Adaptive Log Block Mapping Scheme for Log Buffer-based FTL (Flash Translation Layer)

Duckhoi Koo and Dongkun Shin
Sungkyunkwan University, Korea

***Abstract*—Flash memory has been widely used as an important storage device for consumer electronics. For the flash memory-based storage systems, FTL (Flash Translation Layer) is essential to handle the mapping between a logical page address and a physical page address. Especially, log buffer-based FTLs provide good performances with small-sized mapping information. In designing the log buffer-based FTL, one important factor is to determine the mapping architecture between data block and log block, called associativity. While previous works use static associativities fixed at the design time, we propose a new log block mapping scheme which adjusts the associativity considering the irregularity of I/O workload. Our proposed scheme improves the I/O performance by 5~15% compared to the static scheme by adjusting the associativity based on the run-time workload.**

Keywords- flash memory; flash translation layer; log buffer; hybrid mapping; embedded storage

I. INTRODUCTION

NAND flash memory has become the most important storage media in the mobile embedded systems such as MP3 players, digital cameras and mobile phones due to its low power consumption, non-volatility, reliability, and physical shock resistance [1]. Recently, as the cost of NAND flash memory has been decreased and the capacity of it has been increased, NAND flash-based SSD (Solid-State Disk) is about to be used in laptop computers and enterprise server systems where energy efficiency is important.

Unlike a traditional hard disk, NAND flash memory does not support "overwrite" operation because of its "erase-before-write" characteristic. When the data at a certain page should be modified, we cannot overwrite the page. Instead, the new data must be written to another free page and the old data must be invalidated.

This feature of NAND flash memory requires two storage management schemes: address mapping and garbage collection. The address mapping scheme is to map a logical address from the file system to a physical address of the flash memory by maintaining the address mapping table. The garbage collection scheme makes it possible to reclaim the invalidated pages by erasing the corresponding block after copying valid pages in the block to a free block. To support these two management schemes, a software layer called an FTL (Flash Translation Layer) is used between the file system and the flash memory [2].

In general, FTL schemes can be classified into three schemes according to the method of address mapping: block-level mapping, page-level mapping, and hybrid mapping. In block-level mapping [3], a logical block is mapped to a physical block in flash memory. A page is written by in-place, which means it is placed at the same offset in both the logical block and the physical block. Although the block-level mapping requires a small-sized mapping table, it invokes a large amount of block copy overhead even though only a small portion of a block is changed since it should move all non-updated pages into new free block to maintain the block-level mapping.

In page-level mapping [4], a logical page is mapped to any physical page thus it can be written by out-of-place. So, it invokes no overhead when a page is updated thus provides a high performance. But, it requires a huge memory space for page-level mapping information.

In order to overcome such problems of page-level mapping and block-level mapping, hybrid mapping that uses both page-level mapping and block-level mapping was proposed. Hybrid mapping assigns a small portion of flash memory blocks for log buffer. So, it is called a log buffer-based FTL. The log buffer is temporal storage spaces to be used for overwrite operation. A log block in the log buffer can be used for one or several data blocks. So, data block and log block are associated with each other. The associativity of a log block means the number of associated data blocks of the log block. If there is an update request on the data which is already written at the data block, the new data is written at the associated log block and the old data in the data block is invalidated. Normal data blocks are managed by block-level mapping but the log blocks in log buffer are handled by page-level mapping. Thus, hybrid mapping requires a small-sized mapping table and invokes little block copy overhead.

When there is no free space in the log buffer, FTL has to make new free space by merging a log block with its associated data blocks. This process is referred to log block merge and has following sequences. First, it selects a victim log block to be merged and copies all valid pages in either the victim log block or its associated data blocks into reserved free blocks. Then, it modifies the logical block mapping information by replacing the associated data blocks with the allocated free blocks. Finally, the victim log block and its associated data blocks are erased and become free blocks. Since the log block merge operation requires a large number of page copies and block erases, it is important to reduce the number of merge operations to achieve a good performance in FTL.

