# Resource-Aware Sector Translation Layer for Resource-Sensitive NAND flash-based Storage Systems

Keonsoo Ha, Taejin Kim, Byoung Young Ahn, and Jihong Kim, *Member*, IEEE

**Abstract** — *As a need for high-density storage capacity increases on many high-end mobile devices such as smartphones, large NAND flash-based storage systems are more commonly used in such smart devices. For these storage systems, however, it becomes a challenge to use large NAND flash without incurring a large system overhead such as a large memory requirement. We propose a novel flash translation layer (FTL), called Resource-Aware Sector Translation Layer (RAST), which is optimized to reduce the memory footprint of an FTL for resource-sensitive storage systems. RAST is based on a hybrid mapping scheme which uses a group of blocks as a unit of mapping so that a small mapping table can cover a large number of blocks. RAST further saves the memory footprint by using an on-demand metadata management scheme which brings only recently accessed metadata into memory. RAST employs a sampling-based wear-leveling scheme which provides competitive wear-leveling performance with very small memory. Our experimental results show that RAST can achieve a good performance level for resource-constraint storage systems with the small memory footprint. For 32 GB NAND flash memory, RAST can achieve the write throughput of up to 57 MB/s using only 34 kB memory*[1].

**Index Terms — NAND Flash Memory, Flash Translation Layer, Data Structure, Mobile Storage System.**

## I. INTRODUCTION

As a need for high-density storage capacity increases on high-end mobile devices such as smartphones and tablet PCs, large-capacity mobile secondary storage systems are widely used. For example, latest smartphones and tablet PCs employ large NAND flash memory with a capacity of 16 GB to 64 GB. Although the storage capacity of these high-end mobile NAND-based storage systems has dramatically increased, their power and cost constraints have not been relaxed as much. For example, most high-end mobile storage systems are responsible for delivering their available maximum power budget to mobile storage systems. The storage systems can internally optimize performance under the strict power budget. Furthermore, since most high-end mobile devices such as smartphones and table PCs are consumer products, these mobile storage systems are also very cost-sensitive. Therefore, it is important to optimize the system's resource usage in designing these mobile storage systems.

In NAND flash-based storage systems, as the capacity of NAND flash memory grows, the required amount of memory for implementing a flash translation layer (FTL) also increases. In general, an FTL uses memory to store various metadata such as information about a mapping table, block types, block erasure counts, and the availability of pages in blocks. Since the size of metadata is increasing proportionally to the capacity of NAND flash memory, a large capacity storage system requires a large amount of memory space in storing metadata. For example, even in a memory-efficient mapping scheme such a block-level mapping scheme, 32 GB NAND flash memory with the 4 kB page size requires about 9 MB of memory to keep the required metadata of a block mapping-based FTL. Considering most resource-sensitive storage systems use less than 256 kB of SRAM, 9 MB memory requirement is not acceptable for most mobile storage systems.

One of the most commonly used methods to cope with the large memory requirement of FTLs in resource-sensitive storage systems is to use NAND flash memory to store the metadata of an FTL. In this case, the performance of a storage system is suffered significantly from frequent accesses to NAND flash memory to read metadata. Another common solution is to use SDRAM, which is often used as a data buffering purpose for improving storage performance. In this case, however, if a large portion of SDRAM is used for storing the metadata instead of buffering data, the performance of storage systems will be degraded significantly. Moreover, adopting the extra memory component may be a burden in resource-sensitive mobile storage systems in terms of cost and energy consumption.

In this paper, we propose a novel FTL, called *resource-aware sector translation layer* (RAST), which was specifically designed for resource-sensitive storage systems. RAST dramatically reduces the memory footprint of an FTL by minimizing the memory usage in implementing key functions of an FTL. RAST reduces the memory requirement of storing a mapping table by adopting a hybrid mapping scheme which operates at both page level and block-group level. This large granularity mapping scheme can cover a large number of blocks with limited memory space. RAST further saves the memory footprint by applying an on-demand

[1] This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2011-0020426, No. R33-10095, and No. 2011-0020514). The ICT at Seoul National University provided research facilities for this study.

Keonsoo Ha, Taejin Kim, and Jihong Kim are with the School of Computer Science and Engineering, Seoul National University, Gwanak-ro, Gwanak-gu, Seoul 151-742, Korea (e-mail:{air21c, taejin1999, jihong}@davinci.snu.ac.kr).

Byoung Young Ahn is with Indilinx, Inc., Yatap-dong, Bundang-gu, Seongnam-si, Gyeonggi-do 463-760, Korea (e-mail: eddie@indilinx.com)

Contributed Paper
Manuscript received 04/15/12
Current version published 06/22/12
Electronic version published 06/22/12.

metadata management scheme which maintains only recently accessed metadata in memory, thus requiring a very small-sized memory. In addition to the on-demand metadata management scheme, RAST performs the partial garbage collection process by exploiting long idle times of mobile storage systems in order to avoid performance degradation from excessive garbage collection. RAST also employs a sampling-based block allocation technique which contributes to the wear-leveling management with limited memory resource while minimizing the implementation overhead of a wear-leveling technique such as unnecessary NAND flash memory accesses.

