IEICE TRANS. INF. & SYST., VOL.E94–D, NO.12 DECEMBER 2011

LETTER

On Improving the Reliability and Performance of the YAFFS Flash File System**

Seungjae BAEK†, *Nonmember*, **Heekwon PARK**†a), *Member*, *and* **Jongmoo CHOI**†*, *Nonmember*

SUMMARY In this paper, we propose three techniques to improve the performance of YAFFS (Yet Another Flash File System), while enhancing the reliability of the system. Specifically, we first propose to manage metadata and user data separately on segregated blocks. This modification not only leads to the reduction of the mount time but also reduces the garbage collection time. Second, we tailor the wear-leveling to the segregated metadata and user data blocks. That is, worn out blocks between the segregated blocks are swapped, which leads to more evenly worn out blocks increasing the lifetime of the system. Finally, we devise an analytic model to predict the expected garbage collection time. By accurately predicting the garbage collection time, the system can perform garbage collection at more opportune times when the user's perceived performance may not be negatively affected. Performance evaluation results based on real implementations show that our modifications enhance performance and reliability without incurring additional overheads. Specifically, the YAFFS with our proposed techniques outperforms the original YAFFS by six times in terms of mount speed and five times in terms of benchmark performance, while reducing the average erase count of blocks by 14%.

key words: *flash memory, file system, mount speed, performance evaluation, garbage collection*

1. Introduction

Flash memory has advantages over conventional disks in terms of weight, shock resistance, and power consumption. Hence, a variety of systems make use of Flash memory as a storage medium. However, flash memory has notable limitations, including an overwrite limitation and the limited number of erasures possible to each block. In general, there are two approaches to overcome the limitations. One is introducing a new software layer, called FTL (Flash Translation Layer), between the traditional file system and flash memory. The other approach is designing a specialized flash file system such as YAFFS [4] and JFFS [5].

In this paper, we mainly focus on YAFFS, though our techniques can be applied to other flash file systems and FTLs. YAFFS is one of the widely used flash file systems, deployed in the Google Android platform and phones developed from Motorola, Philips, Samsung, LG, HTC and many others. To overcome the overwrite limitation of flash memory, YAFFS makes use of the out-of place update.

To reduce the mount time even under a power failure, we propose a technique that manages user data and metadata separately on different blocks. The separation leads to reduced mount time and enhanced performance. One concern about the separation is that there is a potential to wear out blocks used for metadata more frequently than others. To handle the problem, we design a simple but effective wear-leveling technique for enhancing the reliability of the storage system. The final contribution is devising an analytic model to anticipate expected garbage collection time. By using the model, we can set limits of the garbage collection time, thus making flash memory more predictable.

Experimental results on a real embedded system show that our modified YAFFS can enhance the mount time up to six times in a normal case and four times in a power failure case respectively, compared with the original YAFFS. Also, our proposal can reduce garbage collection overheads and decrease differences of the erase numbers among blocks.

The rest of paper is organized as follows. In Sect. 2, we describe the structure of the YAFFS. Then, we discuss how to improve performance and reliability of YAFFS in Sect. 3. Performance evaluation results are presented in Sect. 4. Finally, we provide a summary and directions for future works in Sect. 5.

2. Structure of the YAFFS

As shown in Fig. 1, when a file creating request is issued,

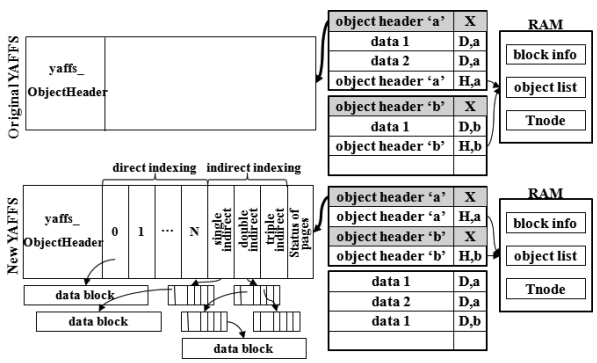


Fig. 1 New structure of YAFFS.

Manuscript received March 15, 2011.
Manuscript revised August 25, 2011.

†The authors are with the Dankook University, Korea.
*Corresponding author.

**This research was supported in part by National IT Industry Promotion Agency (NIPA) under the program of Software Engineering Technologies Development and Expert Education, by the Korea Research Foundation Grant funded by the Korea government (MOEHRD, Basic Research Promotion Fund) (KRF-2008-314-D00340) and by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MEST) (No.2009-0085883).

a) E-mail: parkhk81@dankook.ac.kr

DOI: 10.1587/transinf.E94.D.2528

YAFFS allocates a page for metadata such as file name and size, and several other pages for user data. After writing user data, it writes metadata again for update the field such as file size and it invalidate old metadata for consistency.