There are three types of merge operation: switch merge, partial merge, and full merge [5]. Switch merge can be performed only when all pages in the victim log block are written by in-place scheme. Its merge cost is cheap because it just requires one erase operation without any page copy operation. Partial merge is similar to switch merge, but it requires additional page copy operations. It is also performed

when pages in the victim log block are written by in-place scheme, but there are free pages in the victim log block. On the other hand, full merge is performed when pages in the victim log block are written by out-of-place scheme. The full merge cost is expensive since all the valid pages in the victim log block and its associated data blocks should be moved. The full merge cost of a log block L is calculated as follows:

$$N \cdot A(L) \cdot C_{copy} + (A(L)+1) \cdot C_{erase} \quad (1)$$

where N, $A(L)$, C_{copy}, and C_{erase} represent the number of physical pages in a flash block, the associativity of the log block L, the cost of one page copy, and the cost of one block erase, respectively. In hybrid mapping, therefore, not only reducing the number of log block merge operations but also executing switch merge or partial merge rather than expensive full merge improve the performance.

The number of log block merges and the average cost of log block merge depend on the associativity of log block. Therefore, it is important to find the optimal associativity in order to reduce the log block merge overhead. Although several association schemes are recently proposed, they use static schemes which select fixed associativities at the design time not considering the run-time workload change. In this paper, we propose an adaptive scheme, called A-SAST, which adjusts the associativity between data blocks and log blocks depending on the run-time workload to optimize the log block merge cost. Experimental results show that our scheme increases the performance by 5~15% over the previous scheme which uses the static best associativity selected by considering the target workload pattern at the design time.

The organization of the remainder of this paper is as follows: The next section reviews related works. Section 3 provides detailed descriptions of proposed A-SAST scheme. Then, performance evaluation results of A-SAST are presented in Section 4. Finally, Section 5 concludes with a summary of this paper.

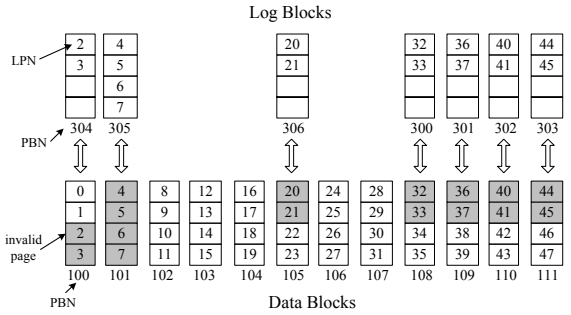
II. RELATED WORKS

There are several kinds of log buffer-based FTL schemes, which can be classified into BAST [5], FAST [6], KAST [7], and SAST [8, 9] depending on the association mechanism.

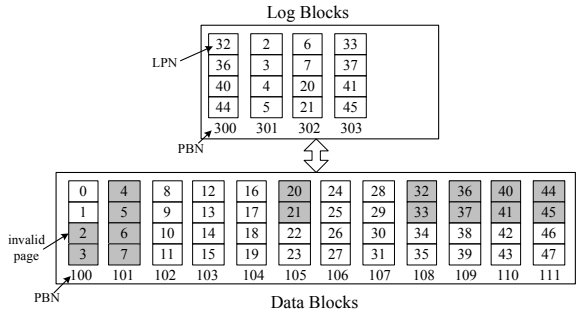
In BAST (Block-Associative Sector Translation) scheme, a log block is used for only one data block at a time as shown in Fig. 1(a). We assume that one block consist of four pages. If there is a write request for the block which is not associated with any allocated log block and there is no free log block, one of allocated log blocks should be merged to make space for a new log block. Therefore, there are frequent log block merges especially when the write pattern is random. Moreover, a log block is merged with its low utilization. This is called a log block thrashing problem.

To solve the log block thrashing problem, FAST (Fully-Associative Sector Translation) scheme was proposed, where a log block can have the updated pages of any data block as shown in Fig. 1(b). Therefore, even when the write pattern is random, there is no frequent log block merges and the log block utilization is high. However, FAST scheme requires a large overhead per one log block merge since it can be associated with many data blocks.