In order to evaluate the proposed FTL, RAST, we have implemented RAST in both a trace-driven FTL simulator and a real platform board. We carried out experiments to evaluate the memory requirement, the write performance, and the wear-leveling performance of RAST. The experimental results show that RAST uses only about 34 kB of memory for 32 GB NAND flash memory-based storage systems. In terms of performance, RAST achieves write throughput up to about 57 MB/s with sequential write requests. Finally, the experimental results show that RAST has wear-leveling performance which is comparable to other existing memory-intensive wear-leveling schemes.

The rest of this paper is organized as follows. In Section II, we show that it is important for resource-sensitive storage systems to reduce the memory footprint of an FTL. In Section III, we review previous works related to memory usage in FTLs. We describe our proposed FTL, RAST, in Section IV. Section V shows the experimental results, and we conclude in Section VI.

## II. FTL Memory Requirements

FTLs use metadata to support its main functions such as address translation, free block allocation, wear-leveling management, and garbage collection. Typical metadata are maintained using a mapping table, a page status table, a block status table, and erasure count table. The mapping table contains translation information between a logical block address used in a host system and corresponding physical address in NAND flash memory. The page status table keeps track of information on the page availability. Before an FTL writes the requested data to a page, the status of the page in the page status table is checked by the FTL to find out whether the page has been written by previously requested data or not. The block status table indicates the current status of blocks. A block can be in one of four states: *free*, *clean*, *dirty*, or *dead*. This block state information is used when an FTL allocates a free block and performs a garbage collection process. The block erasure count table stores the number of performed erase operations of each block. This erasure count information is also necessary to perform wear-leveling management.

Although the size of the metadata necessary for an FTL implementation will be a function of various factors such as the total flash capacity, the mapping policy, and the wear-leveling management scheme, it is generally true that the larger NAND flash memory is, the higher FTL memory footprint is. TABLE I shows how the memory requirements of an FTL with various mapping units are used. We assume that 32 GB NAND flash memory is used with the 512 kB block size and the 4 kB page size. In TABLE I, a block group consists of four blocks. $N_{entry}$ indicates the number of entries in a mapping table, and $S_{Mapping}$, $S_{Page}$, $S_{Block}$, and $S_{Erasure}$ denote the required memory footprint for storing the mapping table, the status of pages, the status of blocks, and erasure counts of blocks, respectively. In the case of a mapping table, as the size of a mapping unit is getting larger, the required memory footprint becomes smaller.

TABLE I
MEMORY REQUIREMENTS FOR STORING METADATA OF FTLS

| Mapping Unit | $N_{entry}$ | Memory Requirements (kB) | | | | |
|---|---|---|---|---|---|---|
| | | $S_{Mapping}$ | $S_{Page}$ | $S_{Block}$ | $S_{Erasure}$ | Total |
| Page | 8 M | 32,768 | 0 | 256 | 256 | 33,280 |
| Block | 64 k | 256 | 8,192 | 256 | 256 | 8,960 |
| Block Group | 16 k | 64 | 8,192 | 16 | 16 | 8,288 |

Although the required memory footprint for storing a mapping table can be reduced by increasing the mapping granularity, however, the total required memory footprint is not decreased as much as the size of a mapping table is reduced. This is mainly because, unlike a page-level mapping table, a mapping table using a bigger mapping unit which is larger than a page does not maintain information about the availability of each page. Therefore, FTLs using a bigger mapping unit need to manage page status in extra memory. Because of the large-sized metadata, even an FTL using a block group as a mapping unit requires at least about 8 MB of memory. Since most resource-sensitive storage systems have a very small-sized SRAM whose size is ranging from 32 kB to 256 kB, the large memory requirement can be a burden in such storage systems.

Unfortunately, an FTL requiring large-sized memory cannot perform well under a limited memory constraint of resource-sensitive storage systems. In performing address translation, performance of an FTL decreases if the whole mapping table cannot be loaded into SRAM. In this case, only some parts of the mapping table can be loaded into SRAM and the rest of the mapping table must be stored in either SDRAM or NAND flash memory. SDRAM can be adopted in a mobile storage system as an option for a data buffering purpose if there is a need for improved performance. If SDRAM stores some portion of the mapping table, it is difficult to expect performance improvement by data buffering because available SDRAM space for data buffering decreases.

On the other hand, if the address information of the requested data has been stored in NAND flash memory, an FTL reads the NAND flash memory so that the address information can be loaded into memory. Although this approach is effective to reduce the required memory, the frequent accesses to NAND flash memory decrease the performance of a storage system. In particular, as the mapping unit gets smaller, an FTL is likely to read NAND flash memory more frequently. The reason is that as a mapping unit gets smaller, the range of addresses covered by a single mapping entry becomes narrower, so it is more likely that a requested address does not exists in the same size of memory. For this reason, the existing page-level mapping-based FTL using this on-demand approach [1] is not suitable for resource-sensitive storage systems.

