

PFFS: A Scalable Flash Memory File System for the Hybrid Architecture of Phase-change RAM and NAND Flash *

Youngwoo Park, Seung-Ho Lim[†], Chul Lee, Kyu Ho Park
Computer Engineering Research Laboratory, EECS
Korea Advanced Institute of Science and Technology(KAIST), Daejeon, Korea
{ywpark, shlim, chullee}@core.kaist.ac.kr, kpark@ee.kaist.ac.kr

ABSTRACT

In this paper, we present the scalable and efficient flash file system using the combination of NAND and Phase-change RAM (PRAM). Until now, several flash file systems have been developed considering the physical characteristics of NAND flash. However, previous flash file systems still have a high performance overhead and a scalability problem of the mounting time and the memory usage because, in most case, the metadata is written with several words at a single update even though the writes in NAND flash must be performed in terms of page, which is typically 2 KiB. The proposed flash file system called PFFS uses PRAM to mitigate the limitation of NAND flash. The PRAM is a next generation non-volatile memory and good for dealing with word level read/write of a small size of data. PFFS hence separates the metadata from the regular data in a file system and saves them into PRAM. Consequently, the PFFS manages all the files and directories in the PRAM and outperforms other flash file systems. The experimental results show that the performance of PFFS is 25% better than YAFFS2 for small-file writes while matching YAFFS2 performance for large writes and the mounting time and the memory usage of PFFS are $O(1)$.

Categories and Subject Descriptors

C.3 [Special-Purpose And Application-Based Systems]: [Real-time and embedded systems]; D.4.3 [Operating Systems]: File systems Management—*Flash file system design*

Keywords

Flash memory, PRAM, File system, Scalability

1. INTRODUCTION

*This research was supported by the Samsung Electronics Co., LTD.

[†]He currently work in Samsung Electronics Co., LTD.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC'08 March 16-20, 2008, Fortaleza, Cear , Brazil

Copyright 2008 ACM 978-1-59593-753-7/08/0003 ...\$5.00.

Various non-volatile memories such as PRAM (Phase-change RAM), FRAM (Ferroelectric RAM) and MRAM (Magnetic RAM) have been developed as a next generation memory technologies. These non-volatile memory technologies attempt to gain acceptance as an alternative of embedded storage by their notable features such as non-volatility, random access capability, in-place update, fast write speed, low power usage, and long write endurance. Among them, PRAM is especially remarkable for its fairly high-density and current progress of commercialization. Already, Samsung announced 512Mbit working prototype of PRAM while the production of MRAM and FRAM are only 4Mbit[13]. This prototype has thirty times faster write speed and ten times longer endurance than conventional flash memory and it is expected that PRAM can replace high-density NOR flash memories within the next decade.

In comparison with the PRAM, NAND flash memory is widely used as data storage in embedded system and has benefits of a large capacity and relatively high performance for a large size of read/write requests. However, a read/write unit of NAND flash is typically the 2KiB of a page with 64byte spare region. In addition, the write operation of NAND flash should be preceded by an erase operation. The erase operation performs in terms of block. The block is much larger than a write unit so the erase time is much longer than the write time. As a result, flash memory systems adopt the out-place update scheme. When the data is modified, the new data will be written to available live pages and the old version of data is then invalidated and considered as dead pages. Later, when there are not enough live pages in the flash memory, the garbage collection reclaims the space occupied by dead pages.

In order to address current limits of NAND flash, several flash file systems, e.g. JFFS2[6], YAFFS2[3], CFFS[14], were designed. JFFS2 is a log structured file system that sequentially stores the nodes containing data and metadata in the flash memory but it does not consider the characteristics of NAND flash such as the spare regions and the read/write units. On the other hand, in the design of YAFFS2, the physical characteristics of NAND flash is fully considered so that it is more efficient than JFFS2. However, JFFS2 and YAFFS2 has serious scalability problems. Because the location of a updated page is saved in NAND flash pages or spare regions, the file system should scans the entire flash space at a mounting time and the entire directory and file system structures must be saved in main memory at that time. Besides, the mounting time and the memory usage is increased linearly according to the NAND flash size and the number

of files in file system. The CFFS stores the entire data index pointers in each inode and uses pseudo-hot-cold separation, which allocates the different flash blocks for metadata and data. By separating the metadata, CFFS can reduce the region to be scanned when the file system is mounted and improve the garbage collection performance. However, CFFS brings additional page writes to modify data index pointers in an inode when every file is updated. Furthermore, because CFFS contains directory and file structures in memory, it still has the overhead to construct and manage them.

