An Improved B+ Tree for Flash File Systems

Ferenc Havasi

Department of Software Engineering, University of Szeged
Árpád tér 2. 6722, Szeged, Hungary
havasi@inf.u-szeged.hu

Abstract. Nowadays mobile devices such as mobile phones, mp3 players and PDAs are becoming evermore common. Most of them use flash chips as storage. To store data efficiently on flash, it is necessary to adapt ordinary file systems because they are designed for use on hard disks. Most of the file systems use some kind of search tree to store index information, which is very important from a performance aspect. Here we improved the B+ search tree algorithm so as to make flash devices more efficient. Our implementation of this solution saves 98%-99% of the flash operations, and is now the part of the Linux kernel.

1 Introduction

These days mobile devices such as mobile phones, mp3 players, PDAs, GPS receivers are becoming more and more common and indispensable, and this trend is expected to continue in the future. New results in this area can be very useful for the information society and the economy as well.

Most of the above-mentioned devices handle data files and store them on their own storage device. In most cases this storage device is flash memory [3]. On smart devices an operating system helps to run programs that use some kind of file system to store data. The response time (how much time is needed to load a program) and the boot time (how much time is needed to load all the necessary code and data after power up) of the device both depend on the properties of the file system. Both of these parameters are important from the viewpoint of device usability.

Here we introduce a new data structure and algorithm designed for flash file systems that work more efficiently than the previous technical solutions. We begin by introducing the workings of flash chips, then describe the previously most prevalent Linux flash file system, namely JFFS2. Its biggest weakness is its indexing method: it stores the index in the RAM memory, not in the flash memory. This causes unnecessary memory consumption and performance penalties. Next, we introduce the new method in a step-by-step fashion, with an efficient combination of storing the index in memory and flash after taking factors like data security and performance into account.

This new solution has been implemented in the UBIFS file system by our department in cooperation with Nokia, and it is now an official part of the Linux kernel.

I. Černá et al. (Eds.): SOFSEM 2011, LNCS 6543, pp. 297–307, 2011.

2 How the Flash Memory Works

Flash memory is a non-volatile computer memory that can be electrically erased and reprogrammed. One of the biggest limitations of flash memory is that although it can be read or programmed one byte or word at a time in a random-access fashion, it must be erased one "block" at a time. The typical size of a block is 8-256K.

The two common types of flashes are NOR and NAND flash memories, whose basic properties are summarized in the following table [8].

	NOR	NAND
Read/write size	Can read/write bytes individually	Can read/write only pages (page size can be 512 or 2048 bytes)
I/O speed	Slow write, fast read	Fast write, fast read
Erase	Very slow	Fast
XIP (Execute in Place)	Yes	No
Fault tolerance, detection	No	Yes
Price/size	Relatively expensive	Relatively cheap

One of the drawbacks of the flash system is that its erase block can be erased about 100,000 times, and afterwards the chip will be unstable. This is why most of the ordinary file systems (FAT, ext2/3, NTFS, etc.) are unusable on flash directly, because all of them have areas which are rarely rewritten (FAT, super block,), and this area would soon be corrupted.

One of the most common solutions to balance the burden of the erase blocks is FTL (Flash Translation Layer) [6], which hides the physical erase blocks behind a layer. This layer uses a map to store data about what the corresponding physical erase block is for each logical number. Initially this map is identical, so for example logical block 5 is mapped to physical block 5. This layer also contains an erase counter for each block (how many times it was erased). If this counter reaches a high number (relative to the average), the system will exchange two erase blocks (using the map), selecting an erase block which has a relatively low strain. This method is used in most pen drives to keep the burden low. It works quite well in practice, but does not provide the optimal solution to performance problems. For instance, to overwrite just a few bytes (such as a pointer in a search tree), an entire erase block (∼128K) has to been erased and reprogrammed.