The metadata is called as *yaffs_ObjectHeader* in YAFFS. Also, it stores some housekeeping information, that is called as *tag* in YAFFS, on the spare area of each page, which includes object id, chuck id, and sequence number. The object id of a page is used for identifying which file owns the page, while the chuck id is used to restore ordering of pages within a file. Finally, the sequence number is used for finding out the latest page when there are more than or equal to two pages that have the same object id and chuck id to differentiate the valid page from other invalid ones.

Also, YAFFS maintains several data structures in SDRAM such as *yaffs_Object* and *yaffs_Tnode*. Each *yaffs_Object* can be constructed by reading the corresponding *yaffs_ObjectHeader* from flash memory and all *yaffs_Objects* form a hierarchy, providing directory functionality. Each file has its own *yaffs_Tnode* that supports page indexing facility It can be constructed by reading housekeeping information stored in the spare areas of pages. During the mount operation, YAFFS builds up all SDRAM data structures, which requires scanning all pages in flash memory leading to a lengthy mount time.

The new version of YAFFS, usually called YAFFS2, employs a checkpoint mechanism that saves the SDRAM data structures on flash memory at the umount time. Then, by restoring the saved structures, the mount operation can bypass scan of all pages, allowing to reduce mount overhead. However, when a power failure occurs or the checkpoint is not performed properly, YAFFS still has to pay non-trivial mount overhead. The problem becomes worse as flash memory size in smartphones increases.

3. Design of Performance-Enhanced and Reliable Flash File System

3.1 Separation of User Data and Metadata

Figure 1 presents internal structure of the original YAFFS and the new YAFFS designed in this paper. New YAFFS manages metadata and user data separately on different blocks. There are two motivations on the separation. The first is that mount operation needs to access all metadata in flash memory. If metadata resides on some specific blocks, the mount process needs to access only those blocks instead of scanning all blocks. The second is that update rate of metadata and that of user data are quite different. The *yaffs_ObjectHeader* contains various file attributes including file size and access time, which are needed to be updated whenever any part of user data is updated. In other words, metadata can be classified as hot data while user data as cold one, and the hot-cold data segregation gives a positive effect on the garbage collection performance [2].

Another challenge of our design is how to construct the *yaffs_Tnode* without scanning of all pages. To accomplish this goal, when we write a *yaffs_ObjectHeader* on a page, we piggyback index information at the same time. In current implementation, we use a large block NAND flash memory whose page size is 2 KB. But, the size of the *yaffs_ObjectHeader* structure in YAFFS is designed to be 0.5 KB due to the backward compatibility purpose. As a result, the remaining 1.5 KB is not utilized in YAFFS. We might coalesce four *yaffs_ObjectHeader*s on a page, but this is not allowed due to the write once requirement of flash memory. In this study, we exploit the remaining space to store index information, a collection of direct index pointers and indirect index pointers, like inode in ext3 file system as shown in the left side of Fig. 1, which enables to construct *yaffs_Tnode* without accessing all pages.

The final question is how to find out the blocks holding metadata at the mount time. There are two possible solutions. The one is storing block information in flash memory and the other is scanning the first page of all blocks to detect whether a block is used for metadata or user data. We implement both solutions since the former supports fast gathering of block information while the latter allows obtaining consistent block information even if a sudden power failure happens. The former solution, however, can cause serious wear-out problem so we use latter solution in this paper.

3.2 Wear-Leveling

The separation of metadata and user data on different blocks may cause skewed wear out among blocks since they have different update rates. To alleviate this problem, we design a simple but effective wear-leveling technique. It consists of two steps. At the first step, when a free block requested, it checks whether the requested block will be used for metadata or not. If so, it chooses a free block from blocks that were used for user data, and vice versa. In actuality, this leads to the swap of blocks between metadata and user data.

At the second step, it checks erase count of the candidate block. If its erase count is over the average number of erase count of all blocks, our technique checks erase count of the next free block and allocates a block that has smaller erase count among the two blocks. This step can give another chance to save the block that was more worn out than others. Note that we can use complex data structures such as a red black tree to allocate the least worn out block. But it may cause considerable overhead, especially in the embedded systems. Hence we choose our technique that can allocate a free block with O(1) time complexity using two free queue data structures, one for metadata blocks and the other for user data blocks. Also note that the technique can redistribute blocks for metadata and user data dynamically in accordance with file system workloads.

3.3 Expected Garbage Collection Time

The garbage collection operation consists of choosing a candidate block to recycle, erasing the block, copying valid pages of the chosen block, updating mapping information.

In YAFFS, there are two modes for garbage collection. One is normal mode where it tries to recycle at most one block that has less than 10 valid pages at each write request. The other is the aggressive mode, and when the total number of free blocks is lower than a predefined number (the number of blocks for *reserved* and *checkpoint* plus 2), the mode is converted from normal into aggressive. On the aggressive mode, YAFFS chooses a block that has at least invalid pages among whole flash memory and reclaims the block regardless of number of valid pages in it.

