A PRAM and NAND Flash Hybrid Architecture for High-Performance Embedded Storage Subsystems

Jin Kyu Kim[1] Hyung Gyu Lee[1] Shinho Choi[2] Kyoung Il Bahng[2]
[1]Samsung Advanced Institute of Technology, CTO, Samsung Electronics Co. LTD
[2]Memory division, Semiconductor Business, Samsung Electronics Co. LTD
South Korea
{goodjk.kim, hyunggyu.lee, shinho.choi, kybang}@samsung.com

ABSTRACT

NAND flash-based storage is widely used in embedded systems due to its numerous benefits: low cost, high density, small form factor and so on. However, NAND flash-based storage is still suffering from serious performance degradation for random or small size write access. This degradation mainly comes from the physical constraints of NAND flash: erase-before-program and different unit size of erase and program operations. To overcome these constraints, we propose to use PRAM (Phase-change RAM) which supports advanced features: fast byte access capability and no requirement for erase-before-program.

In this paper, we focus on developing a high-performance NAND flash-based storage system by maximally exploiting the advanced feature of PRAM, in terms of performance and wearing out. To do this, we first propose a new hybrid storage architecture which consists of PRAM and NAND flash. Second, we devise two novel software schemes for the proposed hybrid storage architecture; FSMS (File System Metadata Separation) and *h*FTL (hybrid Flash Translation Layer). Finally, we demonstrate that our hybrid architecture increases the performance up to 290% and doubles the lifespan compared to the existing NAND flash only storage systems.

Categories and Subject Descriptors

D.4.2 [Operating System]: Storage Management – *Storage Hierarchies*

General Terms

Design, Performance, Experimentation

Keywords

PRAM, NAND flash, flash translation layer (FTL), file system

1. INTRODUCTION

With a rapid growth of embedded system markets, flash-based storage systems are becoming more popular, because their non-volatility, low-power consumption, small form factor, and high reliability are suitable for embedded systems. There exist several types of flash memory devices. Among them, NAND flash [1] is the most commonly used device for bulk data storage systems, due to its high density with low cost. In this paper, we are focusing on enhancing the NAND flash-based storage systems.

Generally, flash memory devices have two main physical constraints: erase-before-program requirement and different granularity on erase and program operations, which have been serious burdens in implementation of flash-based storage systems. To overcome these constraints, flash-based storage systems require an additional layer between file system and physical storage devices called FTL (Flash Translation Layer) [14] that abstracts flash-based storage systems into over-writeable block devices, like HDDs (Hard Disk Drives).

The main role of FTL is to redirect a write request to erase-free space by keeping the mapping information between logical and physical address spaces. Since the performance of flash-based storage systems largely depends on this mapping technology, many approaches have been tried to develop efficient mapping algorithms such as page-level mapping algorithms [16,19], block-level mapping algorithms [15], and hybrid mapping algorithms [5,10,11,12] called log block mapping. Although recent mapping algorithms significantly enhance the performance of flash-based storage systems with reasonable implementation cost, it is known that they are still suffering from the performance degradation for random write requests, even in state-of-art research. This low performance characteristic for random write requests should be clearly problematic in future systems which require more complex and frequent random access requests.

Recently, PRAM (Phase-change RAM) [2,3] device, which supports fast byte- or word-access capability without erase-before-program requirement like normal DRAMs, is introduced as an next generation non-volatile memory device and it will be commercially announced by the several chip vendors very soon [17]. At a first glance, the PRAM may seem to be an ideal non-volatile storage device which can completely overcome the physical constraints of traditional flash memory devices. It surely seems that PRAM successfully replaces NOR flash since PRAM could support higher density and faster write speed than NOR flash while providing similar read performance. However, its density, cost and write performance in current technology step are not enough to replace NAND flash. DeVoss expects it will take

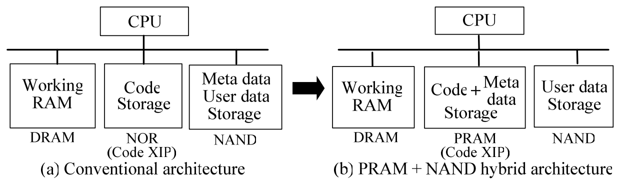


Figure 1. Embedded storage architecture

substantial time for PRAM to replace NAND flash completely [6]. So, instead of providing complete replacement scheme of NAND flash, we propose a hybrid storage architecture of PRAM and NAND flash memory, as shown in Figure 1. PRAM is used not only for code storage but also for the part of data storage. We expect the enhanced features of PRAM can successfully complement the weakness of NAND flash-based storage architecture, especially for low performance in random and small write requests. Furthermore, it would not incur additional hardware implementation cost since we share the remaining space of PRAM which is already equipped for code storage.

To support the proposed PRAM and NAND hybrid storage architecture, we first propose a separate allocation of file system metadata from the user data (FSMS: File System Metadata Separation) by storing the metadata into the PRAM. This idea enhances the efficiency of metadata management of file system, which reduces the number of random access requests to FTL substantially without any significant modification to existing storage component. Second we design a page-based mapping algorithm supported by the PRAM, which shows high performance even for random access requests with reasonable implementation cost. Unlike the FSMS, second idea makes FTL itself robust to random access pattern in block device driver level regardless of file system. Finally, we implement the above two ideas based on an existing file system and FTL, and then demonstrate the effectiveness of the proposed hybrid architecture by comparing them with conventional storage architectures. We assume that the system already has PRAM devices for the code storage replacing NOR flash memory and we use the remaining area of the PRAM device, which means our proposed ideas do not accompany with any additional hardware cost.

