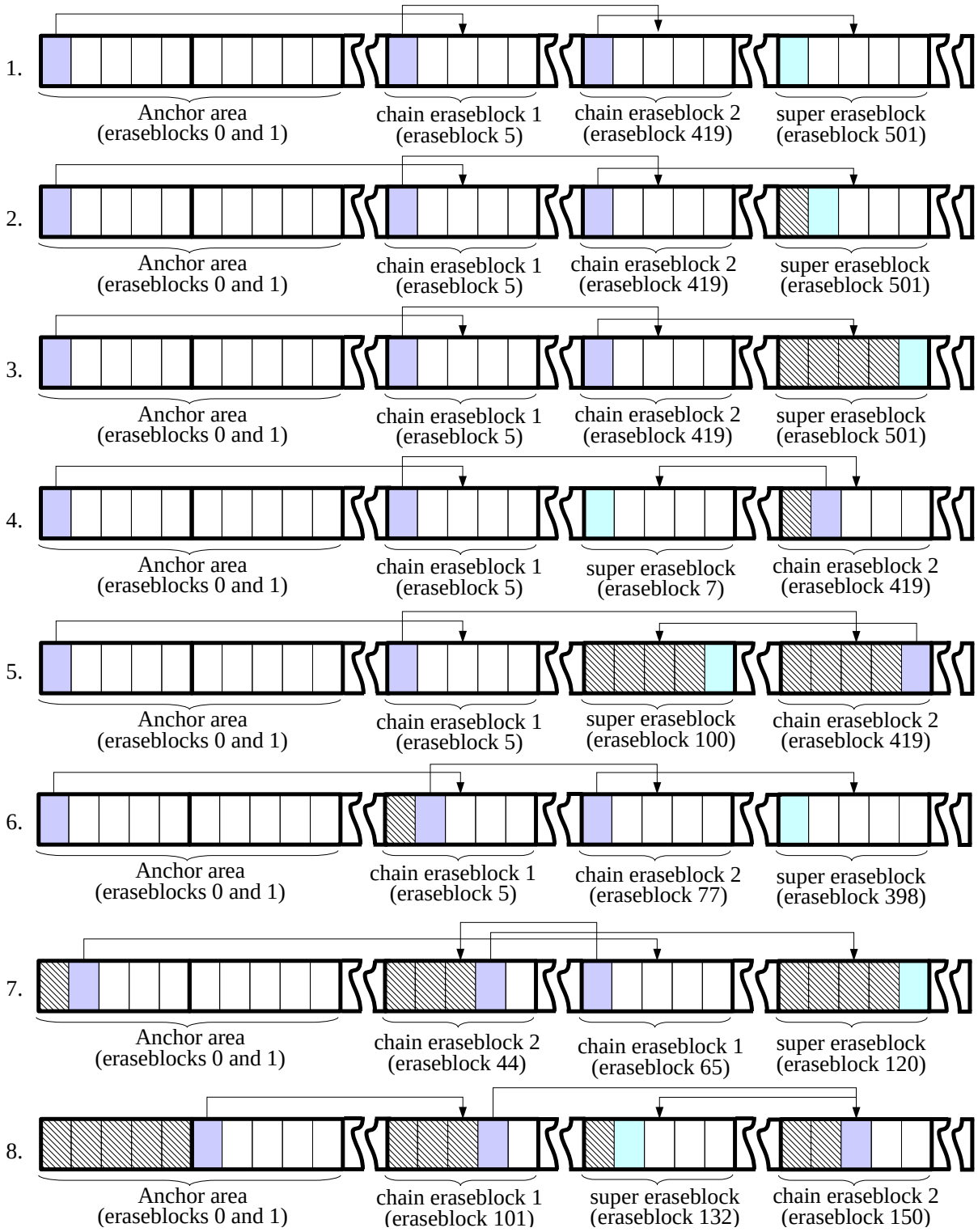

Figure 10: The superblock management example.

reference to the super eraseblock was also updated many times and at the moment the last sector of the chain eraseblock 2 contains the valid reference while the other sectors are obsolete. Similarly, the last sector of the super eraseblock contains valid superblock while the other sectors are obsolete.