The garbage collection operation is quite a time-consuming task, which may delay user requests and exert a bad influence on responsiveness. One solution is estimating the expected garbage collection time and bound the upper limits according to user activities. In this paper, we devise a formula for the estimation as follows:

$$GC(n) = \sum_{i=1}^{n}(e_t + (r_t + w_t) * valid(i) + map(i)) \qquad (1)$$

where n is the number of blocks recycled, *valid(i)* is the number of valid pages in a block i, *map(i)* is the time required to update mapping information, and e_t, r_t, and w_t are the time for erase, read, write operations, respectively. Performance evaluation has shown that the results of Formula 1 match well with the real garbage collection times, implying that it can be exploited effectively to predict flash memory write time precisely.

4. Performance Evaluation

To carry out performance evaluation, we have built an embedded system, consisting of 400 MHz XScale ARM CPU, 64 MB SDRAM, 1 GB NAND flash memory and peripherals. To this hardware platform, we have ported Linux kernel version 2.6.28 and integrated our proposed techniques into the YAFFS. Then, we have measured mount time, garbage collection overhead, distribution of erase counts among blocks.

4.1 Performance Results

Figure 2 presents the mount time comparisons between the original YAFFS and our new YAFFS with various utilizations. Utilization was artificially increased by executing the Postmark [3] before each of the measurements. This benchmark creates a large number of randomly sized files. It then executes read, write, delete, and append transactions on these files. For fill up the utilization, the benchmark configuration is set to the average file size as 512 KB, and initially 100 files are created, and 800 transactions are processed.

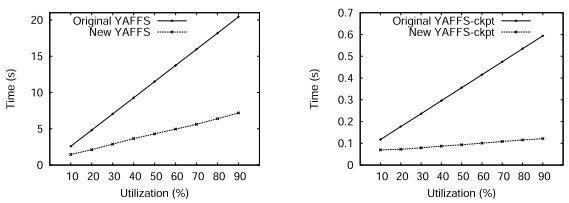
The figure shows two cases, one is the power failure case where YAFFS can not make use of the checkpoint mechanism and the other is the normal case where the checkpoint was done and information for fast boot is available during the mount operation. In the power failure case, our new YAFFS enhances the mount speed up to four times compared with the original YAFFS, since it can avoid the scanning of all pages. Even in the normal case, the new YAFFS improves the mount speed up to six times since we also store status of pages into flash memory during piggybacking as shown in Fig. 1 (note that the units of Y-axis are different between two graphs).

To evaluate the garbage collection overhead, we have measured the response time of the Postmark benchmark [3] as depicted in Fig. 3. The response time is composed of not only the Postmark's execution time but also the garbage collection time occurred during the execution. When the utilization is lower than 90%, YAFFS runs in the normal mode and do not show any distinct differences between the original and new YAFFS. However, in the aggressive mode with the utilization higher than 90%, our new YAFFS performs five times better than the original YAFFS. Sensitivity analysis has shown that the new YAFFS can segregate invalid pages from valid pages on the different blocks and incurs less frequent garbage collections, which leads to the performance differences.

4.2 Wear-Leveling

Our third measurement is the distribution of erase counts among blocks. During the execution of Postmark as shown in Fig. 3, we also have measured the number of erase operation per each block and presented the results in Fig. 4. In the new YAFFS, the average number of erase counts is 5.54 with the standard deviation of 3.41, while it is 6.35 with the standard deviation of 4.09 in the original YAFFS.

Since the new YAFFS incurs less frequent garbage collections than the original YAFFS, it can diminish the average number of erase counts. For the more, our proposed wear-leveling technique swaps metadata blocks with user data blocks, and provides another chance to save more worn-out



(a) Power failure case (without checkpoint) (b) Normal case (with checkpoint)

Fig. 2 Mount time with various utilizations.

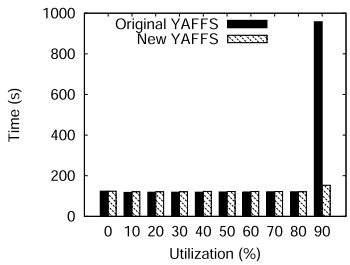


Fig. 3 Garbage collection overhead with various utilizations.

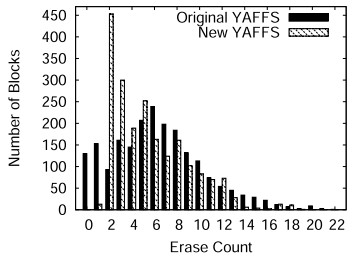


Fig. 4 Wear-level results.