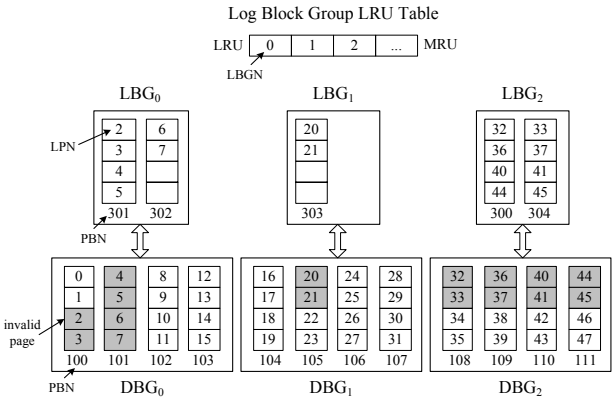
To mitigate disadvantages of both BAST and FAST, KAST (K-Associative Sector Translation) and SAST (Set-Associative Sector Translation) schemes were suggested. KAST is similar to FAST but the maximum associativity of log block can be configured to any value at design time. Thus KAST can limit the maximum merge cost to the predefined value.



(a) BAST scheme



(b) FAST scheme



(c) SAST scheme (N=4, K=2)

Fig. 1. Hybrid mapping schemes

SAST is a compromise between BAST and FAST. The SAST scheme groups N number of sequential data blocks into a data block group (DBG). One DBG can be associated with only one log block group (LBG) that has K number of log blocks at maximum. So, N+K number of physical blocks can be allocated for N number of logical blocks, thus SAST is referred to N:N+K mapping. If both N and K are 1, the SAST scheme is same to the BAST scheme. We can make another

extreme case of SAST by grouping all data blocks into one group and it is equal to FAST. A log block of an LBG can have the updated pages of any data block of the associated DBG.

Fig. 1(c) illustrates the SAST scheme that uses 4:4+2 mapping. One data block group consists of four sequential data blocks and one log block group consists of two log blocks. In Fig. 1(c), LBG_0 and LBG_1 are associated with DBG_0 and DBG_1, respectively. For example, the physical blocks with PBN (physical block number) 100, 101, 102, and 103 are allocated as data blocks for DBG_0, and the physical blocks with PBN 301 and 302 compose LBG_0 which is associated with DBG_0. When the pages with the logical page number (LPN) 2, 3, 4, 5, 6, and 7 are updated, they are written at the log blocks in LBG_0 and the old pages in DBG_0 are invalidated.

When there is no free log block to be used for a DBG, the least recently used (LRU) log block group is selected for a victim to be merged. Generally, since the LRU log block has many invalid pages and there is little possibility to be updated, it is profitable to select the LRU log block in order to reduce the log block merge cost. For that purpose, the SAST scheme maintains the log block group LRU table.

In SAST, the values of *N* and *K* have a significant influence on the FTL performance and the optimal values for *N* and *K* depend on the I/O request pattern. We can find the optimal values by exhaustive simulation for target I/O workloads or workload analysis technique proposed in SAST [8]. To find the proper value for *N*, the workload analysis technique examines the request density of a logical block, which is the ratio of the number of requests accessed in the logical block to the total number of requests. If the request density is high, it means there is a high spatial locality for the logical block. So, a large value should be used for *N*. The proper value for *K* is determined by measuring the temporal locality. If there are many updates, SAST uses a large value for *K*.

However, the optimal values of *N* and *K* are changed during the run time and they are different depending on the logical address since the I/O pattern is varied according to the execution time and the logical address. But, the values of *N* and *K* are equally applied throughout all the address space and the values are fixed during the execution time in the SAST scheme. Moreover, it is difficult to know the run-time workload for analysis at design time. Therefore, it is necessary to develop an adaptive scheme which can change the sizes of DBG and LBG depending on the run-time I/O workload.

III. ADAPTIVE SAST SCHEME

We propose an adaptive SAST (A-SAST) scheme which can solve the problems of SAST scheme. We have made three kinds of improvements on SAST as follows:

- First, the selection policy for victim log block is improved.
- Second, we make it possible to adjust the size of each DBG to the optimal value according to I/O pattern by merging two DBGs into one DBG or splitting one DBG into two DBGs at run time.
- Third, we eliminate the constraint for the maximum number of log blocks in LBG, i.e., the value of K. So, an LBG can have any number of log blocks ($K=\infty$) though the total number of log blocks is fixed.

A. Victim Log Block Selection

In the original SAST scheme, the LRU table is maintained at the log block group level. So, the accuracy is low because a write request for a log block changes the LRU levels of all the log blocks in the corresponding LBG.