Besides address translation, the tight memory resource budget may cause performance degradation in wear-leveling management. The basic approach of a wear-leveling algorithm is to allocate the youngest block with the minimum erasure counts when an FTL requires a free block. In this approach, the FTL has to keep track of erasure counts of all the blocks in memory. Although the required memory footprint for storing the erasure counts information is much smaller than that of a mapping table, 512 kB of memory requirement for keeping the information on erasure count of all blocks cannot be acceptable for the mobile storage systems. Similar to the data block mapping table, since the absence of the erasure counts information in memory causes extra accesses to NAND flash memory, the method which maintains erasure counts of all blocks and finds the youngest block among them is not suitable for such storage systems. From the discussions, above, it is clean that an FTL employed in resource-sensitive mobile storage systems should be designed carefully to satisfy performance requirement under a given resource constraint.

## III. Related Works

The memory requirement of an FTL has been regarded as a critical constraint in designing NAND flash memory-based storage systems. The page-level mapping scheme is an ideal solution in terms of performance if there is no limitation in resource usage. Although this scheme efficiently utilizes blocks within the flash, it requires a large mapping table to be stored in memory. As the size of NAND flash memory increases, the amount of required memory becomes a system burden. In addition to the translation information, other metadata also need memory space to be stored. Therefore, it is impractical to implement the page-level mapping scheme in mobile systems because of limited system resources.

As an alternative for systems with a limited resource environment, the block-level mapping scheme has been used, which reduces the size of a mapping table by a factor of the number of pages per block. However, since requested data must be stored in a particular page within each block, this block-mapping scheme cannot efficiently use a block, thus increasing required blocks for storing data. As a result, the block-level mapping scheme invokes frequent garbage collection processes, and the performance overhead by the garbage collection can dramatically decrease the performance of storage systems. Moreover, although the required data size decreases compared to the page-level mapping, the whole mapping table may not be small enough to be stored in memory if a target storage system has a small-sized memory.

In order to overcome the disadvantages of these mapping schemes, the hybrid mapping schemes based on both page-level and block-level mapping schemes have been proposed [2]-[4]. This hybrid mapping scheme uses a page-level mapping table only for updated data and a block-level mapping table for other data. Since the hybrid mapping scheme can place a given data in any offset within a block where the page-level mapping table covers, it can avoid excessive garbage collection overhead. Moreover, this scheme reduces the required memory footprint for storing mapping information by adopting a block-level mapping scheme. However, like the above mapping schemes, if the target system does not have enough memory space to keep both the mapping tables in the hybrid-mapping scheme, the scheme cannot be applied to the resource-sensitive systems.

As an enhanced page-level mapping scheme, an on-demand based page-level FTL (DFTL) [1] has been proposed. This scheme enables the page-level mapping scheme to be implementable with the limited memory by loading only requested mapping table entries in the memory. The FTL assumes that a high temporal locality exists in requested addresses. Thus, NAND flash memory may suffer from performance degradation by frequent read accesses if the temporal locality of the translated addresses is low. Moreover, since the scheme is based on the page-level mapping scheme, an entry covers only one address, thus limiting the range covered by loaded memory entries. As a result, DFTL is vulnerable to performance degradation when a sequential-dominant workload exists.

*u*-FTL [5] also reduces the required memory footprint for storing metadata of an FTL. This FTL implements a mapping table with a tree data structure called *u*-FTL which consists of mapping entries whose sizes can be varied. Since each mapping entry can translate the consecutive addresses of a sequential workload, *u*-FTL can reduce the size of a mapping table. However, since the size of this FTL is dependent on the workload pattern, the number of entries can reach that of a page-level mapping table under completely random workloads. Therefore, *u*-FTL is not implementable as well in the resource-scarce storage systems because the worst case, in terms of the required memory footprint, may occur.

Besides a mapping table, maintaining block erasure count information in memory is another challenging problem in mobile storage systems. If there is no block erasure information when a free block is requested by an FTL, a NAND flash memory must be accessed in order to find the youngest block. Since the whole block status information is too large to be stored in memory, a sampling-based block selection technique has been suggested [6]. This scheme holds a small number of candidate blocks and sorts them according to performed erasure counts. The candidate free blocks are randomly selected among free blocks. After allocating the youngest free block among the candidate blocks, some parts of the candidate blocks are replaced with newly selected free blocks. This scheme can reduce the amount of required memory for maintaining the certain number of free blocks. However, the selection based on a random function causes several extra NAND flash memory read operations if the block erasure count information is scattered in various pages. As a result, the response time of a NAND flash memory operation is extended corresponding to the status of metadata in NAND flash memory.