In this paper, an efficient and scalable flash file system, which is named PFFS using PRAM and NAND flash is presented. Basically, in PFFS, the metadata and data are separated and only the metadata is maintained in PRAM. Also, the metadata of PFFS includes directory structures as well as data index pointers. Thus, PFFS reduces the cost of metadata updates and totally resolves the scalability problem of previous flash memory file systems. In addition, we propose the wear-leveling technique of PRAM in order to prevent PRAM from being worn out earlier than NAND flash. The proposed technique divides PRAM into segments and switches the data stored in segments to control the write counts equally among segments.

2. FLASH FILE SYSTEM (PFFS) DESIGN

2.1 Metadata Separation

Figure 1 shows the structure of PFFS compared with YAFFS2. In YAFFS2, only NAND flash is used as a storage device and all metadata and data are stored in NAND flash. However the PFFS is designed on a hybrid embedded storage composed of PRAM and NAND flash. Based on PRAM, PFFS separates metadata from regular file data and stores the metadata in PRAM. This metadata separation brings two important advantages for PFFS.

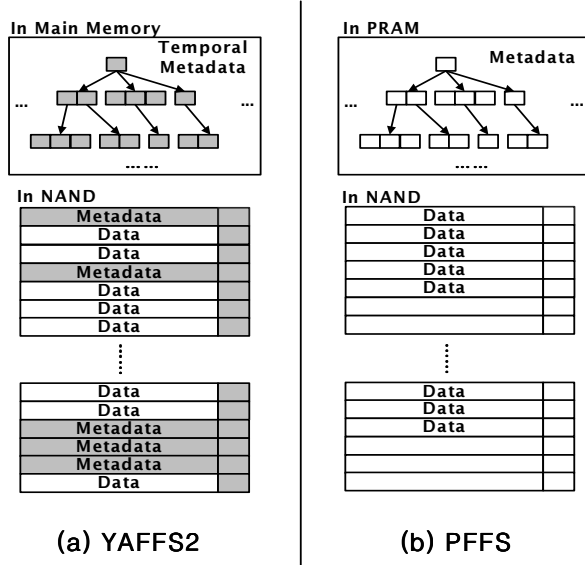


Figure 1: The comparison of YAFFS2 and PFFS structure

First, the metadata updates overhead is reduced. As

shown in Figure 1(a), YAFFS2 writes a 512 or 2KiB of a NAND flash page for each update of metadata. However, it is well-known that the metadata is frequently updated and the most updates of metadata perform in about several bytes[9]. In YAFFS2, metadata updates generate many invalidated pages and reduce the performance of file system. On the other hand, PFFS writes metadata in terms of 2bytes word because PFFS maintains metadata in PRAM such as Fig. 1(b). Consequently, the metadata separation of PFFS reduces the number of page writes and improves the performance of metadata updates.

Second, PFFS provides a fixed mounting time and memory usage. Other flash file systems need to scan a lot of NAND flash pages at mount time and keep the data index and directory structure in main memory after the file system is mounted. For example, YAFFS2 scans all the spare regions and metadata pages and makes a data index tree in main memory such as Figure 1(a). However, in the design of PFFS, each metadata contains directory structures as well as file attributes and data index pointers in PRAM. Because any data in a file of PFFS can be found from the metadata in PRAM, PFFS does not scan NAND flash page and spare region at all. Also, PFFS is not necessary to keep temporal data indexes and directory structures in main memory. This means that the PFFS has no scalability problem of the mounting time and memory usage.

2.2 Metadata Structure

The file system layout and the metadata structure of PFFS is shown in Figure 2. The basic file system layout and metadata structure of the PFFS are similar to Ext2. The superblock is used for basic description of the file system and the inode and data page bitmaps are sequences of bits that specify whether the corresponding inodes or data pages are free or not. The inode region contains the inodes which describe and manage each file in PFFS. However, in PFFS, the data block information region is additionally used for the management of NAND flash blocks and the file or directory name is stored in the file name region of PRAM. In next subsections, we describe metadata structure in more detail.