For instance, in Fig. 1, when write requests for several logical pages occur as the order of (32, 36, 40, 44, 2, 3, 4, 5, 6, 7, 20, 21, 33, 37, 41, 45), the log block at PBN 300 is the oldest one. However, the log block at PBN 301 is selected as a victim log block by the group-level LRU policy.

A-SAST uses the block-level LRU policy. In addition, it considers the merge cost as well as the LRU level when selecting a victim log block. Fig. 2 shows an example of the block-level LRU table of A-SAST. The victim log block is selected within the victim log block region which includes *M* number of little-recently-used log blocks. Among the log blocks in the victim region, the log block with the lowest merge cost is selected as a victim log block. The merge cost can be estimated using Equation (1).

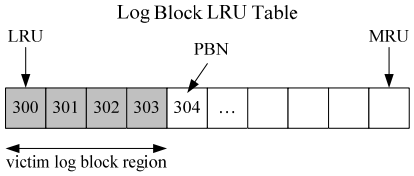


Fig. 2. LRU table in A-SAST scheme.

In Fig. 2, the PBN 300 is the LRU log block, but PBN 302 is selected as a victim log block because it has the lowest merge cost among log blocks in the victim log block region. Because $A(L)$ of PBN 300 is 4, we should copy 16 pages and erase 5 blocks to merge the log block. On the other hand, $A(L)$ of PBN 302 is 1. Even though a log block at the outside of victim region has a lower merge cost than all the log blocks in the victim region, it is better to select one within the victim region to reduce the merge cost. This is because the pages in a log block which is recently used have the high possibility to be invalidated by the future write requests. Since a log block in the victim window that has the highest associativity may not be selected permanently due to its high merge cost, A-SAST selects the log block if it has remained in the victim region too long time. Using such policy, we can consider both the log block LRU level and the merge cost.

In A-SAST, an LBG can have any number of log blocks as long as the total number of log blocks is smaller than the total allowed log buffer size. So, we do not need to find the proper value of *K*. Instead, the block-level LRU policy for log buffer automatically controls the proper number of log blocks. That is, if a DBG needs a large number of log blocks due to its high temporal locality, the victim selection policy takes a log block from the DBG with a low temporal locality and gives the log block to the DBG with a high temporal locality.

B. The effect of Data Block Group Size

The optimal size of a DBG is related to the log block utilization and the log block merge cost. Generally, as we increase the size of data block group, the log block utilization increases because several data blocks can share a log block. But, the log block merge cost also increases because the associativity of a log block increases. Therefore, if an LBG has a low utilization, which means that the associated DBG

cannot utilize log blocks effectively, it is better to use a larger value for the size of corresponding DBG since it has a low request density. If an LBG has too high associativity, which increases the average merge cost, a smaller value is proper to the size of corresponding DBG since it has a random request pattern.

C. Data Block Group Reorganization

A-SAST adjusts the size of each DBG to adapt the I/O pattern of the corresponding address range and the I/O pattern change at run time. The adjustment is performed by merging or splitting data groups.

For example, in Fig. 1, LBG_0 and LBG_1 consume small numbers of pages thus have low utilizations. This is because there were little updates for DBG_0 and DBG_1. In this case, if we create a new larger data block group by merging DBG_0 and DBG_1 and assign one log block group to the merged DBG, the log block utilization will increase. On the other hand, the LBG_2 has a high utilization but the merge cost for each log block is high because each log block is associated with several data blocks. This means that the write pattern for DBG_2 is quite random. If we split DBG_2 into two small data groups, the merge cost for each log block can be reduced.

Fig. 3 shows how A-SAST changes the block groups by merging and splitting for the example in Fig. 1. By merging DBG_0 and DBG_1, a new DBG which consists of 8 data blocks is created and it requires only two log blocks. As a result, it increases the log block utilization and saves one log block. Merging data block groups is performed when the following conditions are satisfied for two consecutive DBGs, DBG_i and DBG_j.

$$\frac{P_{used}(\phi(DBG_i))}{P_{total}(DBG_i)} < \alpha \text{ and } \frac{P_{used}(\phi(DBG_j))}{P_{total}(DBG_j)} < \alpha \text{ and}$$

$$\forall L,\ L \in \phi(DBG_i) \cup \phi(DBG_j), \quad A(L) < \beta$$

where $\phi(g)$, P_{used}, and P_{total} are the associated log block group of data block group g, the number of used pages and the number of allocated pages for DBG or LBG, respectively. The first condition checks the utilizations of two LBGs and the second condition checks the associativities of all log blocks in the LBGs. The second condition prevents DBGs from being merged when they have random write request patterns.