## IV. RAST: Resource-Aware Sector Translation Layer

We thus propose a novel FTL, namely RAST, which is designed for resource-sensitive NAND flash-based storage systems. Fig. 1 shows an overview of the target storage system. Its layout is similar to a typical solid state drive (SSD), which has multi-channel architecture, but the resource of the target storage system is not as abundant as an SSD. This target storage system consists of a NAND flash memory array, a processor, SRAM, and mobile SDRAM. The NAND flash memory array in the storage system is organized with four channels and four ways which can be operated in parallel. This storage communicates with an external host system through an interface.

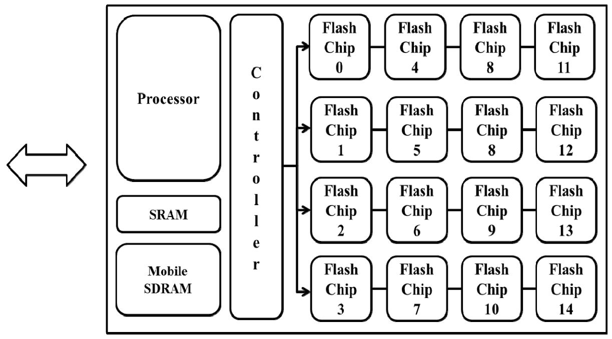

**Fig. 1. An architectural overview of our target storage**

RAST is a hybrid mapping-based FTL which uses page-level mapping scheme and block-group level mapping scheme for log blocks and data blocks, respectively. Fig. 2 shows the layout of memory and NAND flash memory of RAST. The memory is divided into three regions to store the log block mapping table, the data block mapping table, and the blocks status table. The data block mapping table stores mapping information as well as information about the availability of each page. The availability information is checked, in turn, after a requested address is translated using the data block mapping table. The block status table maintains both the status of block and the block erasure counts because they are used together during a garbage collection process.

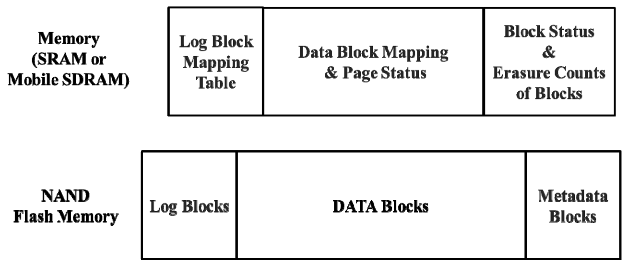
Memory (SRAM or Mobile SDRAM)
Log Block Mapping Table
Data Block Mapping & Page Status
Block Status & Erasure Counts of Blocks
NAND Flash Memory
Log Blocks
DATA Blocks
Metadata Blocks

**Fig. 2. The layout of memory and NAND flash memory of RAST**

On the other hand, NAND flash memory consists of three types of blocks such as log blocks, data blocks, and metadata blocks. Like existing hybrid mapping-based FTLs, updated data are stored in a log block. Besides an update request, since NAND flash memory has a restriction that the pages in a block must be written sequentially, a write request which violates the sequential write restriction is also written to a log block. The original mapping information of written data in the lob block is maintained in a page-level mapping table. With the exception of the two cases, other write requests are written to data blocks. A metadata block stores metadata of RAST. Since a metadata block also does not allow in-place update of data, RAST conducts a garbage collection process for the metadata blocks if there is no free page in the metadata block.

### *A. Large Granularity-based Hybrid Mapping Scheme*

As shown in TABLE I, the most effective way to reduce the required memory footprint is to employ a large mapping unit in a mapping table. In order to reduce the memory footprint, RAST uses a group of blocks as a mapping unit to map data blocks. The blocks located in the same offset in each channel are grouped into a block group. In Fig. 1, the blocks which are located in n-th offset of flash chips 0, 1, 2, and 3 are grouped into one block-group N. RAST can cover the addresses of the pages in the four blocks with just one logical block-group number, whereas existing hybrid mapping-based FTLs require four logical block numbers to map the same four blocks.

Considering I/O characteristics in mobile devices, it is reasonable for RAST to use a block group as a mapping unit. Since a mapping scheme based on a large granularity has a bad performance under the random write request, the performance of RAST may be suffered seriously from the random write requests. However, since most applications in mobile systems handle multimedia-rich applications and the multimedia applications access sequentially files and infrequently update files [7], RAST can save the required memory footprint without serious performance degradation. Although there are some random write requests which write previously written data again or violate sequential write restriction, log blocks can serve the random write requests without performance degradation if there is enough space to store new data in log blocks. In RAST, log blocks as large as possible are allocated for avoiding frequent garbage collection processes.

### *B. Partial Garage Collection Technique*

The behavior of garbage collection process in RAST is similar to that of *FTL using fully-associative sector translation* (FAST) [4]. Since a log block in a channel stores the data allocated to the same channel, the data in the log block are migrated to data blocks in the same channel during garbage collection processes. Since it may take a long time to complete all migrations without a pause in this approach, RAST utilizes long idle times of mobile systems for conducting garbage collection. In order to investigate a portion of idle time among the total execution time, we developed a custom mobile workload generation environment based on the representative usage scenario of mobile applications such as  the personal information management system (PIMS), the short message service (SMS), and the media players [8]. From our observations, many mobile systems are likely to have a long idle time, and the average idle time accounts for about 89% of the total execution time.