The rest of the paper is organized as follows. In the next section, we briefly review the related work. Section 3 describes details of the FSMS scheme implementation. Section 4 includes the design of hybrid FTL. Section 5 compares the proposed hybrid storage with existing storage architectures. Conclusions are given in the Section 6.

2. RELATED WORK

In this section, we revisit the mapping algorithms in NAND flash-based storage architecture and the related work about hybrid storage approaches with next-generation NVRAMs (Non-volatile RAMs) [2,3,4], separately.

2.1 FTLs

FTL [14] is a software layer to provide HDD-like over-writable address space between the flash memory and file system. Generally, NAND flash memory consists of blocks. Usually, a block is composed of 64 or 128 pages and the size of a page is 2KB or 4KB. NAND flash needs to perform erase operation before program operation. Furthermore, write operation of NAND flash is based on a page while erase operation is based on a block. To mimic HDD-like storage, it is important to hide these constraints of NAND flash. To attain it, FTL should do remapping between logical address space and physical NAND address space. Since mapping algorithm of FTL has large impact on the performance and life span of NAND flash-based storage, lots of efforts have been paid to developing FTL mapping algorithms. Existing FTL mapping algorithms are roughly categorized into three: page mapping [16, 19], block mapping [15] and log block mapping [5,10,11,12] according to the granularity of mapping unit.

2.1.1 Page mapping

In early stage, page mapping scheme [16,19] has been studied, because its performance is relatively high while its algorithm is simple. In page mapping, logical space to physical space mapping is based on a physical page unit of NAND flash. Therefore, write request from the file system to FTL can be redirected to any physical pages without any constraints. Due to fine grained mapping, flexible storage management is possible and it shows endurable performance degradation for random access patterns. However, it needs to manage large amount of mapping information. For example, it requires additional 1MB memory space for 512MB NAND flash storage since it must keep physical mapping information (usually 4 bytes) for each logical page. Generally, a large amount of this mapping information should be kept in the main memory space for pursuing high performance, as well as NAND storage itself. Furthermore, scanning and rebuilding mapping information at start-up may delay booting time substantially.

2.1.2 Block mapping

In contrast to the page mapping scheme, the basic mapping unit of block mapping scheme [15] is a physical block of NAND flash. To translate logical address into physical address, logical block number and logical page offset within a logical block are calculated from logical page address, and then physical block number is gained through block mapping information. Since logical pages are stored sequentially in physical pages in the same order of a logical block, address translation can be done in one step. Since block mapping FTL keeps mapping information (usually 4 bytes) only for blocks, the size of mapping table is relatively small (16KB memory space for 512MB NAND flash storage.) However, it incurs so much extra NAND flash operation when update requests occur on a part of a block. At every update, entire block which includes the updated logical pages must be copied to a free block, which results in a significant performance degradation.

2.1.3 Log block mapping

To exploit the merits of page and block mapping schemes, log block mapping schemes have been introduced [5,10,11,12]. In log block scheme, blocks are divided into data blocks and log blocks. Basically, it uses block mapping for data blocks and page mapping for log blocks. In order to overcome disadvantages of block mapping, write requests are always performed on a log block at first. When a log block runs out of free pages or user request is out of range where the log block covers, FTL does merge operations on the data block and log block to make one

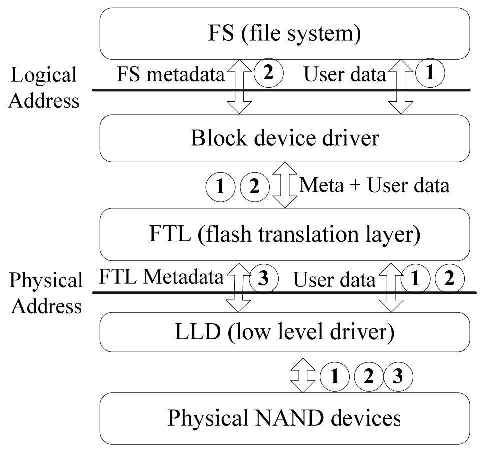


Figure 2. NAND flash-based storage architecture

data block. Although log block mapping is able to mitigate extreme overheads of the block mapping, they are still suffering from the poor performance for random access requests, since the number of log block is fixed at design time to avoid the large main memory consumption. This characteristic will surely be a burden in future storage system because embedded system applications will require more frequent random write requests as the multi-thread and multi-processing environments become more popular.

2.2 Hybrid storage system with NVRAM

Previous research [7,8] proposed hybrid storage architectures that exploit another type of NVRAMs such as MRAMs (Magnetic RAMs). They tried to use the MRAM for storing the file system metadata. In [9], A.Wang proposed hybrid storage file system architecture with battery backup RAM.

Since they directly touched the file system itself, their approaches require large implementation cost. Most of all, their approaches may have a compatibility problem with previous storage system architecture. In addition, their target architecture is only for a HDD-based storage system, which means their goals are mainly to enhance the performance by reducing the negative effects of mechanical mechanism of HDDs.

This paper, by contrast, is focusing on NAND flash-based storage. Since NAND flash-based storage has different characteristics from HDD, different approaches are required to increase the storage performance. In addition to performance, increasing the life span of storage is also important topic in our study because blocks of NAND flash device have limited program/erase cycles (usually 15K ~ 100K). For these purposes, we exploit NVRAM for file system and FTL. Our approach also reduces the implementation cost by using an existing storage software with minimal modifications. To the best of our knowledge, this is the first work to exploit the PRAM with NAND flash to implement an efficient storage system including both file system and FTL.