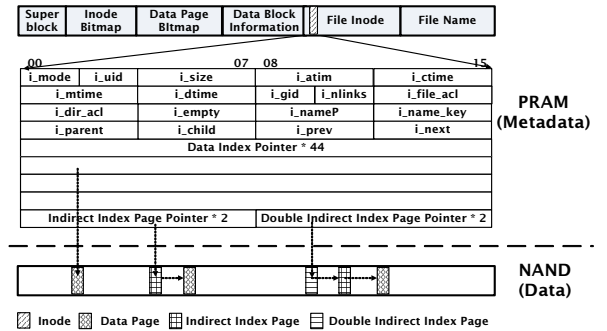


Figure 2: File system layout and metadata structure of PFFS

2.2.1 Data Indexing Structure

For data indexing in PFFS, the inode of a file contains 48 page pointers - 44 data page pointers, 2 indirect index page pointers, 2 double indirect index page pointers. The 44 data

page pointers directly point to the pages corresponding to the first 44 pages of a file while other 4 index page pointers point to the indirect or double indirect index pages such as Figure 2. The indirect index page includes data page pointers and the double indirect index page stores indirect index page pointers and 128 or 512 page pointers are contained in these index pages according to the page size of NAND flash. Therefore, if we assume that the page size of NAND flash is 2KiB, PFFS addresses only 88KiB of data using the data page pointer in PRAM and remaining data can be accessed by index pages in NAND flash. Also, in previous assumption, the 512MiB of data can be address through so maximum file size in PFFS is larger than 1GiB.

The reason why PFFS uses NAND flash for data indexing is related with the size and write speed of PRAM. First, if we assume that file systems can require up to a 1-2% overhead for metadata[7], PFFS needs 40MiB of PRAM for 4GiB NAND flash memory. It is not cost-effective to keep the entire metadata in PRAM even though PRAM is larger than other non-volatile memories. Of all kinds of the metadata, the space-overhead of data indexes is high. For the previous example, because PFFS uses at least 8MiB of PRAM to index all the pages of 4GiB NAND flash, the overhead for data indexing in the metadata is much more than 20%. Therefore PFFS can reduce much PRAM space for data indexing as allocating indirect and double indirect index pages into NAND flash. In PFFS, each inode uses only 192bytes of data indexing and PFFS can store about 60000 files of maximum 1GiB with only 32MiB PRAM.

Second, the write speed of PRAM is not faster than NAND flash if we write more than several hundred bytes of data. For that reason, when a lot of data indexes are updated at once in PFFS, that is, creating or copying large files, it is better to store the data indexes into NAND flash. In addition, recent studies[5, 4, 12] confirm that the most files are small and most write accesses are for small files; however, most storage capacity is consumed by large files that are usually only accessed for reading. Consequently, it is ineffective that PFFS allocates the data indexes, which are infrequently updated, of large files in PRAM. By index pages contained in NAND flash, PFFS rather reduces the overhead of data index updates when it writes large amount of data.

2.2.2 Directory Structure

PFFS organizes its entire directory structure in PRAM. As shown in Figure 2, the file name region includes the name of all files and directories. Also, the inode of a file and directory contains the inode pointers (*i_parent*, *i_child*, *i_prev*, *i_next*) to link the inodes of related files and directories. All the sub-directories and files in a directory are connected by the *i_prev* and *i_next* pointers as a doubly linked list. The *i_parent* and *i_child* in the inode of a directory are used to represent the relation among directories. The *i_parent* points out the inode of a parent directory and the *i_child* is used to indicate the first entry of the list which includes its sub-directories and files. Consequently, through these four inode pointers, we can find the inode of any files or directories from the root directory of PFFS and access any data in the PFFS from the data indexing structure of that inode.

However, the simple linked list structure is not effective for directory lookup because the time to search a file is increased according to the number of files in a directory. For the fast directory lookup, PFFS generates the hash key (*i_hash_key*)

of the file name and saves it in the each inode of a file. After that, PFFS sorts the directory list by the hash keys. Hence, we can quickly find the entries which have same hash key of desired file then compare the file name only for those entries and reduce the total directory lookup time.

2.3 Garbage Collection

During the garbage collection, any live pages in the reclaimed block must be copied to the other block. For efficient garbage collection, previous researches have tried to minimize the amount of copying the pages that store hot and live data because hot data, which has a higher probability for updates, has more chance to generate a dead page in the near future.