RAST carries out a partial garbage collection process when a long idle time is detected. If there is a longer idle time than a preset threshold, RAST copies part of data in a log block, which are associated with only one data block, to a new data block group. RAST distributes the overheads of garbage collection by conducting the partial garbage collection process across multiple times. In order to decrease the number of occurrences of partial garbage collection processes, the partial garbage collection process is performed if above 50% of log blocks are filled with data.

## C. *On-Demand Metadata Management Scheme*

Although RAST reduces the amount of required memory by improving mapping tables, it is not still affordable for a resource-sensitive storage system to keep all metadata in its small-sized memory. Instead of maintaining the metadata in memory, RAST manages the metadata excluding a log block mapping table by using an on-demand approach. Since data stored in a log block are likely to be accessed again compared to other data, the log block mapping table always stays in memory. As a result, some parts of block group level mapping table and the whole page-level mapping table are loaded on memory.

In the case of the data block mapping table, RAST keeps only some parts of mapping entries in the data block mapping table in memory. By limiting the number of mapping entries in memory, RAST has an upper bound of memory usage. Fig. 3 shows a snapshot of the on-demand management scheme in RAST. Each mapping entry in the data block mapping table has a logical block-group number, a physical block-group number, and a page status bitmap. A logical block group number is computed by dividing a logical block address from a file system by the number of pages in a block group. On the other hand, a bit in the page status bitmap is set to one if the page associated with the bit is written. The bits in a bitmap are set to zero when the block associated with the bitmap is erased.

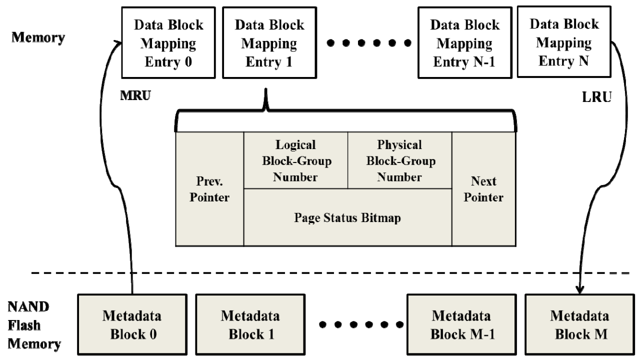
Memory
Data Block Mapping Entry 0
Data Block Mapping Entry 1
Data Block Mapping Entry N-1
Data Block Mapping Entry N
MRU
LRU
Prev. Pointer
Logical Block-Group Number
Physical Block-Group Number
Page Status Bitmap
Next Pointer
NAND Flash Memory
Metadata Block 0
Metadata Block 1
Metadata Block M-1
Metadata Block M

**Fig. 3. A snapshot of the on-demand metadata management scheme.**

As shown in Fig. 3, the mapping table entries loaded in memory forms a linked list of tracks according to recency of access. When a request arrives, RAST searches the linked list to find the mapping table entry which covers the requested logical block address. If it is a hit in the linked list, RAST utilizes the entry for address translation of the currently requested data. In the opposite case, RAST reads metadata blocks to load the requested mapping table entry into memory. At this moment, if there is not enough space in the memory to load new mapping entry, the mapping table entry located in the tail of the linked list is evicted to a metadata block. This evicted entry will be loaded again when a requested logical block-group address is covered by the evicted mapping entry.

As well as the data block mapping table, the rest of metadata are also managed by an on-demand approach. Compared to mapping table, a very small amount of memory is required to present information about the status of blocks and block erasure counts. This means that a page can store a wide range of metadata excluding the mapping tables. In Table 1, a block-group mapping scheme requires only 32 kB for storing them if 32 GB NAND flash memory is used. In this case, they are stored in only 8 pages in a metadata block, and each page can present the blocks status and erasure counts of 4096 block groups. Since each page can cover the metadata of a large number of blocks, RAST maintains only the information in a page among the several pages, and manages them with an on-demand approach.

## D. *Sampling-Based Wear-Leveling Management Scheme*

RAST induces block groups to be erased evenly by allocating a free block group in the sequence of a physical block-group number. As time goes by, however, the gap between the erasure counts of the oldest and the youngest block groups grows. In order to minimize the difference of the erasure counts, RAST maintains a sample of the physical addresses of relatively young block groups in an extra queue and gives a high priority to the young block groups in the sample when allocating a free block group. In order to find young block groups, RAST compares the erasure counts between the currently erased block group and the oldest block group whenever a block group is erased. Note that the erasure count of the oldest physical block group is maintained in RAST. If the gap of erasure counts of them is greater than a preset threshold, the erased block group is classified as a young block group. Since the young block group has to be allocated in the near future, RAST inserts a physical block-group number of the young block group into an extra queue, and the block groups in the queue are allocated prior to other block groups.

