Janus-FTL: Finding the Optimal Point on the Spectrum Between Page and Block Mapping Schemes

Hunki Kwon
School of Computer Science
University of Seoul
kwonhunki@uos.ac.kr

Eunsam Kim
School of Information & Computer Engineering
Hongik University
eskim@hongik.ac.kr

Jongmoo Choi
School of Computer Science & Engineering
Dankook University
choijm@dku.edu

Donghee Lee
School of Computer Science
University of Seoul
dhl_express@uos.ac.kr

Sam H. Noh
School of Information & Computer Engineering
Hongik University
samhnoh@hongik.ac.kr

ABSTRACT

NAND flash memory based storage such as SSDs is gaining popularity in commodity computer systems. Some low-end SSDs use the block mapping FTL (Flash Translation Layer) that is good for sequential write patterns but poor for random ones. On the other hand, high-end SSDs tend to use the page mapping FTL that is effective for random write patterns, but whose performance degrades after successive random writes. Designing an FTL that adapts to various workload patterns and provides long-term stable performance is a challenging issue. To resolve this issue, we propose a new FTL, which we call Janus-FTL, that provides a spectrum between the block and page mapping schemes. By adapting along the spectrum, Janus-FTL can provide long-term superior write performance for various workload patterns. We also present a cost model of Janus-FTL that shows the existence of the optimal point on the spectrum for a given workload. Our experimental results show the superiority of Janus-FTL, which adapts itself along the spectrum for a given workload, over state-of-the-art hybrid mapping FTLs and the pure page mapping FTL.

Categories and Subject Descriptors

D.4.2 [**Operating System**]: Storage Management - Secondary storage

General Terms

Design, Experimentation, Performance

Keywords

SSD, Janus-FTL, Block Mapping, Page Mapping, Cost Model

EMSOFT'10, October 24–29, 2010, Scottsdale, Arizona, USA.

1. INTRODUCTION

NAND flash memory, which is the storage medium of SSDs (Solid State Drives), has multiple blocks, and each block has multiple pages. Data can be written to a page within a block only after the block has been erased. Data once written cannot be modified without erasing the block again. Like this, flash memory has different properties from magnetic disks. To provide readable and writable sector interface of magnetic disks, flash memory storage employs a sophisticated software module called the FTL (Flash Translation Layer) that maps sectors to physical locations in flash memory storage [1, 3].

The two basic mapping techniques used in FTLs are page mapping [26] and block mapping [12]. The block mapping FTL relocates a group of consecutive sectors to a new block altogether if one or more sectors of the group are modified. Naturally, the block mapping FTL works well for sequential writes but not for random writes. On the other hand, the page mapping FTL relocates each modified sector (or multiple sectors in a page) separately to any available page in pre-erased blocks. Hence, random writes need not be randomly written any more, and this is why high-end SSDs using the page mapping FTL perform well for random writes. However, performance of the page mapping FTL degrades after successive random writes [6].

Designing an FTL that adapts to evolving workloads while providing long-term stable performance for various workload patterns has been a challenging issue. The design goal of Janus-FTL, which is the FTL of interest in this paper, is to resolve this issue. In particular, Janus-FTL provides a spectrum between the page and block mapping schemes. The cost model of Janus-FTL shows the existence of the optimal point on the spectrum for a given workload. Through various experimental results, we show the superiority of Janus-FTL that adapts to evolving workloads.

The rest of the paper is organized as follows. In the next section, we provide background information about flash memory storage and the various techniques for FTL design. Then, we explain the motivation and the basic operations of Janus-FTL in Section 3. In Section 4, we derive the cost model of Janus-FTL that shows the existence of the optimal point on the spectrum between the page and block mapping

schemes. In Sections 5 and 6, we discuss the implementation issues and present the performance evaluation results. Finally, we conclude this paper in Section 7.

2. CHARACTERISTICS OF FLASH MEMORY AND RELATED WORK

Typically, FTLs reserve some over-provisioning blocks and relocate modified sectors to clean pages of these over-provisioned blocks. In the block mapping FTL, a block contains a fixed number of sectors and a group of sectors are relocated altogether to an over-provisioned block when one or more sectors of the group are modified. In contrast, the page mapping FTL separately relocates each modified sector (or multiple sectors in a page) to an available page in any over-provisioned block [26]. Operations of the page mapping FTL are similar to those of LFS (Log-structured File System) because all modified data are appended to over-provisioned blocks and used blocks are reclaimed by *garbage collection*, which is called *segment cleaning* in LFS [21].