3. FILE SYSTEM META DATA MANAGEMENT USING PRAM

In this study, our new ideas are implemented as two parts; separate allocation of metadata from user data by storing the metadata into the PRAM called FSMS (File System Metadata Separation) and redesign of page-mapped FTL called *h*FTL, for enhancing random write performance. This section explains the problems of conventional NAND flash-based storage systems and describes the details of first idea – FSMS.

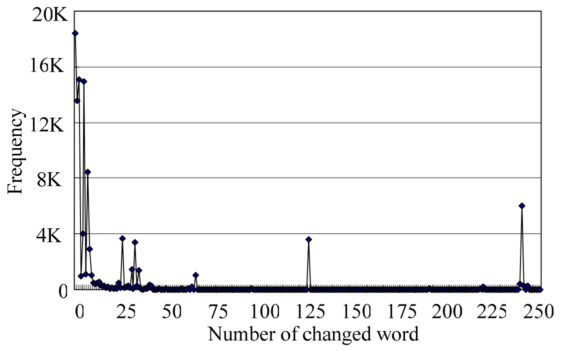


Figure 3. Distribution of changed words in metadata write request (request size = 256 word)

3.1 Problems in conventional embedded storage systems

The conventional NAND flash-based storage software consists of two main parts; file system and FTL as shown in Figure 2. Generally, file system has two types of data. One is user data and the other is metadata. At every file update, metadata (② in Figure 2) should be updated together with user data (① in Figure 2) and this metadata is directly sent to the FTL through general block device driver, in the forms of block. In general, the amount of metadata to be really updated in a metadata block is much smaller than the size of a block[1]. This means that metadata updates accompany with unnecessary storage access overhead. This inefficient metadata management is surely problematic because metadata tends to be updated frequently (around 50% of total write requests [13].)

Our analysis conducted on a FAT-based file system shows the distribution of the number of modified words (a word - 2bytes) per a metadata block write request as shown in Figure 3. About 80% of metadata write requests need to modify less than 25 words out of 256 words. On the average, the only about 16% of a metadata block is actually updated. We expect that similar results are found in other various workloads.

Even worse, the metadata updates occurring between consecutive user data updates break sequential access of user data into random access. In most file systems, metadata and user data are stored in non-contiguous address space. Random access in FTL may increase the number of merge operations and garbage collection cost, which finally results in serious performance degradation and fast worn-out of NAND flash memory.

[1] The block in this context does not mean physical block of flash memory but sector in file system – usually 512 bytes.

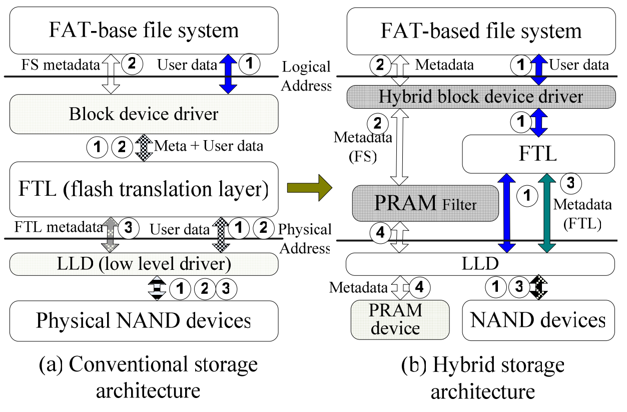


Figure 4. Software architecture of embedded storage

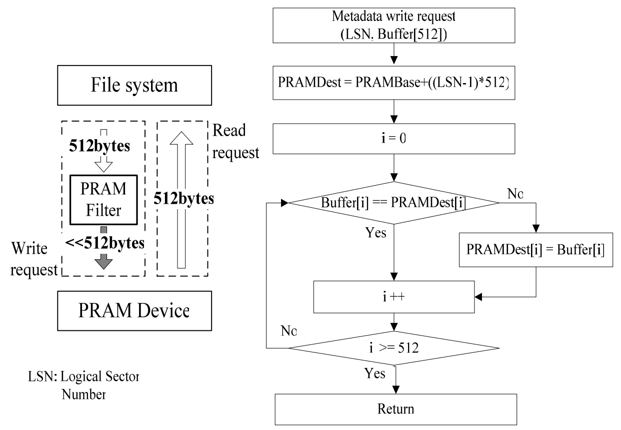


Figure 5. PRAM filter layer

3.2 FSMS: File system metadata on PRAM

We design FSMS, which stores metadata of file system on PRAM as shown in Figure 4(b). And then, we implement it on a FAT-based file system and a log block FTL which are optimized for embedded systems.

Our hybrid approach is implemented in a way to minimize the modification of existing file system and FTL so that the approach would be easily adopted in real world. Grayed boxes indicate modified and added modules to implement FSMS. First, we modify the general block device driver to distinguish the metadata update from the user data update. Since the FAT file system stores metadata into fixed position, the hybrid block device driver can easily separate metadata from the user data using file system format information. Once write request is identified as a metadata write, it is directly sent to the PRAM filter, otherwise it is sent to the FTL.

Second, we add a PRAM filter which is responsible for converting block-based write requests into byte-based write requests. Since writing entire metadata block on PRAM takes more time than writing on NAND flash, the PRAM filter is necessary for optimization. When a metadata block write request arrives, the PRAM filter pre-reads the entire destination block from PRAM and then compares them with write request data as shown in Figure 5. Then, only the words to be modified (as represented as ④ in Figure 4) are delivered to the PRAM. Since read performance of PRAM is about 125 times faster than its write performance, pre-reading and comparison do not make much overhead. For read request to PRAM, the request is forwarded to the file system without filtering. This filtering layer successfully compensate for slow write performance of PRAM for metadata writing.