Accordingly, especially in the case of root file systems, it is worthwhile using flash file systems which are designed specifically for flash devices. Now we will discuss Linux flash files systems (JFFS and JFFS2), which are freely available to everyone.

3 JFFS, JFFS2: Flash File System without Flash Index

The basic idea behind JFFS [2] is quite simple: the file system is just a concentric journal. In essence, all of the modifications on the file system are stored as a

journal entry (node). When mounting, the system scans this journal and then replays the events in the memory, creating an index to register which file is where. If the journal entries are such that the device is nearly full, the system performs the following steps:

- From the beginning of the used area copy all of the journal entries which are still valid, so the corresponding file is not deleted or overwritten.
- Erase the emptied area (erase block), so the there will be new space to store the new journal entries.

This very simple approach eases the burden on the erase blocks, but it also has its drawbacks:

1. If a dirty (deleted or overwritten) data area is in the middle of the current journal area, to free it, it is necessary to copy half of the entire journal.
2. When mounting, it is necessary to scan the entire medium.

Problem 1 is solved by the new version of this file system called JFFS2. It utilizes lists to register the dirty ratio of erase blocks, and if free space is required, it frees the dirty erase blocks (after moving the valid nodes to another place). There is a slight chance that it will free clean erase blocks as well, just to ease the burden, but this will rarely occur.

Problem 2 is only partially solved by JFFS2. It collects information via erase blocks needed when mounting, and it stores them at the end of the erase blocks, so only this needs to be scanned. Afterwards it attempts to minimize the size of the index in the RAM. However, the memory consumption and the mounting time are still linearly proportional to the size of the flash, and in practice over 512M may be unusable, especially in the case of large files e.g. video films.

Because the root of this problem lies in the base data structures and operating method of the JFFS2, we should construct a new file system to eliminate the linear dependency. To achieve this, it is necessary to store index information on the flash so as to avoid always having to rebuild it when mounting.

4 B+ Tree

Most file systems employ search-trees to index the stored data, and the B+ tree [4] is a special search-tree with the following features:

- It stores records: $r = (k, d)$; k = key, d = data. The key is unique.
- Data is stored only in leaves, inner-nodes are only index-nodes.
- In an index-node there are x keys, and also $x + 1$ pointers, each pointing to the corresponding subtree.
- The B+ tree has one main parameter, namely its order. If the order of a B+ is d, then for each index node there is a minimum of d keys, and a maximum of $2d$ keys, so there are minimum of $d + 1$ pointers, and maximum of $2d + 1$ pointers in the node.
- From the above, if a B+ tree stores n nodes, its height must not be greater than $log_d(n) + 1$. The total cost of insertion and deletion is $O(log_d(n))$.

This data structure is used by some database systems like PostgreSQL and MySQL, and file systems like ReiserFS, XFS, JF2 and NTFS. These file systems are based on the behaviour of real hard disks; that is, each block can be overwritten an unlimited number of times. Thus if there is a node insertion, only an update of the corresponding index node is needed at that location. Unfortunately it does not work well with flash storage devices, so it was found necessary to improve the flash-optimized version of the B+ tree.

5 Wandering Tree

A modified version of the B+ tree can be found in the LogFS file system [7], which is a flash file system for Linux. It is still in the development stage, and probably will be never finished, because UBIFS offers a much better alternative. This B+ variant is called a wandering tree. The general workings of this tree can be seen in Figure 1.

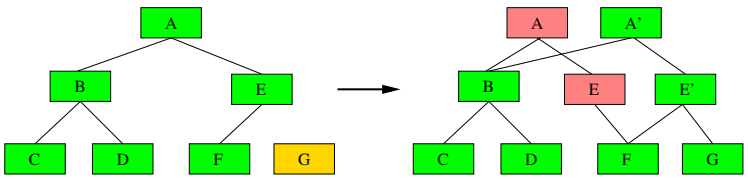

Fig. 1. Wandering tree before and after insertion

Like the ordinary B+ tree algorithm, during a node insertion it is normally necessary to modify a pointer at just one index node. In the case of flash memory the modification is costly, so this wandering algorithm writes out a new node instead of modifying the old one. If there is a new node, it is necessary to modify its parent as well, up to the root of the tree. It means that one node insertion (not counting the miscellaneous balancing) requires h new nodes, where h is the height of the tree. It also generates h dirty (obsolete) nodes, as well. Because h is $O(log_d(n))$, where n is the tree node number, the cost of this operand is still $O(log_d(n))$.