Like other flash memory file systems, PFFS needs garbage collection to reclaim dead pages. The garbage collection of PFFS is much more efficient than other flash memory file systems because PFFS stores the metadata in PRAM. From the viewpoint of file system, the metadata is usually hot data because the metadata is more frequently updated than the regular data[9, 14]. Anytime and anywhere, a write operation accompanies metadata updates. Even when there is no writing of data, the metadata should be modified for several reasons; file movement, renaming operations, file attribute modifications and so on. PFFS totally separates hot metadata from the regular data and stores the metadata in PRAM. This means that the most of copied live pages during garbage collection of the PFFS are cold data and relatively have low possibility to be updated and dead pages later. As the result, the metadata separation decreases the overhead of garbage collection and increases overall performance of the PFFS.

3. WEAR-LEVELING OF PRAM

Before we explain our wear-leveling design, it is necessary to refer to the worn-out characteristic of PRAM in the PFFS. Actually, PRAM has at least 10-times longer endurance than the conventional flash memory[13]. It seems to be acceptable to use PRAM without considering worn-out possibility. However, in PFFS, we have to notice that PRAM contains the metadata. As we previously mentioned, the metadata is more frequently updated than regular data. Because PRAM also uses the in-place update scheme, the metadata update pattern of PFFS is directly reflected to PRAM. This is a big problem because generally the metadata update pattern is highly irregular. For example, the super block which records the number of free page is updated after every NAND page write. The write counts of the superblock region in PRAM is much higher than other metadata regions. In summary, in the current PFFS design, the total life time of PRAM is seriously reduced by the frequent and ununiform updates of the metadata.

For accurate wear-leveling of PRAM, it is the best that we control the write count of PRAM in word level. However, PFFS manages the PRAM as 4KiB segment unit because it causes a lot of management overhead. Figure 3 shows the wear-leveling desing of PFFS. First of all, PFFS virtually divides the metadata in Figure 2 into 4KiB segments and the virtual segment is allocated to any physical segment in PRAM. Therefore, the physical location of metadata can be changed in PRAM. In order to manage the segments, PFFS allocates a segmentation table, usage count and bitmap in the fixed regions of PRAM. The segmentation table has the

mapping information where the virtual segment is physically saved. The write usage count region holds the total write count of each segment and PFFS can find the free segment which is not currently used by the segmentation bitmap.

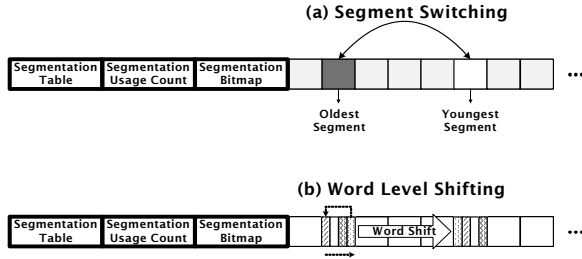


Figure 3: PRAM Segmentation and Wear-leveling Design

Based on this segmentation, the key features of our wear-leveling algorithms is the segmentation switching and word level shifting. The wear-leveling of PRAM is similar with hot-cold swapping[11, 10]. If the difference between the write count of the oldest segment and youngest segments is larger than a threshold, the data in oldest segment and in the youngest segment is swapped as shown in Figure 3. Segment changing prevents the segments from being unevenly worn but the worn-out level of each word within segments cannot be controlled. For example, if the segment holds the inodes of the files, the words, which store the size of a file (i.size), are more frequently updated in the segment. To moderate this problem, the word-level shifting is proposed as shown in Figure 3. Whenever PFFS swaps two segments, PFFS store the segments after shifting several words of the data from start of the segments. The word-level shifting prevents that the specific usage pattern in the virtual segment is repeated in other physical segments.

4. EXPERIMENTAL RESULTS

In order to test PFFS, we developed the evaluation board as shown in Figure 4. This evaluation board has a 266MHz ARM processor and 64MiB SDRAM. It also includes most of storage devices which are currently used for embedded systems; NOR, SLC and MLC NAND, OneNAND, UtRAM as well as PRAM. We were offered PRAM prototype by Samsung and implemented the PFFS in a Linux OS with kernel version 2.6.19 using 32MiB PRAM prototype¹ and Samsung K9F2G08U0A NAND flash memory[2] in our evaluation board. The NAND flash characteristic is summarized in Table 1.

However, PRAM is not freely available for us to test various and heavy workloads. We just verify the functionality of PFFS using PRAM. The performance test is actually carried by emulated PRAM. We analyze the read/write time of PRAM and emulate that by power backed UtRAM[1] with delay. Our emulation method is quite accurate because the PRAM read/write time is deterministic and the data in power backed UtRAM is not volatilized. We compared the performance and scalability of PFFS with YAFFS2 and show the wear-leveling results of PRAM.