4. HYBRID FLASH TRANSLATION LAYER

As embedded systems become more complex, multiple simultaneous access requests may frequently occur, which makes user data patterns into serious random access patterns. Since our FSMS scheme is applicable only to file system metadata management, a more novel scheme is required to improve the random write performance for the file system user data. Thus, we design a new FTL which guarantees high performance even for the random access pattern of user data. In this section, we describe the details of redesign of page-mapped FTL called *h*FTL, which maximally exploits the enhanced features of PRAM.

Among the existing mapping algorithms, it is known that the page mapping algorithm shows high performance and relatively stable performance even for random access pattern. However, the demand of large main memory to keep page mapping information has been a main hurdle to adopt the page mapping algorithm. In our scheme, by keeping the mapping information into the PRAM, we avoid excessive consumption of main memory space.

4.1 *h*FTL: A page-mapped *Hybrid* FTL with PRAM

Our *h*FTL is based on existing page mapping schemes, but we optimize it with three advanced features. First, we use the PRAM to store the FTL metadata such as page mapping table, physical NAND block information, bad block management information, and so on. This can significantly reduce the implementation cost including main memory footprint. Second, we support *logical-page-delete* function for reducing garbage collection cost, which is not considered in previous FTLs. Last, we add reserved block scheme to prevent frequent merge operations when the utilization of storage is high (almost close to 100%). Reserved block is a group of blocks that are not counted for user space although they are used to store user data.

Figure 6 illustrates the mapping scheme of our *h*FTL and its related data structures. The *h*FTL stores the FTL metadata such as a mapping table, physical page status bitmaps, and physical block information on the PRAM rather than NAND flash. NAND flash stores only the data from the file system. The blocks of NAND flash are divided into data blocks, buffer block, and garbage blocks as shown in Figure 7. The *h*FTL stores the newly arrived data only on the buffer block. When a buffer block runs out of free page, the *h*FTL changes the status of current buffer block into a data block and tries to allocate a new buffer block from garbage blocks. If the number of garbage blocks is below the given threshold, it performs merge operation and garbage collection by selecting a victim block among the data blocks. In current implementation, the number of buffer block is one, but it can be changed depending on the optimization policy.

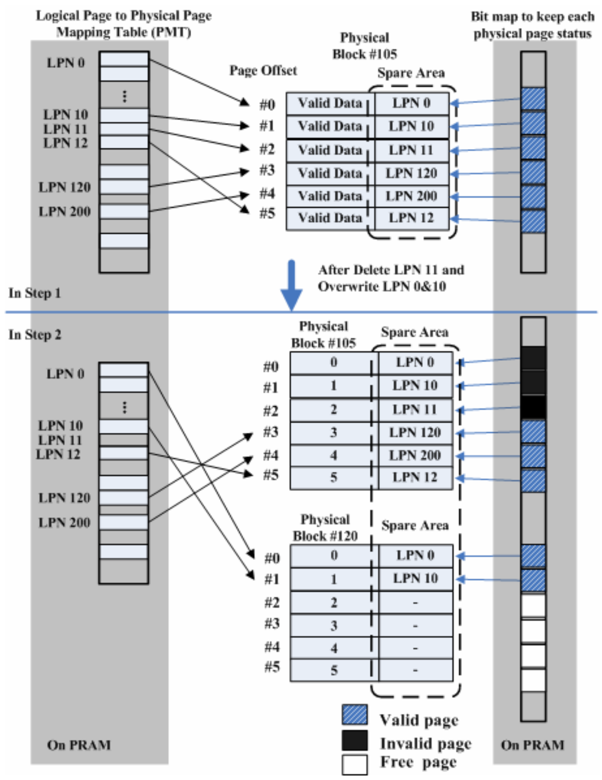

Logical Page to Physical Page Mapping Table (PMT)
Bit map to keep each physical page status
Physical Block #105
Page Offset
Spare Area
Valid Data
After Delete LPN 11 and Overwrite LPN 0&10
In Step 1
In Step 2
Physical Block #120
Valid page
Invalid page
Free page
On PRAM

Figure 6. Page-mapped FTL with PRAM

The basic philosophy of *h*FTL is to store the metadata into the PRAM but this may result in performance degradation because the write performance of PRAM is relatively low compared with a main memory. Thus, we temporarily keep the metadata related with current buffer block in the main memory and write them back to the PRAM when the buffer block is changed into a data block. Details of read and write operations are shown in Figure 8. For easy power-failure recovery, *h*FTL writes logs into the PRAM at every PRAM update. The page mapping scheme in our *h*FTL is simple but powerful. However, we found two problems which degrade the overall performance.

First problem is from garbage collection. In the garbage collection, the FTL selects a victim block and then copies valid pages in the victim block into new free block. During this process, FTL may copy deleted-pages [2] with valid pages because conventional FTLs do not support *logical-page-delete* function. This may result in increasing the cost of merge operation significantly. To prevent this, we implement *logical-page-delete* function which invalidates deleted-pages in FTL level. Whenever the file system deletes a file, it calls *logical-page-delete* function to invalidate the physical pages which belong to the deleted file. *Logical-page-delete* function updates the physical page status bitmap as well as the logical page mapping in PMT (Page Mapping Table). The example cases are showed in Figure 6. In step 1, logical pages #0, #10, #11, #12 #120 and #200 have valid data, and corresponding bitmaps are set to valid. Step 2 shows the changed bitmap status and PMT after deletion of logical page #11 and updates of logical pages #0 and #10. For the deleted logical page #11, corresponding entries of PMT and bitmaps are marked as *invalid*. For the updated logical pages #0 and #10, their corresponding entries of PMT are remapped to new physical pages #0 and #1 of block #120, and bitmaps corresponding to physical pages #0 and #1 of physical block #105 are set to *invalid*. When FTL needs to merge on block #105, it uses bitmap information to avoid unnecessary copy of invalid physical pages such as pages #0, #1 and #2. Figure 9 shows the effects of *logical-page-delete* function. It improves our *h*FTL in every case.