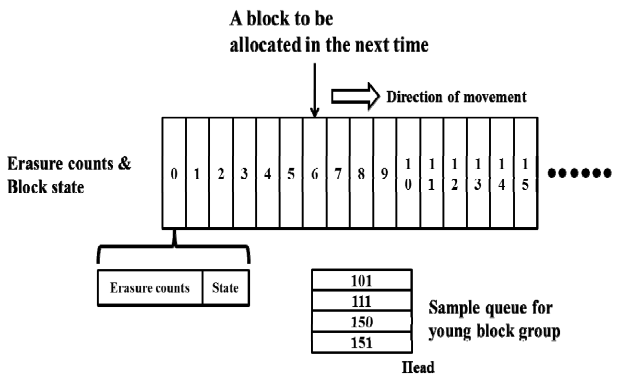
A block to be allocated in the next time
Direction of movement
Erasure counts & Block state
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Erasure counts
State
101
111
150
151
Sample queue for young block group
Head

**Fig. 4. An example of selecting a young block group to perform wear-leveling management in RAST.**

Fig. 4 shows a snapshot of young block-group selection. RAST keeps a pointer which points to a block-group number which is to be allocated in next allocation. When RAST allocates a free block group, if the sample queue is not empty, the block group in the queue is allocated, and it is removed from the queue. In Fig. 4, the block group 101 is allocated. If the queue is empty at that moment, the pointer takes a step forward after the pointed block group is allocated. By allocating block groups evenly and keeping young block groups separately, RAST performs efficient wear-leveling management.

## V. Experiments

### A. Experimental Environment

In order to evaluate RAST, we implemented it in both a trace-driven FTL simulator and an SSD prototype board. The prototype board was used to evaluate the memory footprint and performance of RAST, whereas wear-leveling performance was evaluated in the trace-driven FTL simulator. Fig. 5 shows a snapshot of the prototype board which has an embedded processor, 96 kB of SRAM, and 64 MB of SDRAM. In order to evaluate in a resource-sensitive storage system, SDRAM less than 1 MB and the whole SRAM were used for our experiments. We used 32 GB NAND flash memory array which consists of 16 MLC NAND flash memory chips. The size of a block is 512 kB, and each block has 128 pages. Experiments were performed using Iometer and an in-house I/O generator which can change workload characteristics such as the working set size and the access patterns.

\

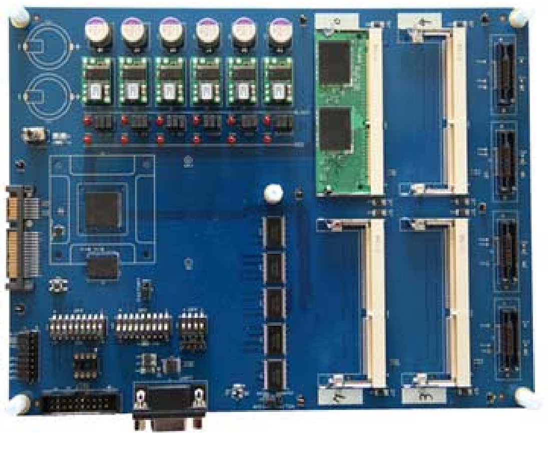

**Fig. 5. A snapshot of the prototype board used for our experiments.**

### B. Memory Requirement of RAST on the Real Platform

In order to judge the memory requirement of RAST, we used a cross-compiler which can generate the breakdown of memory usage of a compiled binary. Since RAST uses only static memory allocation, we could figure out the total memory requirement for metadata and code, respectively. Fig. 6 shows the breakdown of the memory usage of RAST. RAST requires only about 34 kB of memory which is as small as it can work with only small-sized SRAM. The memory footprint for a page mapping table accounts for 40% of the total amount of memory. Meanwhile, regions for a data block mapping table and the block status information in RAST take up only 32% of the required memory footprint. Since these regions are managed by using an on-demand scheme, the size of those memory regions can be changed according to system parameters.

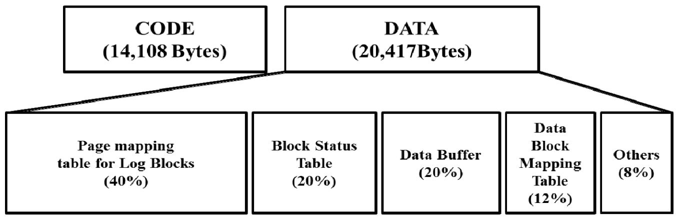

**Fig. 6. The breakdown of memory usage of RAST.**

### C. Performance Evaluation

In order to investigate the potential maximum performance of RAST, we ran *Iometer* in a raw device, which had not been formatted with a file system. *Iometer* has generated 4 kB write requests for one minute without a break in various patterns from sequential to random ones. TABLE II shows the write throughput and average IOPS of RAST with the different access patterns. The result shows that RAST write performance achieves up to 57 MB/s with sequential write requests, whereas it has weak performance with random write requests. Since RAST exploits inherent parallelism of the flash array, it has a competitive write performance with a sequential pattern. On the other hands, RAST has a very bad performance for random write requests because random write requests are likely to violate restrictions of NAND flash memory such as the sequential write restriction and in-place update. Since these data requests quickly consume free pages in log blocks, garbage collection processes happen without a break, thus decreasing performance rapidly.