The group merge condition is examined for the DBG whose one of associated log blocks is selected as a victim log block. By merging data block groups which have small number of updates and sequential write patterns, we can use the log blocks more efficiently.

In Fig. 3, DBG_2 is split into two data block groups. If write requests for each page occur as the order of (32, 36, 40, 44, 33, 37, 41, 45), the merge cost of each log block is reduced by half. Like this, when the log block association is high due to many random updates for a certain group, the performance will be enhanced by splitting the DBG. The condition for DBG splitting is as follows:

$$A(L) > \gamma$$

where γ is the split threshold and L is the lastly written log block of the log block group. We check the condition when we should allocate a new log block for the DBG.

The optimal values of α, β, and γ depend on the workload pattern. We provide the experiment results to explore the optimal values in Section IV.

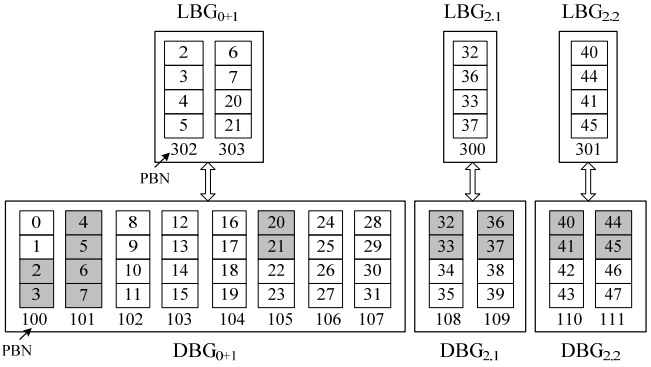


Fig. 3. Adjusted groups in A-SAST.

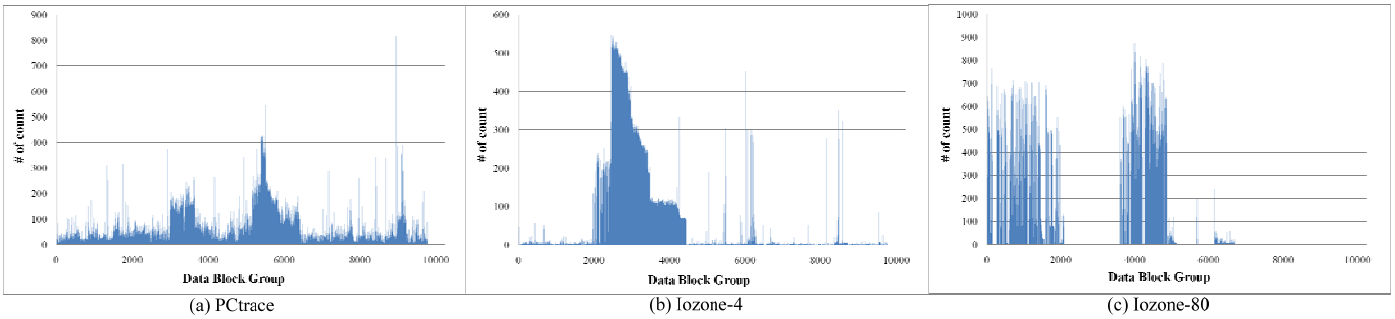
IV. EXPERIMENTS

Our proposed schemes are evaluated using simulation. Three I/O workloads are used: PCtrace, Iozone-4, and Iozone-80. PCtrace workload is collected in the Windows system by executing the applications such as the word processor, moving picture, web browsing, games, and so on. Iozone-4 and Iozone-80 are collected using Iozone benchmark program [10]. While Iozone-4 generates 1~4 KB write requests thus has a random I/O pattern, Iozone-80 generates 2~80 KB write requests thus has both random and sequential I/O requests. We assume that each block of NAND flash memory is composed of 64 pages and the page size is 2 KB. The number of total log blocks is 256. The timing parameters for NAND flash operations are based on [11].