[2] In traditional file systems, when unlinking a file, the file system only deletes the metadata. So the data pages belonging to the deleted file, still exist in the physical page of NAND flash pretending to contain valid data even though the pages has no valid data any more logically. We call this page as deleted page.

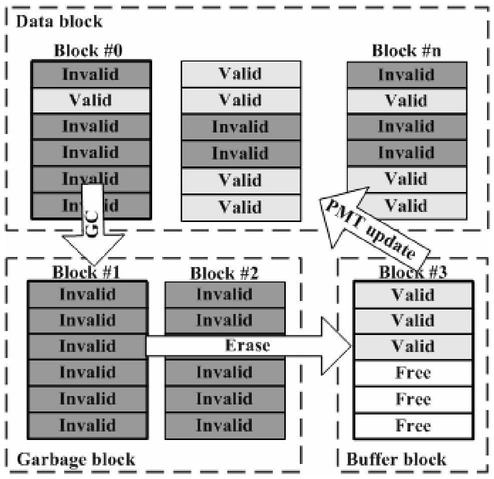


Figure 7. Type of physical NAND blocks

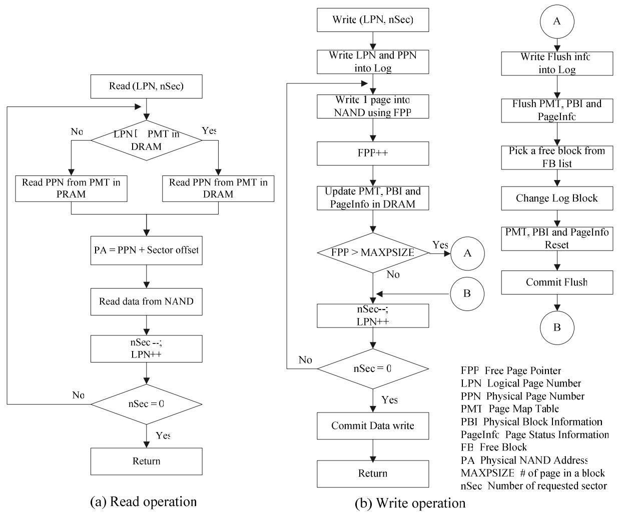


Figure 8. Read/Write operation of *h*FTL

Second, we found significant performance degradation when the storage utilization is almost close to 100%, which means the number of garbage or free blocks to be used in merge operation goes under the given threshold. Thus, the *h*FTL tries to perform merge operation frequently. To avoid this problem, we set a certain number of blocks aside as reserved blocks. These blocks are not counted for logical space. The percentage of reserved

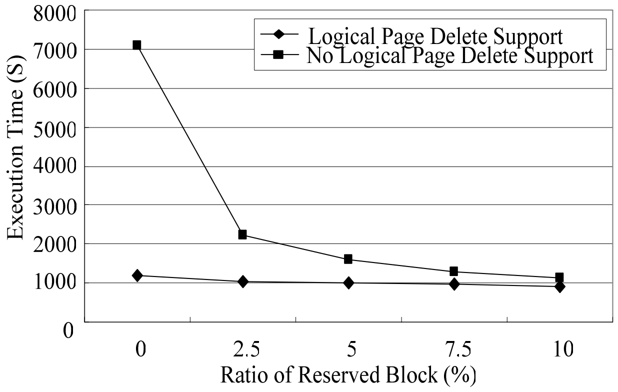


Figure 9. Performance enhancement of logical delete and reserved block

blocks in total blocks has large impact on the performance. Figure 9 also shows the performance enhancement according to the percentage of reserved blocks. As shown in Figure 9, the performance enhancement largely depends on whether the *logical-page-delete* function is supported or not. Even in the case of *logical-page-delete* function support, the performance of 10%-reserved block is 22% higher than that of 0%-reserved block. Our experiment shows that 5%-reserved block seems enough to avoid severe performance degradation in normal applications, no matter how the file system and FTL support *logical-page-delete* function. So we set the reserved block percentage to 5% for our all experiments. For the other miscellaneous functions of the FTL such as bad block management and wear-leveling, the *h*FTL follows those of the conventional FTLs.

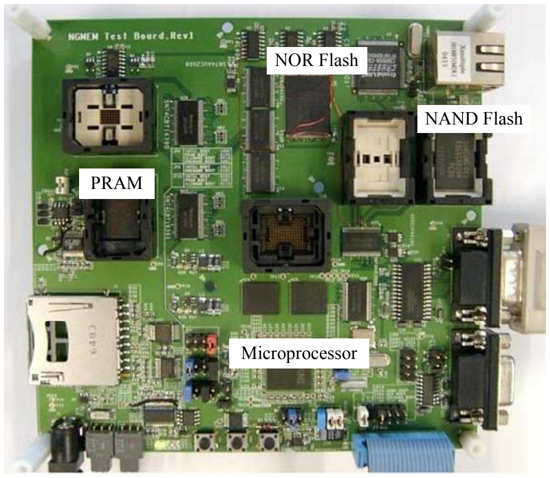
5. EVALUATION

5.1 Experimental setup

In this section, we evaluate our two proposed schemes with a conventional NAND flash-based storage system. For this purpose, we developed a test board as shown in Figure 10. It equips an S3C2413 (ARM926) microprocessor, one 1GB MLC (Multi-Level Cell) NAND flash, and one 64MB PRAM. The access timing information of each memory device is presented in Table 1. The read/write access timing includes data transfer time between the microprocessor and page register in NAND flash as well as transfer time between the page register and NAND flash cells. Compared to SLC (Single-Level Cell) NAND flash, MLC NAND flash has longer cell operation time but its density is double of SLC NAND flash. Due to these merits, MLC NAND flash becomes more attractive for data storage in embedded systems. However, its poor performance will be a burden for system designers. Thus, we choose MLC NAND flash as our target device instead of SLC NAND flash. But we note that our idea can be also applied to SLC NAND without any modification.