6. When the next superblock update came, there were no free sectors at the super eraseblock and new super eraseblock was picked (eraseblock number 398) and the valid copy of the superblock is currently at the first sector of the eraseblock number 398, Also, there were no free sectors at the chain eraseblock 2 and new chain eraseblock 2 was found (eraseblock number 77), so the first sector of the eraseblock 77 contains the valid reference to the super eraseblock. Since the chain eraseblock 2 changed its position, the corresponding reference at the chain eraseblock 1 was updated and at the moment the second sector of the chain eraseblock 1 contains the valid reference to the chain eraseblock 2 while the first sector is dirty.

7. And analogously, after many superblock updates, the chain eraseblock 1 was updated many times and when it became full it changed its position. Sure, the chain eraseblock 2 and the super eraseblock changed their positions many times as well. So, at the moment, the chain eraseblock 1 is at the eraseblock number 65, the chain eraseblock 2 is at the eraseblock 44 and the super eraseblock is at the eraseblock 120. When the chain eraseblock 1 changed its position, the corresponding reference at the anchor area was updated and currently the second sector of the anchor eraseblock 1 contains the valid reference to the chain eraseblock 1 while the firs sector is dirty.

8. And even more superblock updates happened. The anchor area was updated many times. When there were no free sectors at the anchor eraseblock 1, the anchor eraseblock 2 was used. So, at the moment, the valid reference to the chain eraseblock 1 is at the first sector of the anchor eraseblock 2. From now on, the first anchor eraseblock may be erased and may be used again when the second anchor eraseblock becomes full.

The following are important notes about the JFFS3 superblock management.

- The superblock takes one sector so the super eraseblock may be updated at most N times (N is the number of sectors in the eraseblock).

- In case of NAND flash, the sector is the real minimal physical input/output unit, so only N updates are possible in the anchor eraseblock and in the chain eraseblocks. But if the real input/output unit is smaller then the sector (i.e., if JFFS3 works on top of NOR flash) the advantage of this may be used and more references may be packed into one anchor or chain eraseblock.

- When JFFS3 picks new chain/super eraseblock, the common JFFS3 wear-levelling scheme is utilized.

- Anchor area has 2 eraseblocks in order to ensure the tolerance to unclean reboots – one anchor eraseblock may be safely erased while the other is being used.

- When a new reference is written to anchor/chain eraseblocks, the previous reference becomes dirty and on mount JFFS3 should find the valid reference. To facilitate this, each reference has its version number. Each subsequent reference has higher version then the previous. Hence, JFFS3 may use the binary search algorithm to quickly find the valid reference.

- As unclean reboot may happen anytime, no anchor/chain/super eraseblocks are erased before the whole chain has been updated. This makes it possible to recover from unclean reboots if they happen while the chain of the superblock-related eraseblocks is being updated.

4.2 The length of the chain

The number of required eraseblocks in the superblock management scheme depends on the size of the JFFS3 partition. The larger the partition, the more levels are needed. This is determined by the need to ensure that the anchor area is not worn out earlier then the rest of the JFFS3 partition.

Let's Denote the number of required chain eraseblocks plus one (the super eraseblock) m and calculate m assuming the worst case scenario: any file system data update requires the superblock update. This would correspond to synchronous JFFS3 operation mode with zero-length journal.

Obviously, what is wanted is to be sure that the anchor area is not worn out earlier then the data area, i.e. the following inequality should be true:

$$\frac{T_A}{T_D} \geqslant 1, \tag{1}$$

where T_A is the period of time of the total anchor area wear and T_D is the period of time of the total data area wear. Note, the whole JFFS3 partition excluding the static superblock and the anchor area is referred to as the *data area*.

T_A and T_D may be expressed as

$$T_A = \frac{2D}{R_A},$$
$$T_D = \frac{(M-2) \cdot D}{R_D},$$

where R_A is the average rate of the anchor area updates (sectors per second), R_D s the average rate of the data area updates, D is the maximum number of flash eraseblock erase cycles, and M is the total number of non-bad eraseblocks on the JFFS3 partition. Hence,

$$\frac{T_A}{T_D} = 2 \cdot \frac{R_D}{(M-2) \cdot R_A}. \tag{2}$$

If $m = 0$, i.e., there are no chain/super eraseblocks and the superblock is stored in the anchor area, then taking into account (2) and that in this case $R_A = R_D = R$, we have