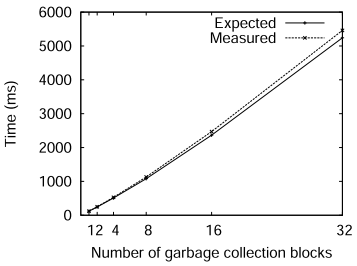


Fig. 5 The expected and measured garbage collection time.

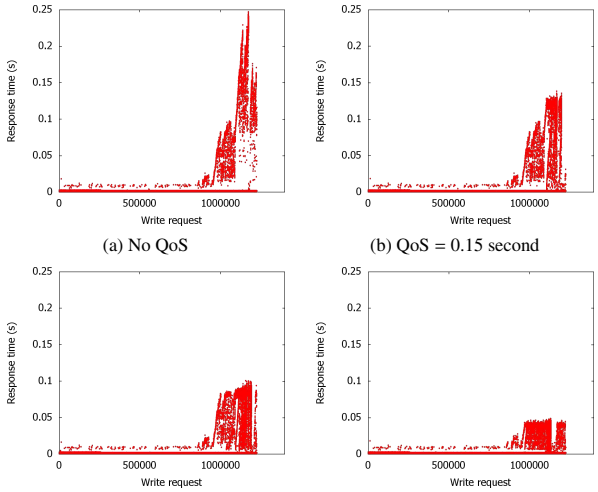


Fig. 6 QoS test results.

blocks, it can reduce the standard deviation by decreasing the difference of erase numbers among blocks. The results imply that our proposed technique can eventually expand lifetime of flash memory.

4.3 Prediction of Garbage Collection Time and Its Application

Figure 5 shows comparison between the expected garbage collection time calculated by the formula 1 and the measured time. In the calculation, based on datasheet we set the values of r_t, w_t, and e_t to 60 μs, 800 μs and 1.5 ms [1], respectively, and the time for $map(i)$ of a page to $r_t + w_t$. The results show that the estimation accords well with the measurement, though there are a little difference mainly due to the software overheads. The results imply that our proposed prediction technique of a garbage collection time can be utilized usefully for more predictable flash memory management. As a proof of concept, we have designed a quality of service mechanism that supports a guaranteed write request processing time specified by users. For example, let us assume that a user requires a write request should be satisfied less than t_qos time. Also, assume that the service time of a write request is t_ser. Note that since there is no seek time in flash memory, the estimation of t_ser is straight forward in flash memory, that is $(w_t + map) * n$ where n is the number of writing pages. Then, the maximum allowable time for garbage collection is defined as $t\_qos - t\_ser$. Finally, based on the Formula 1 and the maximum allowable time, we can figure out the upper limits of blocks to be recycled at a garbage collection trial that satisfies the quality of service required by users.

Figure 6 shows the quality of service experimental results with various t_qos as 0.15 second for (b), 0.1 second for (c) and 0.05 second for (d), respectively. For comparison purpose, we also have measured the response time of a write request in the original YAFFS as shown in figure (a), where quality of service is not specified. For these tests, we first set the utilization of flash memory as 0% and run Postmark while measuring the time consumed by each write request of a page. The results reveal that the original YAFFS violates user requirements, especially when the total number of write requests becomes 100,000 where garbage collection is triggered frequently. On the contrary, the new YAFFS indeed satisfies the quality of service requested by users.

5. Conclusion

In this paper, we have designed a metadata/user data separating management technique, a wear-leveling technique, and a garbage collection time prediction technique. Experimental results conducted in YAFFS have shown that these techniques can enhance the mount speed and garbage collection time and increase reliability by diminishing differences of erase counts among blocks. Though we focus on YAFFS in this study, we expect that our proposed techniques can be used effectively in other flash file systems and FTLs.

We are considering following two research directions for future work. One direction is designing more predictable flash memory file systems. In this work, we only show some preliminary results as a proof of concept. We plan to extend our work to support quality of service not only in a single write request level but also in a user-perceived response time level. Another direction is designing a more strict wear-leveling technique since recently developed large-capacity flash memory, such as TLC (Triple-Level Cell) or QLC (Quadruple-Level Cell) flash memory, has less than 1,000 erase cycles.

References

[1] Samsung electronics. NAND flash data sheet. “http://www.samsung.com/products/semiconductor/nand-flash”.

[2] S. Baek, S. Ahn, J. Choi, D. Lee, and S.H. Noh, "Uniformity improving page allocation for flash memory file systems," EMSOFT '07: Proceedings of the 7th ACM & IEEE International Conference on Embedded Software, pp.154–163, New York, NY, USA, 2007.
[3] J. Katcher, "Postmark: A new file system benchmark," Technical Report TR3022, Network Appliance, 1997.
[4] Aleph One, YAFFS: Yet another flash file system. "http://www.yaffs.net".
[5] D. Woodhouse, "JFFS: The journaling flash file system," Ottawa Linux Symposium, 2001.