6 TNC: An Improved Wandering Tree

The above wandering tree algorithm still has performance issues, because its insert method is inefficient: it requires $log_d(n)$ new nodes, and it also generates a number of garbage nodes.

To solve these problems we decided to improve this algorithm, which now works in the UBIFS file system [1]. Due to its efficient caching technique it is able to collect node insertions and deletions in the memory, so fewer flash operations are required. At the same time, the method can ensure that if there

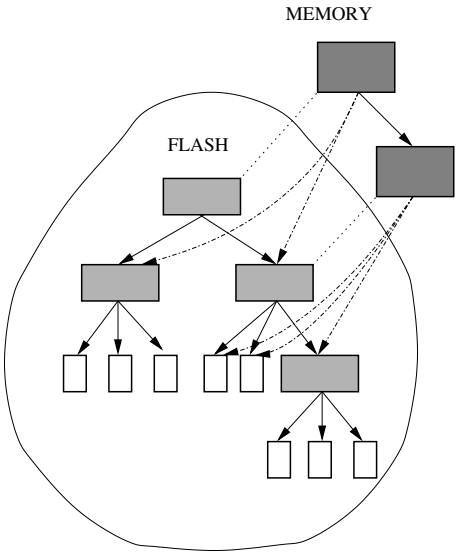

Fig. 2. The TNC data structure

is a sudden power loss (in the case of an embedded system this could happen at any time) there will be no data loss.

This new data structure and the algorithm are both called the TNC (Tree Node Cache). It is a B+ tree, which is partly in the flash memory, and partly in the memory (see Figure 2). Its operators are improved versions of those used in the wandering tree algorithm.

6.1 Data Structure of TNC

When TNC is not in use (e.g. the file system is not mounted) all the data is stored in the flash memory, in the ordinary B+ tree format.

When in use, some index nodes of the tree are loaded into the memory. The caching works in such a way that the following statement is always true : if an index node is in the RAM memory, its children may also be in the memory, or in the flash memory. But if the index node is not in the memory, all of its children are in the flash memory.

If an index node is in the memory, the following items are stored in it:

- **Flag clean:** it tells us whether it has been modified or not.
- **The address of the flash area where the node was read from.** (In the case of being eliminated from the memory, and it was not modified, this address will be registered in its parent in the memory. If it was modified, a new node will be written out, and the old location will be marked as garbage.)
- **Pointers to its children.** Each pointer stores information about whether the child is on the flash or has been read into memory.

6.2 TNC Operations

Using the data structures above, the following operators can be defined:
Search (read):

1. Read the root node of the tree into the memory, then point to it using the pointer p.
2. If p is the desired node, return with the value of p.
3. Find at node p the corresponding child (sub tree), where the desired node is.
4. If the child obtained is in the memory, set pointer p to it, and jump to point 2.
5. The child is in the flash memory, so read this into memory. Mark this child in p as a memory node.
6. Set pointer p to this child, and jump to point 2.

Clean-cache clean-up (e.g. in the case of low memory):

1. Look for an index-node in the memory which has not yet been modified, and for which all of its children are in the flash memory. If there is no such index-node, exit.
2. Set the pointers in the identified node's parent to the original flash address of the node, and free it in the memory.
3. Jump to point 1, if more memory clean-up is needed.