TABLE II
WRITE THROUGHPUT AND IOPS OF RAST

| Pattern Type | Average Throughput (MB/s) | Average IOPS |
|---|---|---|
| Sequential | 57.1 | 14,616 |
| Random | 0.8 | 176 |

We generated random write requests to the target board with the in-house I/O generator in order to investigate the effectiveness of random write requests on performance of RAST. Fig. 7 shows the write throughput and the number of performed garbage collection processes with various working set sizes of random write requests. The x-axis presents the working set sizes of random write requests. The left y-axis gives the write throughput, whereas the right y-axis indicates the number of performed garbage collection processes. In terms of the write throughput, as the working set size is larger, the write throughput decreases from about 6 MB/s to less than 1 MB/s. On the other hand, the number of performed garbage collection processes describes the reason by showing it increases according to the working set size. Given that RAST uses only 8 MB of log blocks, the results mean that RAST can archive write throughput up to about 6 MB/s with random write request if the working set size is less than the that of log blocks.

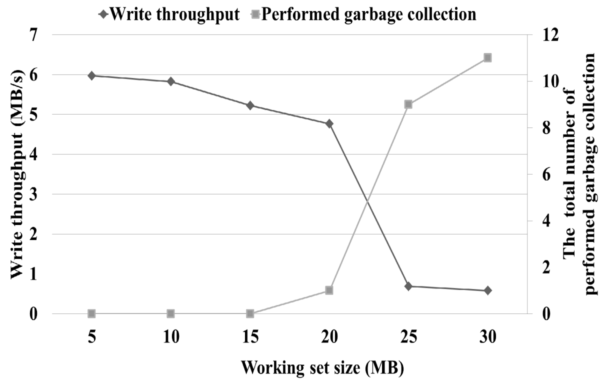

**Fig. 7. The write throughput and the number of performed garbage collection processes with various working set sizes of random write requests.**

In order to evaluate the effectiveness of the partial garbage collection technique of RAST, we measured both the average response time and the maximum response time with different idle times. *Iometer* has generated 4 MB random write requests with various time intervals ranging from 0 to 10 seconds. This time interval provides an idle time to RAST. TABLE III shows the response times of RAST with different idle times. From the idle time 0 to 7, both the average response time and the maximum response time are far longer than those of the idle time 10 seconds. These results mean that if idle time is as long as about 10 seconds, RAST can avoid a response time delay by excessive garbage collection processes.

TABLE III
RESPONSE TIMES OF RAST WITH DIFFERENT IDLE TIMES

| Idle Time (second) | 0 | 5 | 7 | 10 |
|---|---|---|---|---|
| Average Response Time (ms) | 8.51 | 9.89 | 8.17 | 1.94 |
| Maximum Response Time (ms) | 9121.42 | 9426.54 | 9404.20 | 251.08 |

### D. Wear-Leveling Performance Scheme

In order to evaluate wear-leveling performance scheme of RAST, we performed experiments in the trace-driven FTL simulator with a disk access trace collected from a running desktop PC for a month [9]. In order to induce block groups to be erased frequently, the size of NAND flash memory is set to 512 MB and the input trace is repeatedly given to the simulator.

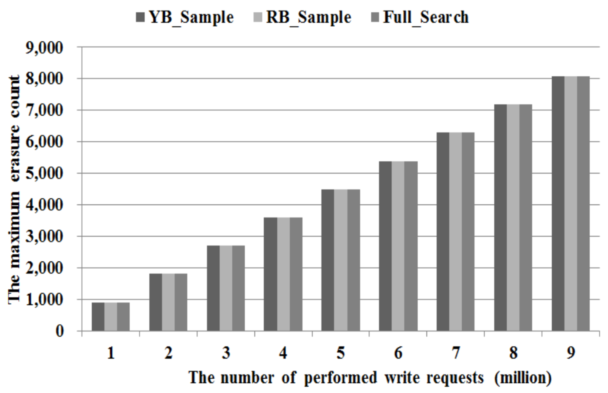

**Fig. 8. The maximum erasure count with different wear-leveling schemes**

Fig. 8 shows the erasure counts of the oldest block group in NAND flash memory with different wear-leveling schemes. The x-axis and the y-axis present the number of write request and the erasure counts of the oldest block group. Since our proposed technique makes a sample of young block groups, the scheme is denoted as YB Sample. On the other hand, Full Search and RB Sample [6] denote existing wear-leveling schemes used for comparisons. Full Search finds the oldest block groups among all block groups, whereas RB Sample scheme selects the youngest block group in a sample of randomly selected blocks. The results show that YB Sample has similar wear-leveling performance compared to other techniques. Since Full Search maintains erasure counts of all blocks, this approach cannot used for resource-sensitive storage system. In the case of RB Sample, it uses the same size of memory as that of YB Sample, but it accesses to NAND flash memory more frequently than YB Sample because it randomly selects some block groups for making a block group sample. In our experiment, YB Sample and RB Sample reads NAND flash memory on average 0.26 and 3.14 times, respectively, when allocating a free block group. These results mean that RAST performs competitive wear-leveling management without serious performance degradation.