Table 1. Access timing for MLC NAND and PRAM [21,22]

Memory device	Access time		
	Read	Write	Erase
MLC NAND	121*us* (page)	861*us* (page)	1.5*ms* (block)
PRAM	80*ns* (word)	10*us* (word)	N/A



Microprocessor	Samsung S3C2413 - 200MHz
NAND Flash	Samsung K9G8G08UOM - 1GB (1 channel, 1 way)
NOR Flash	Intel JS28F128 - 16MB
PRAM	Samsung KPS1215EZM - 64MB
DRAM	Samsung K4S51163PF - 64MB

Figure 10. Prototype implementation

The evaluation is conducted to show read/write performance, NAND flash block erase count which directly affects wearing-out of NAND flash storage, and implementation cost focusing on required memory space. A log block FTL [5] (LBFTL) and a FAT-based file system are chosen as reference FTL and file system which are optimized for embedded system including power-failure recovery. LBFTL uses 4 log blocks and FAT file system uses 2KB sized cluster.

5.2 Performance evaluation

We measure the performance in two layers: FTL and file system. Before the performance evaluation, we measure the LLD performance which indicates the maximum read/write performance that the hardware can provide. LLD write performance is 1.73MB/s and read performance is 4.24MB/s. These values seem to be lower than the maximum performance that can be calculated using Table 1. This is mainly because the NAND controller of the test board is not fully optimized for MLC NAND flash. However, this does not matter since all configurations are evaluated on the same test board and we use their relative differences as performance metrics. Figure 11 depicts the performance comparison of LBFTL and *h*FTL. Due to the FTL overhead, overall performance is lower than LLD level performance. In order to show the performance degradations, we also present LLD level read/write performance as dashed lines in the figure. The performance of both FTLs reaches LLD level performance for sequential access patterns. For random write access patterns, our *h*FTL always shows stable performance while LBFTL shows poor performance for random access patterns. The performance of LBFTL is getting lower as the I/O request unit size is getting smaller. The reason is that the LBFTL is based on

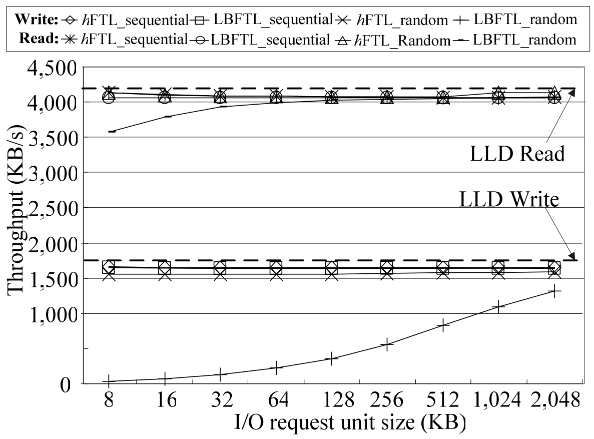
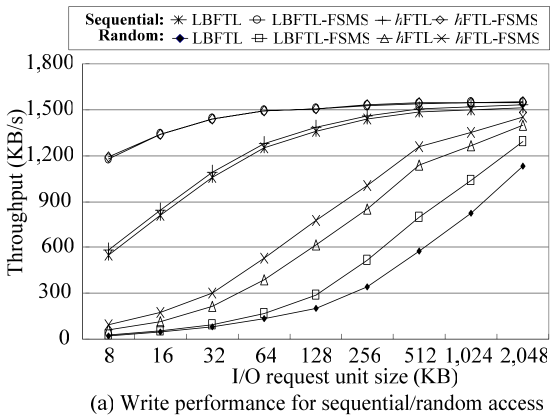


Figure 11. Performance evaluation in FTL level

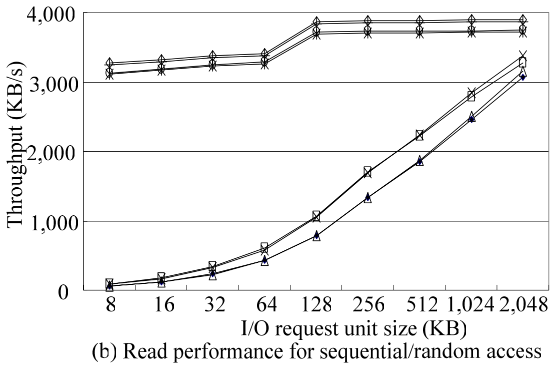
log block mapping scheme where the smaller size of I/O request unit increases the number of merge operations, while our *h*FTL is based on page mapping scheme where the number of merge operation is not affected by I/O request unit size. As a result, it shows up to 42 times better performance than that of LBFTL at the 8KB I/O request unit size. Figure 12 presents the file system level performance of four schemes; log block FTL (LBFTL) as a baseline configuration, log block FTL with FSMS (LBFTL-FSMS), *h*FTL (*h*FTL) and *h*FTL with FSMS (*h*FTL-FSMS). Again, we note that the FAT-based file system is layered on top of the FTL for all configurations. We use IOZone file system benchmarking software [18] which is widely used for evaluating the file system performance. We set test storage capacity to 1GB and I/O request unit size varies from 8KB to 2MB.