Insert (write):

1. Write out the data as a leaf node immediately. UBIFS writes them out to the BUD area[1], which is specially reserved for leaf nodes, just to make it easier to recover when necessary.
2. Read (search) all of the nodes into memory that need to be modified using the B+ algorithm. (In most cases it is just one index node)
3. Apply the B+ tree modifications in the memory.
4. Mark all modified nodes as dirty.

In the method described above node insertions can be collected, and we can apply them together with significantly lower flash overheads.

Commit (Dirty-cache clean-up):

1. Look for a dirty index node that has no dirty child. If found, call it node n.
2. Write out a new node n onto the flash, including its children's flash addresses.
3. Mark the place dirty where the node n was previously located, and update the flash pointer in the memory representation of the node to the new flash address.

[1] There are two kinds of data earase block in UBIFS: BUD erase block, and non-BUD erase block. UBIFS stores only the leaf nodes in the BUD erase blocks, all other type of data nodes are stored in non-BUD erase blocks.

4. Mark the parent of node n as dirty (if it is not the root node), and mark node n as clean.
5. Jump to point 1 until there is a dirty node.

Deletion:

1. Read (search) all of the nodes into memory that need to be modified using the B+ algorithm. (In most cases it is just one index node)
2. Apply the B+ tree modifications in the memory.
3. Mark all modified nodes as dirty.

7 Power Loss Handling in TNC

In the case of power-loss, the information stored in the memory is lost. To prevent this from happening, UBIFS combines TNC with a journal, where the following information is stored:

- A journal entry with a pointer to new BUD erase blocks. BUD erase blocks in UBIFS are reserved areas for leaf nodes. If the BUD area is full, a new free erase block will be reserved for this purpose.
- Delete an entry after each node deletion.
- A journal entry after each commit with a list of still active BUD areas.

In the event of power loss, the correct TNC tree can be recovered by performing the following steps:

1. Start with the tree stored on flash.
2. Look for the last commit entry in the journal. All of the events that occurred from that point have to be scanned.
3. All of the node insertions stored in the BUD areas marked in the journal, and all of the deletion nodes stored in the journal have to be replayed in the memory.

8 Experiments

Measuring file system performance objectively is not a simple task, because it depends on many factors like the architecture behaviour and caching of the operation system. To avoid theese strong dependencies, we decided to measure just the most important factor of the flash file system performance, namely the size of flash I/O operands to evaluate different TNC configurations and examine their properties using the UBIFS implementation.

The method applied was the following: we unpacked the source of Linux kernel version 2.6.31.4 onto a clean 512M file system, and deleted data using the commands below. During the test, the system counted how many flash operands (in terms of node size) were made with and without TNC.

```
mount /mnt/flash
mkdir /mnt/flash/linux1
tar xfz linux-2.6.31.4.tar.gz /mnt/flash/linux1
rm -rf /mnt/flash/linux1
umount /mnt/flash
```

We measured the performance using different TNC configuration. A TNC configuration has the following parameters:

TNC buffer size: The maximal size of the memory buffer that TNC uses to cache. If it is full, it calls commit and shrink operands. (In our test case, the maximal size of TNC was 23158 (*sizeof(node)), so if the TNC buffer was larger than 23158, shrink was not called.)

Shrink ratio: In the case of shrink, the shrink operand will be called until this percentage of the TNC nodes is freed.

Fanout: B+ tree fanout number: the maximum number of children of a tree node. ($2d$, where d is the order of the B+ tree.)