We first observed the irregular behavior of log block merges under the original SAST scheme. Fig. 4 shows the total number of log block merges for each data block group under the SAST scheme. From the results, we can know that the number of log block merges significantly different depending on the address space. Therefore, we can know that the proper values for the group size are different depending on the address space.

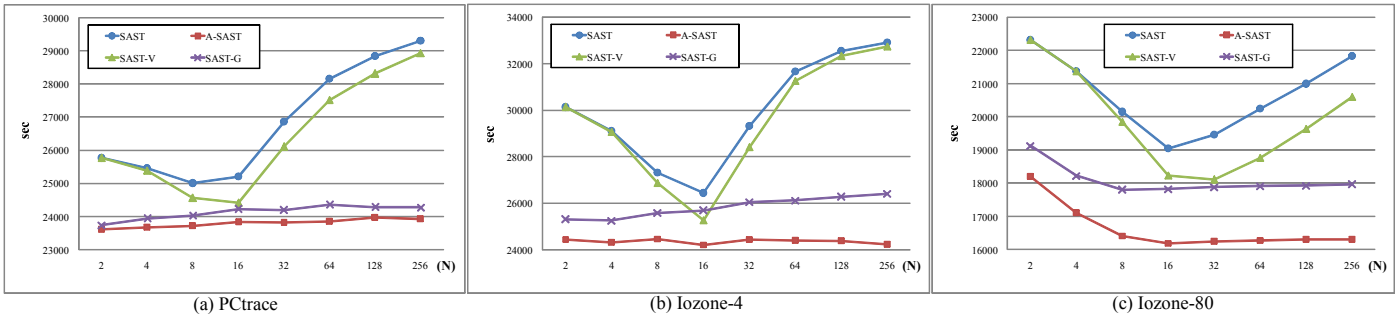
We then compared the performance improvement of A-SAST over SAST. Fig. 5 shows the total flash I/O execution times under the original SAST (SAST), SAST with block-level LRU victim selection policy (SAST-V), SAST with group merge/split technique without block-level LRU victim selection policy (SAST-G), and our proposed A-SAST (A-SAST). SAST-V and SAST-G are experimented to examine the effects of two proposed techniques independently. The X-axis shows the initial value of data block group size N. The size of log block group K is set to $N/2$ at the SAST scheme. While SAST uses the value for all data block groups and does not change it during the run time, A-SAST adapts the value depending on the characteristic of I/O workload.

Under SAST, the performance changes significantly depending on the value of N. In addition, the optimal values for N are different depending on benchmarks. While the optimal value is 8 for PCtrace, it is 16 for Iozone benchmarks.



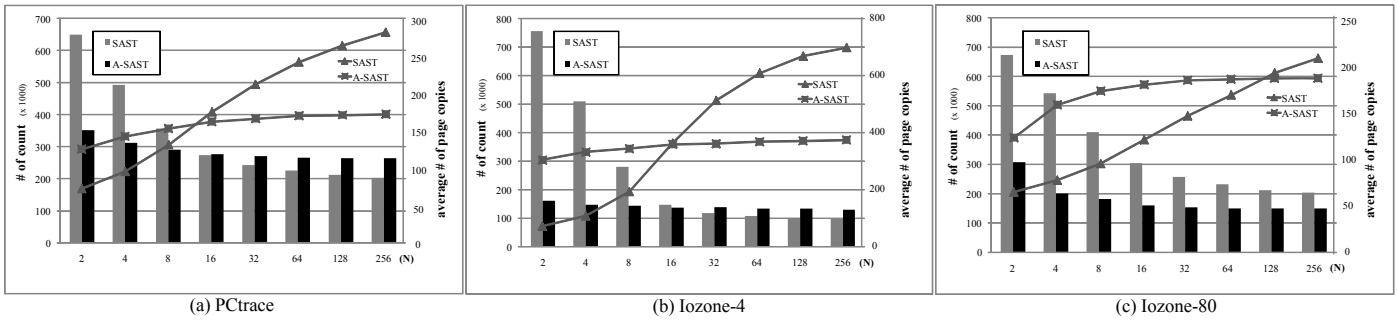
(a) PCtrace (b) Iozone-4 (c) Iozone-80

Fig. 4. Log block merge counts of logical block group in SAST. (N=4, K=2)



(a) PCtrace (b) Iozone-4 (c) Iozone-80

Fig. 5. Comparison of I/O execution time.