$$\frac{T_A}{T_D} = 2 \cdot \frac{1}{(M-2)}.$$

Suppose $m = 1$. i.e., there are no chain eraseblocks and only the super eraseblock is used. In this case each file system data update will require (a) the superblock update and (b) the anchor area update. Therefore, providing N is the number of sector in the eraseblock, the anchor area will be written N times less frequently then when $m = 0$ and

the data area will be written 2 times more frequently then when $m = 0$. This means, that $R_A = R/N$ and $R_D = 2R$ and from (2) we have

$$\frac{T_A}{T_D} = 2 \cdot \frac{2N}{M-2}.$$

When $m = 2$, i.e. the chain eraseblock 1 and the super eraseblock are used, the anchor area will be written N^2 times less frequently, while the data area will be written $2 + 1/N$ times more frequently then when $m = 0$ (one superblock update on each file system update and one chain eraseblock 1 update per N superblock updates). Therefore, $R_A = R/N^2$ and $R_D = (2 + 1/N) \cdot R$ and from (2) we have

$$\frac{T_A}{T_D} = 2 \cdot \frac{2N^2 + N}{M-2}.$$

For $m = 3$, analogously,

$$\frac{T_A}{T_D} = 2 \cdot \frac{2N^3 + N^2 + N}{M-2},$$

and for $m = 0, 1, 2, \ldots$

$$\frac{T_A}{T_D} = 2 \cdot \frac{2N^m + N^{m-1} + \ldots + N}{M-2}.$$

Consequently, from (1) we have the following inequality:

$$2 \cdot \frac{2N^m + N^{m-1} + \ldots + N}{M-2} \geqslant 1,$$

or neglecting the minor components,

$$\frac{4N^m}{M} \geqslant 1,$$

or

$$m \geqslant log_N \frac{M}{4}. \tag{3}$$

Thus, form (3) it is obvious that the JFFS3 superblock management scheme scales logarithmically.

Table 1 shows the value of m for different types of existing NAND flashes.

Type	**Size**	**Sect. size**	**M**	**N**	**m**
Toshiba TC58DVM92A1FT	64MB	16KB	4096	32	2
Toshiba TH58NVG1S3AFT05	512MB	128KB	4096	64	2
ST Micro NAND08G-B	1GB	128KB	8192	64	2
Samsung K9K1G08X0B	2GB	128KB	16384	64	2

Table 1: The length of the JFFS3 superblock management chain for different types of existing NAND flashes.

Note, providing that $N = 64$, $m = 3$ is enough to guarantee acceptable anchor area wear leveling for up to 128GB flash, $m = 4$ – for up to 8TB flash (the inequality 3).

4.3 The superblock search

To find the superblock during mount, JFFS3 finds the valid reference in the anchor eraseblocks, then finds the valid reference in chain erase blocks 1, 2, ..., $m-1$, and finally finds the valid superblock in the super eraseblock. Since JFFS3 assigns versions to sectors of anchor/chain/super eraseblock and the versions are increased by one on every update, the binary search algorithm may be used to find the valid sector.

The valid reference in the anchor area may be found after $log_2(2N)+2$ steps (one step involves one sector read operation), the reference in chain/super eraseblocks – after $log_2(N)+2$ steps. Thus, to find the superblock, JFFS3 must read

$$S = 2(m+1) + log_2(2N) + m \cdot log_2(N)$$

sectors.

Table 2 contains the approximate superblock search time for different existing NAND flashes. [2]

Type	**Size**	**N**	**m**	**Sect. read**	**S**	**SB find**
Toshiba TC58DVM92A1FT	64MB	32	2	$\sim 50\mu$s	22	$\sim$1.1ms
ST Micro NAND08G-B	1GB	64	2	$\sim 130\mu$s	25	$\sim$3.3ms
Samsung K9K1G08X0B	2GB	64	2	$\sim 70\mu$s	25	$\sim$1.6ms

Table 2: The superblock search time for different existing NAND flashes.

For larger flash chips which would utilize the superblock management scheme with $n = 3$ (no such flashes exist at the moment), the superblock search time would be about 4.3ms, providing the flash characteristics are the same as ST Micro's (see table 2).