Table 1. TNC flash operations (measured in terms of node size)

Max. TNC size	Without TNC	With TNC	Shrink Ratio	With TNC / without TNC
5000	2161091	38298	25 %	1.77 %
10000	2211627	31623	25 %	1.43 %
15000	2191395	24632	25 %	1.12 %
20000	2244013	20010	25 %	0.89 %
25000	2192044	12492	25 %	0.57 %
5000	2163769	36273	50 %	1.68 %
10000	2250872	31570	50 %	1.40 %
15000	2225334	22583	50 %	1.01 %
20000	2225334	20002	50 %	0.92 %
25000	2183596	12457	50 %	0.57 %
5000	2215993	36759	75 %	1.66 %
10000	2290769	32578	75 %	1.42 %
15000	2244385	29956	75 %	1.33 %
20000	2238633	20002	75 %	0.89 %
25000	2205709	12958	75 %	0.59 %

Table 1 and Figure 3 show the results of measuring flash performance when the TNC *buffer size* and *shrink ratio* were varied. As can be seen, TNC saves 98.23-99.41% of the flash operands. Increasing the TNC size, more of the flash operations are saved, but varying the shrink ratio has no noticeable effect here.

Table 2 shows what happens if we change the fanout value of the tree. The number of TNC nodes decreases, but the size of a TNC node increases, because a TNC node contains more pointers and keys. The size of the flash operations is the product of these two factors, and it has a minimum fanout value of 32.

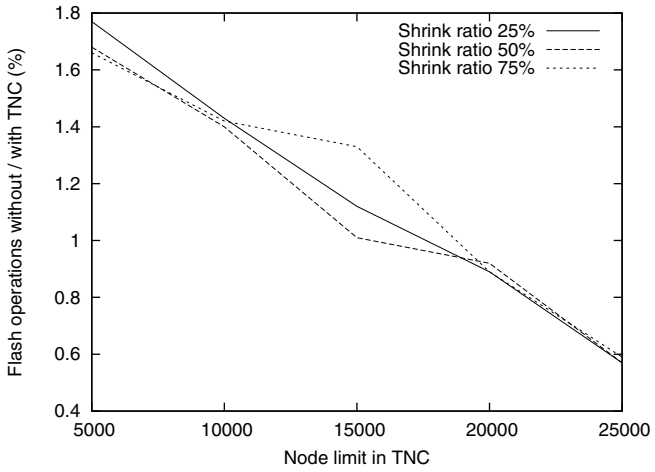

Fig. 3. Performance of TNC flash operations compared to the simple wandering algorithm

Table 2. Effect of varying the TNC fanout

Fanout	Without TNC in nodes	With TNC in nodes	Max TNC in nodes	TNC node size	Max TNC in MB	Flash ops in MB
4	1134784	48392	64801	176	10.88	8.12
8	2168308	12405	23189	304	6.72	3.6
16	1304212	3577	9662	560	5.16	1.91
32	1024363	1317	4669	1072	4.77	1.35
64	1140118	3420	3671	2096	7.34	2.35
128	767005	1245	1586	4144	6.27	3.35
256	930236	1641	980	8240	7.7	4.35

Table 3. TNC size depending on the tree fanout

I/O Size (MB) \ Fanout	8	16	32	64
50	3302	1456	703	351
100	6364	2818	1355	671
200	12925	4620	2224	1106
400	23518	8978	4282	2861
600	43320	18426	8846	5840
800	44948	22070	12273	8527

In the remaining tests we took different samples from the source code of Linux kernel version 2.6.31.4. Table 3 and Figure 4 tell us the maximal TNC size (setting no limit) when the fanout is varied, and the size of the I/O operands (size of the "file-set" above) as well.

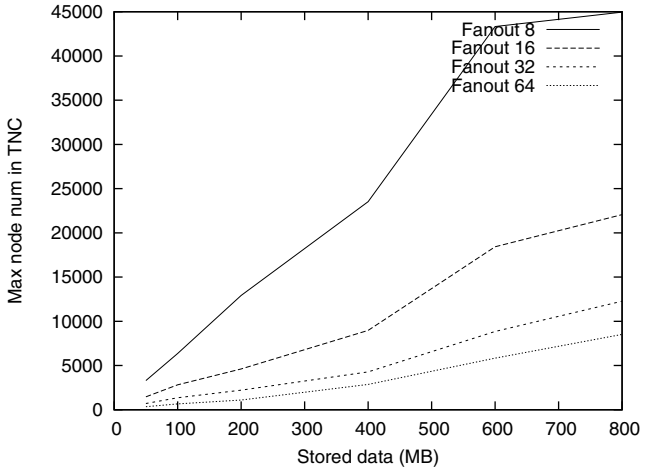

Fig. 4. TNC size depending on the tree fanout

It is very helpful to know its behaviour, especially if we want to use this technique in an embedded system where the system performance and the maximal memory usage of the file system are both of crucial importance.