(a) PCtrace (b) Iozone-4 (c) Iozone-80

Fig. 6. Comparison of number of log block merges.

One important observation is that the performance of A-SAST has no significant change depending on the value of *N*. This is because A-SAST changes the data block group size to the optimal value even though the initial value is far from the optimal data block group size. Even though we use the static optimal value for *N* in SAST, A-SAST shows better performances. A-SAST efficiently improves the performances by 5%, 8%, and 15% compared to SAST under the best values of *N* for PCtrace, Iozone-4, and Iozone-80, respectively.

A-SAST shows significant improvements for the workloads which have both sequential and random requests (Iozone-80) rather than the random workloads (PCtrace and Iozone-4). This is because A-SAST can handle effectively the irregularity and the dynamic change of workload.

SAST-G improves the performance significantly especially when the initial value of *N* is too small or too large. The effect of SAST-V is significant even when the initial value of *N* is same to the static optimal value. If we use a small value for *N*, the group-level LRU and block-level LRU have similar behaviors since the log group size is small. Therefore, SAST-V and SAST show similar performances when the group size is small. Since the block-level LRU technique finds a better victim DBG for group merge and split, it elevates the performance of group size adjustments.

Fig. 6 shows the numbers of log block merges (bar graph) and the average merge costs (line graph) under SAST and A-SAST. When the initial value of *N* is small, SAST has a low merge cost due to its low log block utilization but generates a large number of log block merges. However, A-SAST reduces the number of log block merge count because the log block utilization is elevated by data block group merge operation. When the initial value of *N* is large, SAST has a high merge cost due to high log block associativity. A-SAST reduces the average log block merge cost by performing data group split.

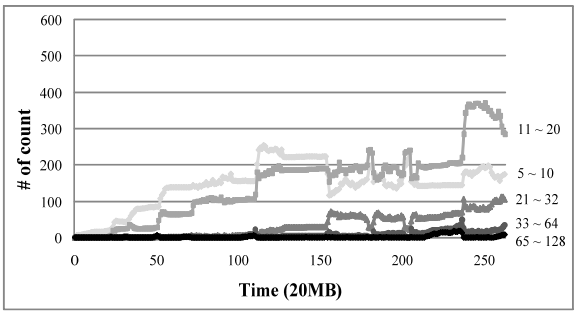
Fig. 7 shows the change of data block group size distribution at run time under the A-SAST scheme. We observed for Iozone-4 by using two different initial values of *N*. When the initial value of *N* is 4, the number of data block groups with *N* larger than 4 increases due to many data block group merges. By contrast, when the value is 64, the number of data block groups with *N* smaller than 64 increases due to many data block group splits. Therefore, two experiments that

use different initial values for *N* show similar group size distributions finally. Therefore, we can say that A-SAST can find the optimal group sizes irrespective of the initial value of *N*. A-SAST maintains diverse group sizes at a time suitable to each address space. In addition, the group size distributions are changed dynamically at run time. A-SAST repeats the group merge and split as the workload pattern varies. This means that A-SAST can adapt the workload change effectively.

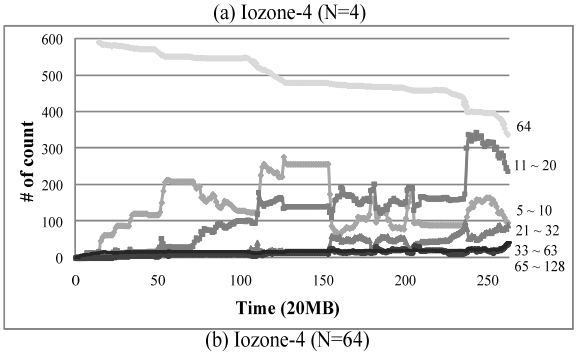
Finally, we examined the effects of parameters related to group merge/split conditions. The proper values for α, β and γ are observed to be similar at different benchmarks. Fig. 8(a) shows the performance of the PCtrace with varying the values of β and γ. The value of α is fixed to 0.4. The possible ranges of β and γ are 0~64 since it is compared to the associativity of log block. As we use a smaller value for β and a larger value for γ, the group merge hardly occurs thus there will be many small data groups. We can get the best performances when β is 4~16 and γ is 4~8. Fig. 8(b) shows the effect of parameter α when β and γ is 4 and 8, respectively. The best performance is when α is 0.2~0.4.