¹We cannot reveal the detailed specification of PRAM because it is confidential for the Samsung electronics.

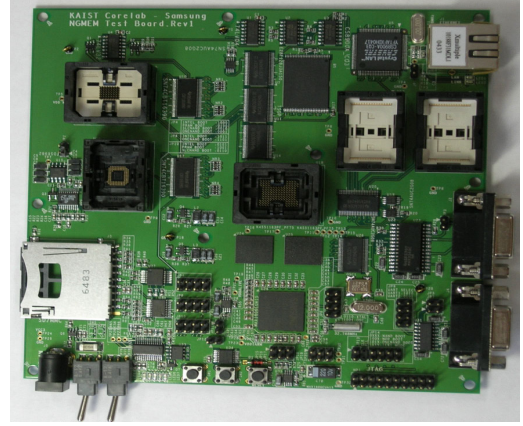


Figure 4: The evaluation board to test the PFFS

Table 1: NAND flash memory characteristics

Part number	K9F2G08U0A
Block size	(128K+4K) Bytes
Page size	(2K + 64) Bytes
Page read time	25 μ s
Page write time	200 μ s
Block erase time	1.5ms

4.1 Read/Write Performance

At first, we compared the the small-file performance of PFFS with YAFFS2. We used the PostMark[8] benchmark program, which is well-known for measuring file system performance over a workload composed of many short-lived, relatively small files. We configure the PostMark to create 500 files with a file size range of 512 bytes to 16KiB. We increase the number of transactions from 15000 to 30000 for every run and collect the results. For all experiments, the ratio of file creation/deletion is 5:5 and the ratio of read/write operation is 1:9.

Figure 5 shows the results of performance comparison. The bar graph represents the total execution time of PostMark and the line graph shows the number of NAND page writes during experiments. This results shows that the performance of PFFS is better. It is mainly because the number of NAND page writes is reduced using PRAM as metadata region.

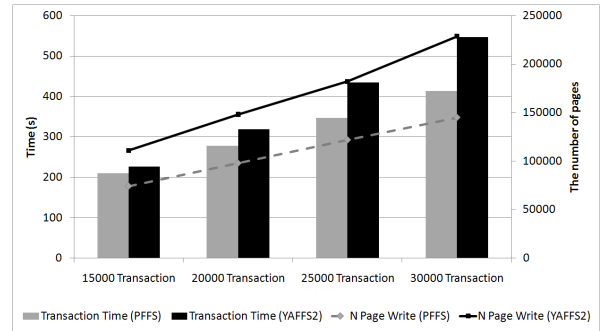


Figure 5: Small-file performance under PostMark benchmark

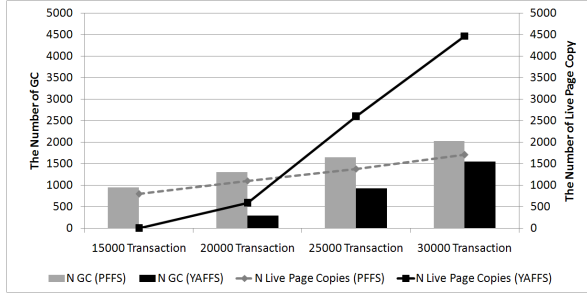


Figure 6: Comparison of garbage collection under PostMark benchmark

Especially the performance gap between PFFS and YAFFS2 is increased as the number of transaction is performed. This is the effect of garbage collection. Figure 6 shows the results of garbage collection when we execute above performance test. The bar graph shows the number of GC (Garbage Collection) and the line graph shows the number of total live page copies during GC. From the Figure 5 and 6, we note that the difference of the number of GC affects to the difference of performance between PFFS and YAFFS2. However, after 30000 transactions, the number of GC in the PFFS is larger than YAFFS2 but the number of live page copies of YAFFS2 is much more than PFFS. This means that the performance of GC in the PFFS is better than YAFFS2. On average, the extra writes of the PFFS is only 0.84 times per GC although YAFFS2 is more than 2.7 times per GC. As we mentioned in section 2.4, metadata separation also increases the garbage collection performance and this will affect overall performance of the PFFS.