## VI. CONCLUSION

We have proposed a novel resource-aware FTL, RAST. The proposed RAST FTL minimizes the memory footprint for address translation, metadata management, and wear-leveling management so that a large capacity NAND flash-based mobile storage systems can perform well while satisfying cost and power constraints. For reducing the required memory, our proposed FTL applies a hybrid mapping scheme with a large granularity mapping for data blocks. In addition to the mapping scheme, RAST keeps only recently accessed metadata into memory to reduce the memory footprint. Finally, RAST provides competitive wear-leveling performance with limited memory resource which is comparable to other wear leveler. Experimental results show that RAST can operate with a very small amount of memory without serious performance loss. Although the current version of RAST performs reasonably well, there still is a room for improvement. For example, since RAST uses a block group as a mapping unit, the overhead of a garbage collection process is very high. In order to reduce the overhead, we plan to employ the advanced data separation techniques such as hot cold separation techniques [10] and program context-based data separation technique [11].

## References

[1] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings," in *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, pp. 229-240, March 2009.

[2] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for compact flash systems," *IEEE Transactions on Consumer Electronics*, v. 48 n. 2, pp. 366–375, May 2002.

[3] S. Lee, D. Shin, Y. –J. Kim, and J Kim, "LAST: locality-aware sector translation for NAND flash memory-based storage systems," *ACM SIGOPS Operating Systems Review*, v.42 n.6, pp. 36-42, October 2008.

[4] S.-W. Lee, D. –J. Park, T. -S. Chung, D. –H. Lee, S. Park, and H. –J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Transactions on Embedded Computing Systems*, v. 6 n. 3, pp.18-es, July 2007.

[5] Y. -G. Lee, D. Jung, D. Kang, and J. -S. Kim, "μ-FTL: A memory-efficient flash translation layer supporting multiple mapping granularities," in *Proceedings of the 8th ACM international conference on Embedded software*, pp. 21-30, October 2008.

[6] B. Debnath, S. Krishnan, W. Xiaoy, D. J. Lilja, and D. H. C. Du, "Sampling-based garbage collection metadata management scheme for flash-based storage," in *Proceedings of the 27th IEEE Conference on Mass Storage Systems and Technologies*, pp.1-6, May 2011.

[7] H. Kim, Y. Won, and S. Kang, "Embedded NAND flash file system for mobile multimedia devices," *IEEE Transactions on Consumer Electronics*, v. 55, n. 2, pp. 545-552, May 2009.

[8] H. Verkasalo and H. Hämmäinen, "Handset-based monitoring of mobile subscribers," in *Proceedings of Helsinki Mobility Roundtable*, v. 6 n. 50, June 2006.

[9] L. -P. Chang and T. -W. Kuo, "An adaptive striping architecture for flash memory storage systems of embedded systems," in *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 187-196, September 2002.

[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 20th ACM symposium on Applied computing*, pp. 334-339, March 2005.

[11] K. Ha and J. Kim, "A program context-aware data separation technique for reducing garbage collection overhead in NAND flash memory," in *Proceeding of the 7th IEEE International Workshop on Storage Network Architecture and Parallel I/O*, May 2011.

## Biographies

**Keonsoo Ha** received the B.E. degree in information and communication engineering from Sungkyunkwan University, Korea, in 2005. He is currently working toward the Ph.D. degree at Seoul National University. His research interests include storage systems, operating system, and embedded system software.

**Taejin Kim** received the B.E. degree in computer engineering from Sungkyunkwan University, Korea, in 2010, and the M.E. degree in computer science and engineering from Seoul National University, Korea, in 2012. He is currently working toward the Ph.D. degree at Seoul National University. His research interests include storage systems and hardware design.

**Byoung Young Ahn** received the B.E. degree in computer engineering from Seoul National University, Korea, in 1999. He is a firmware engineer in Indilinx Inc. From 2000 to 2005, he was a software engineer in NCSoft Co., Seoul, Korea. From 2005 to 2008 prior to coming to Indilinx, he worked at Memory Division in Samsung Electronics as a lead programmer for core FTL. His research interests include file systems and software for storage systems.

**Jihong Kim** (M'00) received the B.S. degree in computer science and statistics from Seoul National University, Seoul, Korea, in 1986, and the M.S. and Ph.D. degrees in computer science and engineering from the University of Washington, Seattle, WA, in 1988 and 1995, respectively. Before joining SNU in 1997, he was a Member of Technical Staff in the DSPS R&D Center of Texas Instruments in Dallas, Texas. He is currently a Professor in the School of Computer Science and Engineering, Seoul National University. His research interests include embedded software, low-power systems, computer architecture, and storage systems.