For sequential read patterns, the performances of proposed *h*FTL and *h*FTL-FSMS are slightly higher than those of LBFTL and LBFTL-FSMS. For random read patterns, LBFTL-FSMS and *h*FTL-FSMS slightly outperform LBFTL and *h*FTL. This is mainly because the seek operation of file system for random read requires frequent metadata read, and the FSMS scheme can save time in reading metadata due to the fast read performance of PRAM. Similarly, we observe that the performances of LBFTL-FSMS and *h*FTL-FSMS are higher than those of LBFTL and *h*FTL for sequential write access. Even though the test workload has a sequential pattern, the file system makes a lot of metadata writes, which makes sequential workload pattern into random access pattern in the view of FTL, as we pointed out in Section 3. If the FSMS scheme is applied, the sequential access pattern of test workload is delivered to the FTL as it is. So FSMS brings great benefits in this case. We expected that performance of *h*FTL should be better than that of LBFTL for sequential access since file system metadata makes access pattern to FTL into random access pattern. However, they show almost same performance because of the simple access pattern which means the number of accessed spots in logical address space is small For this simple access pattern, LBFTL is already optimized using enough log blocks to absorb its negative effects. LBFTL-FSMS and *h*FTL-FSMS show better performance than LBFTL and *h*FTL by up to 63%.

For random write on which we mainly focused, the *h*FTL-FSMS shows the highest performance for the all I/O request unit sizes since our proposed FSMS and *h*FTL take effects orthogonally.



(a) Write performance for sequential/random access



(b) Read performance for sequential/random access

Figure 12. Performance evaluation in file system level

The *h*FTL-FSMS increase the throughput up to 290% compared to LBFTL. Even without FSMS, *h*FTL shows higher performance than LBFTL-FSMS up to 130% because the benefit of *h*FTL originally designed to be robust for random access at FTL level, is greater than that of FSMS in random write. Although the performance gain becomes smaller as the I/O request unit size becomes larger, we note that our approach is effective for normal embedded systems whose I/O request unit size usually ranges from 4KB to 32KB. It shows much better performance than that of LBFTL at these ranges.

We also perform the evaluation with TIObench [20] benchmark but do not present them in this paper because the results are almost similar to those of IOZone benchmark. We expect more performance enhancement with *h*FTL if we separate the hot data from cold data in FTL level. This will be easily implemented in *h*FTL by increasing the number of current buffer block more than two.

5.3 Life span evaluation

In performing the evaluation of life span for storage system, test workload should reflect the dynamic behavior of the embedded systems. Since the workload of benchmark has limitation for this purpose, we collect the traces from our test systems. The traces reflect small/large size file creation/deletion, random read/write and sequential read/write which accumulate about 2GB write operations for NAND flash. And we run those traces on our test board and count the number of block erase operation for each scheme. As shown in Figure 13, both of FSMS and *h*FTL reduce the block erase count dramatically. The efficient metadata

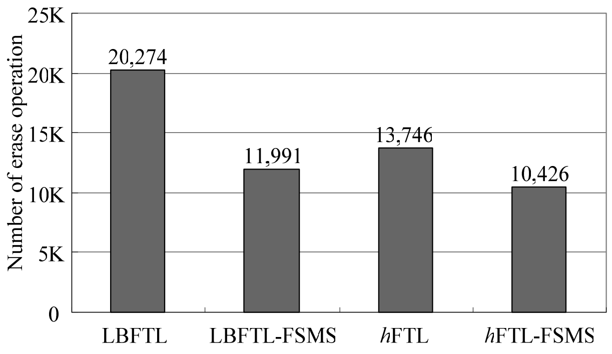


Figure 13. Block erase operation count

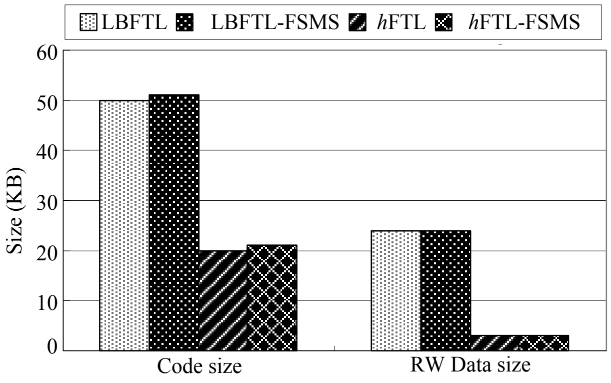


Figure 14. Memory requirement

management in FSMS contributes to reduce the number of merge operations. The mapping algorithm of *h*FTL also contributes to decrease the number of merge operations. Basically, the buffer block in *h*FTL is changed to the data block when there is no free page in it, which means the utilization of the buffer block is always 100%. However, in LBFTL, the limited number of log blocks lowers the utilization of log blocks when the access pattern is random [10], which causes frequent merge operations.