The weak point of the PFFS is the performance of large file writes. Table 2 shows the execution time and the number of NAND page write when we wrote 100MiB file with block size 2KiB. The time to write of the PFFS is about six percent longer than YAFFS2. It is mainly because there is only one metadata updated for file creation so the effect of metadata separation and PRAM is not shown. Rather, the PFFS needs to update the page bitmap in metadata continuously and many writes in PRAM cause the overhead. However this overhead will be reduced more because the write time of current PRAM prototype is continuously improved.

Table 2: Large-file write performance

	PFFS	YAFFS2
Write time (s)	90.2	85.3
# of Flash page writes	51301	51202

4.2 Scalability Test

In this section, we compare the mounting time and memory usage of PFFS and YAFFS2. We measure mounting time and memory usage of each flash file system after we writes a large file or many small files. Table 3 and 4 show the results of the scalability test.

For the mounting time in Table 3, we can easily know that the mounting time of YAFFS2 is linearly increased according to the growth of used pages in NAND and the number of files in YAFFS2. During file system mounting, the mounting time is proportional to the number of data pages and

Table 3: Comparison of mounting time

Test environment	PFFS	YAFFS2
No file write	0.159 (s)	0.281 (s)
After a 50MiB file write	0.159 (s)	2.054 (s)
After a 100MiB file write	0.159 (s)	3.817 (s)
After a 200MiB file write	0.159 (s)	7.201 (s)
After 12800 2KiB files write	0.159 (s)	3.655 (s)
After 25600 2KiB files write	0.159 (s)	8.406 (s)

Table 4: Comparison of memory usage

Test environment	PFFS	YAFFS2
No file write	64KiB	48KiB
After a 50MiB file write	64KiB	84KiB
After a 100MiB file write	64KiB	124KiB
After a 200MiB file write	64KiB	300KiB
After 12800 2KiB files write	64KiB	2572KiB
After 25600 2KiB files write	64KiB	5228KiB

files because YAFFS2 must scan the metadata pages and spare regions of NAND flash. On the other hand, the PFFS scans only fixed regions in PRAM regardless of any other conditions.

Similar to the mounting time, the memory usage of YAFFS2 is not scalable as shown in Table 4. This proves that YAFFS2 contains dynamically many data in memory. Especially if there are many files in YAFFS2, it needs much memory to save the directory structures. However, the memory usage of the PFFS is fixed because the PFFS store whole directories and flash memory management structures in PRAM. As a results, the PFFS is completely scalable in all aspect and we can apply the PFFS to any resource constrained embedded system.

4.3 Wear-leveling of PRAM

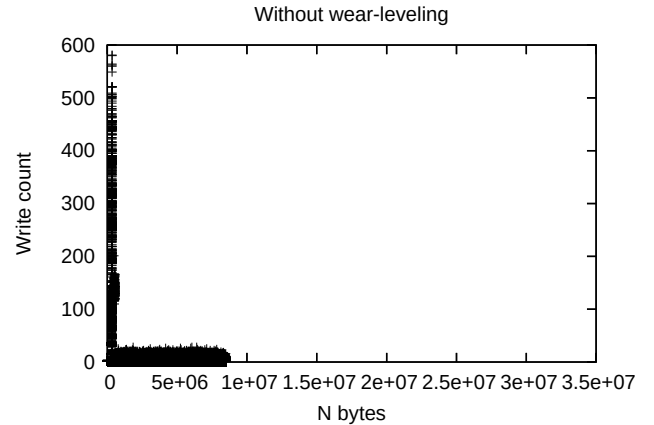


Figure 7: Write access pattern of PRAM without wear-leveling

Figure 7 shows the write counts of PRAM in the PFFS when we execute the PostMark benchmark with the configuration of executing 40000 transactions without wear-

leveling. The x-axis shows the byte address in PRAM and the y-axis represents the write counts of each bytes. From this Figure, we can see that the write is concentrated in the specific region of PRAM and some region has about 600 write counts although the write counts of most other region are under 100 times and what is worse is that there are a lot of unused region in PRAM. This proves that the PFFS has the worn-out problem of PRAM.