Note, the calculated SB search time doesn't contain the ECC/CRC checking overhead as well as any other CPU overhead.

5 Issues/ideas/to be done

This section is a temporary list of issues which should be solved, ideas which should be thought and analyzed more or things which were thought about but are not yet described in this document.

The following is the list of things which should be thought about more.

1. Quota support. Will quota be supported? How will it look like – lust generic linux quota or something better?

2. Transactions:
   `transaction_open()/do_many_fs_modifications()/transaction_close()` semantics? Reiser4 pretends to support this via special `sys_reiser4()` syscall. Would be nice.

[2] the calculated superblock search time doesn't contain the ECC/CRC checking overhead as well as any other CPU overhead.

3. Does it make sense to "compress" keys. For example, if there are many consecutive keys with the same prefix, we may write this prefix only once. Will there be some problems when these compressed keys are changed and cannot be compressed any longer? Will this need some horrid tree and re-balancing? The idea of "extents" is similar.

4. How can one select the compression mode on the per-inode basis? Xattrs with some reserved name?

5. ACLs... Will they be implemented via xattrs or it will be something tricker/better?

The following is the list of topics which should be highlighted in this document as well.

1. Garbage collection.

2. Caching, write-behind cache.

3. An assumed flash model and the model of interactions between JFFS3 and the flash I/O subsystem.

4. How the track of eraseblocks will be kept? Space accounting, good/bad, erase count?

5. The wear-levelling algorithms.

6. The format of keys.

7. Branch nodes' links are sector numbers, twig nodes' links are absolute flash offsets. So, the length of twig and branch keys are different and branches have greater fanout.

8. Different optimizations may be achieved by means of changing the format of keys. So, JFFS3 should be flexible in this respect and have a mechanism to change/select the formats of keys.

The following is the list of ideas which were thought about but are not yet in the document.

1. If the compression is disabled for an inode, then its nodes are (`PAGE_SIZE` + header size) in size, i.e., they do not fit into integer number of flash sectors. For these nodes we may keep the header in the OOB area. In this case we should not mix compressed nodes and uncompressed nodes in one eraseblock.

2. For large files which are mostly read-only, we may fit more then one page of date in one node. This will mace compression better. When the file is read, all the uncompressed pages are propagated to the page cache, like in the zisofs file system.

3. If there are few date in the superblock, we may keep this data in the root node. In this case the root will have smaller fanout then branches.

The "to do" list.

1. Review the "definitions section". Add more terms there, e.g., checkpointing, quota, branching factor, indexing, leaf, journal, anchor, chain, super eraseblocks.

6 Definitions

1. **Access Control List, ACL** – a modern mechanism to control accesses to files, see [6] for more details.

2. **Anchor eraseblock, anchor area** – the second and the third *good* eraseblocks of the JFFS3 partition which are reserved for the superblock management.

3. **Branch node** – any node that is not leaf, not twig and not root.

4. **Chain eraseblock** – an eraseblock containing references to other chain eraseblocks or to the super eraseblock. Chain eraseblocks facilitate quick SB searching and are part of the JFFS3 superblock management scheme (see section 4.1).

5. **Directory entry** – basically associates a name with an inode number. Directories may be considered as a list of directory entries.

6. **Erasable block, eraseblock** – the minimal erasable unit of the flash chip from the JFFS3's viewpoint.

7. **Data area** – the whole JFFS3 partition excluding the static superblock and the anchor eraseblocks.

8. **Dirty sector** – a sector with data which is not valid any longer, recycled by Garbage Collector.

9. **Garbage Collector,** – a part of any Flash File System which is responsible for recycling dirty space and producing free eraseblocks.

10. **Indexing information, index** – data structures which do not contain files, directories, extended attributes or whatever is seen by user, but instead, keep track of this data. For example, indexing information allows to quickly find all the directory entries for any specified directory. I case of the FAT file system, the File Allocation Table is the index, in case of ext2 the inode table, the bitmap and the set of direct, indirect, doubly indirect and triply indirect pointers may be considered as the index. In JFFS3, the indexing nodes may be referred to as the index.

11. **Journal** – contains all the recent JFFS3 changes. Its purpose is to accumulate a bunch of JFFS3 file system changes and and to postpone updating the index. See section 3.5 for more information.