5.4 Implementation cost

Finally we evaluate our schemes with previous storage systems in terms of implementation cost. Figure 14 shows the implementation cost for each scheme. The code size of *h*FTL is reduced to 40% of LBFTL. The main reason is that the use of PRAM significantly reduces the efforts to manage the metadata and to handle the power-failure recovery. Compared to *h*FTL, the code size of *h*FTL-FSMS is slightly increased due to the code for separating the file system metadata. Although it is not always true that the small code size consumes less CPU computation, we observed that the *h*FTL requires less CPU computation power in run time. In addition, the *h*FTL keeps the most data structures on the PRAM rather than the main memory. As a result, the *h*FTL requires less amount of main memory. Depending on the storage capacity and configuration, the *h*FTL and *h*FTL-FSMS may require large amount of PRAM, and this may be a burden for system designer. In our experiment, LBFTL does not require PRAM while LBFTL+FSMS, *h*FTL and *h*FTL+FSMS require 5.47MB, 2.17MB, 7.64MB of PRAM respectively. However, it does not increase hardware implementation cost since our schemes assume to exploit the remaining space of the PRAM which is already equipped in the systems for the code storage. If the size of PRAM used for *h*FTL is bigger than the remaining size of PRAM, we can increase the logical page size into double or quadruple, which decreases the demand of PRAM by half or a quarter. In this case, there should be side effects in terms of performance, and the decision is a due of system designers.

6. CONCLUSION AND FUTURE WORK

In this paper, we proposed a NAND and PRAM hybrid storage solution and devised two approaches to exploit the hybrid storage architecture. First, we implement a file system metadata separation (FSMS) scheme to minimize the number of random write requests caused by inefficient metadata management. Second, we devised an efficient page mapping FTL (*h*FTL) exploiting the advanced features of PRAM, to enhance the random write performance. Finally, our performance evaluation using conventional benchmark tools demonstrates that the proposed hybrid architecture increase the performance up to 290% at file system level and reduce the number of erase operations up to 50% which directly contributes to increase the life span of NAND flash.

Our study contributes to verify the feasibility of PRAM as data storage as well as code storage and shows that the enhanced features of PRAM successfully complement the weakness of the conventional NAND flash-based storage systems. For future research, we are planning to design demand paging mechanism between NAND and PRAM to solve the scalability problem when the size of PRAM is not enough to hold entire metadata of file system and FTL.

7. REFERENCES

[1] R. Bez, E. Camerlenghi, A. Modelli and A. Visconti, , 2003. Introduction to Flash Memory, Proceeding of the IEEE, Vol 91, No 4, 2003

[2] G.H. Koh and et el., 2004. PRAM Process Technology. Proceeding of the IEEE International Conference on Integrated Circuit Design and Technology, 2004.

[3] Kinam Kim and G.H. Koh, 2004. Future Memory Technology including Emerging New Memories. Proceedings of 24th International Conference on Microelectronics, NIS, Serbia and Montenegro, May, 2004.

[4] Mun-Kyu Choi and et el., 2002. A 0.25um 3.0V 1T1C 32Mb Nonvolatile Ferroelectric RAM with Address Transition Detector and Current Forcing Latch Sense Amplifier Scheme. IEEE Journal of Solid-State Circuits 37, 2002.

[5] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y.Cho, 2007. A space-efficient flash translation layer for CompactFlash systems, IEEE Transactions on Consumer Electronics, 48(2), pp. 366-375 (2002).

[6] Mark DeVoss, 2007. The Winds of Phase Change are Blowing. Market Brief, iSuppli, May 2007.

[7] Ethan L. Miller, Scott A. Brandt and Darrell D. E. Long, 2001. HeRMES: High-Performance Reliable MRAM-Enabled Storage. in Proceedings of 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII), Schloss Elmau, Germany, pp. 83-87, May (2001).

[8] Nathan K. Edel, Ethan L. Miller, Karl S. Brandt and Scott A. Brandt, 2004. Measuring the Compressibility of Metadata and Small Files for Disk/NVRAM Hybrid Storage Systems. in Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems(SPECTS'04), San Jose, CA, July (2004).

[9] An-I A. Wang and et el., 2002. Conquest: Better Performance Through A Disk/Persistent-RAM Hybrid File System. in proceedings of the 2002 USENIX Annual Technical Conference, Monterey, June, 2002.

[10] S. Lee, and et el., 2007. A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation. ACM Transactions on Embedded Computing Systems, Vol. 6, No 3, July 2007.

[11] J.U. Kang, H. Jo, J.S. Kim and J.W. Lee, 2006. A Superblock-based Flash Translation Layer for NAND Flash Memory. EMSOFT'06 , Seoul Korea, October, 2006.

[12] Chin-Hsien Wu and Tei-Wei Kuo, 2006. An Adaptive Two-Level Management for the Flash Translation Layer in Embedded Systems. International Conference on Computer Aided Design (ICCAD'06), San Jose CA, November 2006.

[13] D. Roselli, J. Lorch, and T. Anderson, 2000. A Comparison of File System Workloads. USENIX Technical Conference, 2000.

[14] Intel Corporation, 1998. Understanding the flash translation layer (FTL) specification. http://developer.intel.com, 1998.

[15] A. Ban. Flash file system. United States Patent, No. 5,404,485, April (1995).

[16] C. Association, http://www.compactflash.org.

[17] EETIMES, Samsung introduces working prototype of PRAM http://www.eetimes.com (2006/9/11).

[18] IOZone benchmark, http://www.iozone.org.

[19] A. Birrel, M. Isard, C. Thancker, and T. Wobber, 2007. A design for High-Performance Flash Disks. ACM SIGOPS Operating Systems Review, Vol. 41(2), April 2007.

[20] Threaded I/O benchmark, http://sourceforge.net/projects /tiobench

[21] Samsung Electronics, Datasheet K9G8G08UOM, 2006.

[22] Samsung Electronics, Datasheet KPS1215EZM, 2006.