9 Related Work

The authors of [9] outlined a method that has a similar goal to ours, namely to optimize the B+ tree update on a flash drive. The method collects all the changes in the memory (in LUP = lazy-update-pool), and after it has filled up, data nodes are written out in groups. It also saves flash operations, but unlike our method, using LUP means a lower read speed because, before searching in the tree, it always has to scan the LUP. In the case of TNC, there is usually a higher read speed because the nodes (at least the modified ones) are in the memory. Our method is power-loss safe, but the authors of [1] do not discuss what happens when the information is stored in the LUP. The advantage of their method is the following: the node modifications can be grouped more freely (not just sequentially), so it may be easier (and require less memory) to close the tree operations intersecting the same tree-area.

The goal outlined in [10] is also a B+ tree optimization on a flash memory. It collects as well any changes made in the memory. It calls this area the Reservation Buffer. It is filled up and these changes are written out and grouped by Commit Policy into flash as an Index Unit. It makes use of another data structure called the Node translation table to describe which node has to be transformed by which Index Unit. To search in the tree it is necessary to scan both the Node Transaction Table and the Index Units.

The method described in [5] is essentially an improved version of that described in [10]. Instead of the simple Reservation buffer it utilizes the Index

Buffer, which monitors the tree modifications and if any intersect the same node, it closes them or, where possible, deletes them. In the case of commit, it collects data about the units belonging to the same nodes, and writes them out to one page.

10 Summary

Here the author sought to improve the wandering tree algorithm used by flash file systems, so as to make it more efficient and save over 98% of the flash operands. It has a power-loss safe variant, and it has a much better performance than a simple wandering tree.

The new generation of the Linux flash file system (UBIFS) uses this algorithm and data structure, enabling one to use a flash file system efficiently on 512M or larger flash chips. This implementation is now part of the Linux kernel mainline.

Acknowledgement. This research was supported by the TÁMOP-4.2.1/ B-09/1/KONV-2010-0005 program of the Hungarian National Development Agency.

References

1. Ubifs website, `http://www.linux-mtd.infradead.org/doc/ubifs.html`
2. JFFS: The journalling flash file system. In: Proceedings of the Ottawa Linux Symposium (2001)
3. Bez, R., Camerlenghi, E., Modelli, A., Visconti, A.: Introduction to flash memory. Proceedings of the IEEE 91(4), 489–502 (2003)
4. Comer, D.: Ubiquitous b-tree. ACM Comput. Surv. 11(2), 121–137 (1979)
5. Lee, H.-S., Sangwon Park, H.J.S., Lee, D.H.: An efficient buffer management scheme for implementing a b-tree on nand flash memory. Embedded Software and Systems, 181–192 (2007)
6. Intel: Understanding the flash translation layer (ftl) specification. Tech. rep., Intel Corporation (1998)
7. Jrn Engel, R.M.: Logfs - finally a scalable flash file system
8. M-Systems: Two technologies compared: Nor vs. nand - white paper. Tech. rep., M-Systems Corporation (2003)
9. On, S.T., Hu, H., Li, Y., Xu, J.: Lazy-update b+-tree for flash devices. In: Mobile Data Management. In: IEEE International Conference on Mobile Data Management, pp. 323–328 (2009)
10. Wu, C.-H., Kuo, T.-W., Chang, L.P.: An efficient b-tree layer implementation for flash-memory storage systems. ACM Trans. Embed. Comput. Syst. 6(3), 19 (2007)