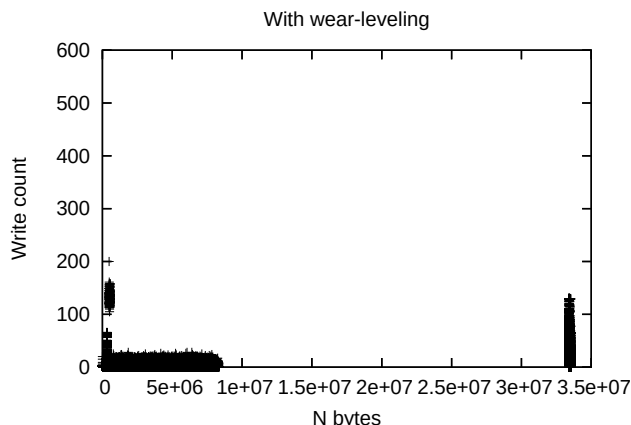


Figure 8: Write access pattern of PRAM with wear-leveling

Figure 8 shows the results after we applied the wear-leveling design to solve this wear-leveling problem. If we apply our segmentation design, the maximum write counts are reduced under 200 for all bytes in PRAM. In special, the write that is concentrated on the beginning parts of PRAM in previous test is moved to the unused segments by segmentation changing. If more write are occurred in PRAM, this unused segments is continuously used instead of increasing the write counts of previously used segments. Therefore, using our segmentation technique, all bytes in PRAM will not wear out before flash memory is worn out.

5. CONCLUSION

In this paper, we presented a scalable flash memory file system, called PFFS, for the hybrid architecture of PRAM and NAND flash memory. PRAM is one of the next generation non-volatile memories and has many good features comparing with flash memory. We use PRAM as the metadata storage for NAND flash file system. By separating metadata from NAND flash memory, we totally solve the scalability problem of previous NAND flash file system. The experimental results show that the mounting time and the memory usage of PFFS are $O(1)$. This also improves the file write and garbage collection performance. The performance of PFFS is 25% better than YAFFS2 for small-file writes while matching YAFFS2 performance for large writes. In addition, we propose the segmentation of PRAM and solve the wear-leveling problem of PRAM in the PFFS. Currently

it is the limits of the PFFS that we need the PRAM as metadata storage. However, PRAM is expected to replace the NOR flash memory for code execution and PRAM will be used with NAND flash memory in the future embedded systems. Then, our PFFS will be more easily applied to various embedded systems.

6. ADDITIONAL AUTHORS

Additional authors: Jin Kyu Kim(Samsung Electronics Co., email: goodjk.kim@samsung.com), Kyoungil Bahng(Samsung Electronics Co., email: kybang@samsung.com)

7. REFERENCES

- [1] K1S5616BCM Data Sheet. <http://www.samsung.com>.
- [2] K9F2G08U0A Data Sheet. <http://www.samsung.com>.
- [3] YAFFS A Flash file system for embedded use. <http://www.yaffs.net/>.
- [4] An-I A. Wang et al. Conquest: Better Performance through a Disk/Persistent-RAM Hybrid File System. In *Proceedings of 2002 USENIX Annual Technical Conference*, 2002.
- [5] D. Relli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *Proceedings of 2000 USENIX Annual Technical Conference*, 2000.
- [6] David Woodhouse. JFFS:The Journaling Flash File System. In *Proceedings of the Ottawa Linux Symposium(OLS)*, Red Hat, Inc., 2001.
- [7] E. L. Miller, S. A. Brandt and D. D. E. long. HeRMES: High-performance reliable MRAM-enabled storage. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, 2001.
- [8] J. Katcher. PostMark: A New File System Benchmark. Technical report, Technical Report TR3022, Network Appliance Inc., 1997.
- [9] J. Piernas, T. Cortes, J. M. Garcia. The Design of New Journaling File Systems: The DualFS Case. *IEEE Transaction on Computers*, vol. 56:pp. 267-281, 2007.
- [10] J.-W. Hsieh, L.-P. Chang and T.-W. Kuo. Efficient on-line identification of hot data for flash-memory management. In *Proceedings of the 2005 ACM symposium on Applied computing*, 2005.
- [11] L.-P Chang, T.-W Kuo and S.-Wu. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Transactions on Embedded Computing Systems*, vol. 3:pp. 837-863, 2004.
- [12] N. Agrawal, et al. A Five-Year Study of File-System Metadata. In *Proceedings of the 5th Conference on File and Storage Technologies (FAST '07)*, 2007.
- [13] Sean Shim. Samsung introduces working prototype of PRAM. <http://www.eetimes.com>.
- [14] Seung-Ho Lim and Kyu-Ho Park. An Efficient NAND Flash File System for Flash Memory Storage. *IEEE Transaction on Computers*, vol. 5(no. 7):pp. 906-912, July 2006.