12. **Journal eraseblock** – an eraseblock containing the journal data.

13. **Journal tree** – an in-memory tree referring Journal nodes which were not committed so far. For more information see section 3.5.

14. **Leaf node** – any node from the leaf level of the tree (level 0). Leaf nodes contain only data and do not further refer other nodes. For more information see section 3.4.

15. **Node** – a pile of the tree (the tree consists of nodes). There are different types of nodes in JFFS3. For more information see section 3.4.

16. **Out-of-place updates, out-of-place writes** – a sort of data update when the updated version is not written to the same physical position, but instead, written to some other place and the previous contents is treated as garbage afterwords. Opposite to in-place updates.

17. **Sector** – the smallest writable unit of the *flash chip*, from the JFFS3's viewpoint. E.e. the NAND page in case of NAND.

18. **Static eraseblock** – the fist good erasable block of the JFFS3 partition where the per-file system static data is stored. JFFS3 may only read it and it is created/changed by external formatting tools.

19. **Superblock** – a data structure describes the whole JFFS3 file system. Only dynamic data is stored in the superblock, all the static data is kept in the static superblock.

20. **Tree** – the main object JFFS3 design revolves about. The JFFS3 tree is the wandering B^+-tree where all the file system stuff (files, directories, extended attributes, etc) is stored.

21. **Twig nodes** – reside one level upper then leaf nodes (level 1).

22. **xattr** – extended attributes, associate name/value pairs with files and directories, see attr(5) Linux manual pages for more information.

7 Symbols

The following is the list of symbols which are used to denote different things thought this document.

- D – number of guaranteed erases of flash eraseblocks (typically $\sim 10^5$ for NAND flashes);
- L – the number of levels in the tree.
- m – the number of eraseblocks used in the superblock management scheme without the anchor eraseblocks, i.e. the number of chain eraseblocks plus one (the super eraseblock).
- M – the total number of non-bad eraseblocks on the JFFS3 partition.
- n – the branching factor (fanout) of the tree.
- N – the number of sectors per eraseblock.
- S – the size of the JFFS3 flash partition.

8 Abbreviations

1. **ACL** – Access Control List
2. **ECC** – Error Correction Code
3. **CRC** – Cyclic Redundancy Check
4. **JFFS2** – Journalling Flash File System version 2
5. **JFFS3** – Journalling Flash File System version 3
6. **MTD** – Memory Technology Devices
7. **RAM** – Random Access Memory
8. **VFS** – Virtual File System

9 Credits

The following are the people I am grateful for help (alphabetical order):

- **David Woodhouse** <dwmw2@infradead.org> – the author of JFFS2, answered a great deal of my questions about MTD, JFFS2 and JFFS3 design approaches, English (e.g., "how to express this correctly in English"), etc.
- **Joern Engel** <joern@wohnheim.fh-wedel.de> – discussed a lot of JFFS3 design aspects with me. Some ideas described in this document were pointed by Joern.
- **Nikita Danilov** <nikita@clusterfs.com> – used to work in Namesys and implemented ReiserFS and Reiser4 file systems. Nikita answered many of my questions about Reiser4 FS internals.
- **Thomas Gleixner** <tglx@linutronix.de> – helped me with many MTD-related things, especially concerning flash hardware and low-level flash software. Proposed some ideas which I'm exploiting in the JFFS3 design.
- **Victor V. Vengerov** <vvv@oktetlabs.ru> – my colleague from OKTET Labs who spent a lot of time discussing the JFFS3 design approaches with me and suggested many interesting ideas. He also reviewed some of my writings.

10 References

1. JFFS : The Journalling Flash File System,
   http://sources.redhat.com/jffs2/jffs2-html/
2. The Design and Implementation of a Log-Structured File System,
   http://www.cs.berkeley.edu/~brewer/cs262/LFS.pdf
3. Who wants another filesystem?,
   http://cgi.cse.unsw.edu.au/~neilb/conf/lca2003/paper.pdf

4. Samsung Flash memory products,
   http://www.samsung.com/Products/Semiconductor/Flash/index.htm

5. Reiser4 File System, http://www.namesys.com/

6. POSIX Access Control Lists on Linux
   http://www.suse.de/~agruen/acl/linux-acls/