V. CONCLUSION

In this paper, we proposed a novel FTL scheme to proficiently deal with irregular I/O patterns in NAND flash memory. While the previous FTL schemes do not consider the variations of I/O workload depending on the execution time and the address space thus use the fixed log block associativities determined at the design time, the proposed A-SAST scheme finds the optimal value for the associativity of each address space and adjusts dynamically it during the run time. As a result, we can reduce the log block merge overhead of FTL significantly.

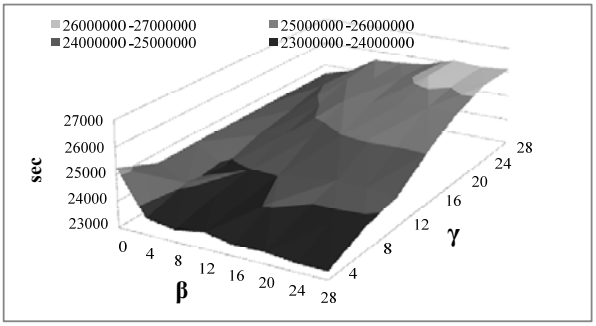


(a) Iozone-4 (N=4)

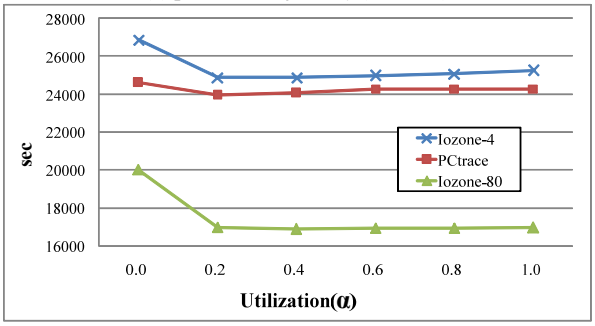


(b) Iozone-4 (N=64)

Fig. 7. The changes of group sizes.



(a) parameters β and γ (PCtrace)



(b) parameter α

Fig. 8. The effect of parameters.

REFERENCES

[1] G. Lawton, "Improved flash memory grows in popularity," IEEE Computer, vol. 39, no. 1, 2006, pp. 16-18.

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

[3] A. Ban. Flash file system optimized for page-mode flash technologies. US Patent 5,937,425, Aug. 10, 1999.

[4] A. Ban. Flash file system. US Patent 5,404,485, Apr. 4, 1995.

[5] 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, vol. 48, no. 2, 2002, pp. 366-375.

[6] S. W. Lee, D. J. Park, T. S. Chung, W. K. Choi, D. H. Lee, S. W. Park, and H. J. Song. "A log buffer based flash translation layer using fully associative sector translation," ACM Transactions on Embedded Computing Systems, vol. 6, no. 3, 2007.

[7] H. Cho, D. Shin and Y. Eom. "KAST: K-Associative Sector Translation for NAND Flash Memory in Real-Time Systems," Proc. of Design, Automation and Test in Europe (DATE'09), pp. 507-512, 2009.

[8] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, and J.-S. Kim, "A reconfigurable FTL architecture for NAND flash-based applications," ACM Transactions on Embedded Computing Systems, Vol. 7, No. 4, Article 38, 2008.

[9] J. U. Kang, H. Jo, J. S. Kim, and J. Lee. "A superblock-based flash translation layer for NAND flash memory," in Proc. International Conference on Embedded Software, 2006, pp. 161-170.

[10] Iozone Filesystem Benchmark, http://www.iozone.org

[11] Samsung Electronics, NAND Flash Memory Datasheet, http://www.samsung.com/Products/Semiconductor/NANDFlash/SLC_LargeBlock/16Gbit/K9KAG08U0M/ds_k9xxg08uxm_rev10.pdf.