Though the page mapping FTL generally works well for random write patterns, it has the disadvantage of performance instability that is caused by the cost variance of garbage collection. On average, garbage collection cost is determined by the ratio of the total number of valid sectors to the size of the over-provisioned space. If we reserve more over-provisioned space in flash memory storage, the average garbage collection cost comes down. However, reserving more over-provisioned space has the effect of reducing the capacity of storage for a given set of flash memory chips. Furthermore, the garbage collection cost can fluctuate according to how the hot and cold data are intermixed within the blocks regardless of the size of the over-provisioned space.

Chen et al. showed interesting characteristics of commercial SSDs [6] and, according to their results, performance of both high-end and middle-class SSDs deteriorate significantly when they are fragmented by successive random writes. Usually, this degraded performance cannot be recovered without explicit rearrangement of data. To alleviate the performance degradation, some techniques have been proposed to distinguish and separate hot and cold data for LFS [5, 18] and for flash memory storage [7, 8]. In reality, however, distinguishing hot and cold data at the storage level involves considerable overhead.

In magnetic disks, logically adjacent sectors are physically adjacent except when they belong to different tracks. In page mapping FTLs, however, logical adjacency does not guarantee physical adjacency because the actual location of data is determined when it is written. Also, sectors are relocated to new locations whenever they are updated. Thus, file systems could be wasting much of their efforts when they try to assign adjacent sector numbers to relevant data. However, this is not a serious matter for block mapping and hybrid mapping FTLs where logically adjacent sectors have high probability to reside in the same data block except when they are temporarily placed in log blocks (a name used for over-provisioned blocks) or when they belong to different logical blocks[1].

Many hybrid mapping FTLs have been proposed to combine the advantages of page and block mapping schemes. Lee et al. proposed a hybrid mapping scheme called FAST (Fully Associative Sector Translation) that applies page mapping to log blocks, while applying block mapping to data blocks [16]. Kang et al. proposed a super-block based FTL [10], where multiple blocks form a super-block and multiple log-blocks can be assigned to a super-block. Also, the super-block based FTL applies the page mapping scheme to pages inside the super-block and distinguishes hot and cold data to lower the garbage collection cost. Park el al. proposed the Reconfigurable FTL that assigns K log blocks to N data blocks and present experimental results for determining K and N for various workloads [20]. Liu et al. proposed an adaptive block-set based management scheme where a hot data block has an exclusive replacement block (another name used for over-provisioned blocks), while many cold blocks share one replacement block [17].

There have been other studies on SSD design and characterization. Agrawal et al. gave a taxonomy of the many design choices for SSDs [4]. Also, Shin et al. analyzed tradeoffs of various configurations such as striping and allocation methods [23]. Chen et al. analyzed the performance characteristics of state-of-the-art SSDs through various experiments [6]. Kim et al. proposed methods to extract performance parameters of SSDs such as the read/write/erase unit sizes, the type of flash memory chips, and the buffer capacity for read/write [13]. Finally, Seo et al. investigated the power issue in SSDs [22].

3. JANUS-FTL

3.1 Motivation of Janus-FTL

To better understand the motivation behind Janus-FTL, in the following, we first explain the operations of a virtual FTL, which we will refer to as the vFTL. For data blocks, vFTL uses block mapping, keeping all sectors in sequence within data blocks, while for log blocks, that is, the over-provisioned blocks, page mapping is used. (The merge operation of vFTL, depicted in Figure 1, is almost identical to that of the FAST scheme [16] and is similar to that of many hybrid mapping FTLs. We believe that the vFTL example that we use is a close example of typical contemporary FTLs.)

In Figure 1, three log blocks comprise the *Page Mapping Area (PMA)* and all other data blocks comprise the *Block Mapping Area (BMA)*. In the BMA of Figure 1, the three large rectangles represent the data blocks, where a group of consecutive sectors are stored. Each large rectangle has five small rectangles that represent pages in the block. Each block in the BMA is designated by a block number, which refers to the logical block number. In a real implementation, a *block map* would translate the logical block number to the physical block number; however, for our explanation, the physical block number is not important. Each page in the block is designated by a white number, which refers to the sector number stored in the page. In the PMA, block numbers are primed, and these will be used to refer to specific blocks during our explanation. A *page map* translates the sector number to the physical location in the PMA. In the PMA, at least one log block is erased and designated as the *active log block*, where updates will be written to.

Let us assume that, in Figure 1(a), log block 0' had been the active log block and all updated sectors 0, 1, and 7 had been written to the pages in it. Then, log block 1' was erased and designated as the new active log block. Thereafter, sectors 1, 2, 3, 9, and 10 were written to the pages in log block

[1]See Section 3 for logical blocks

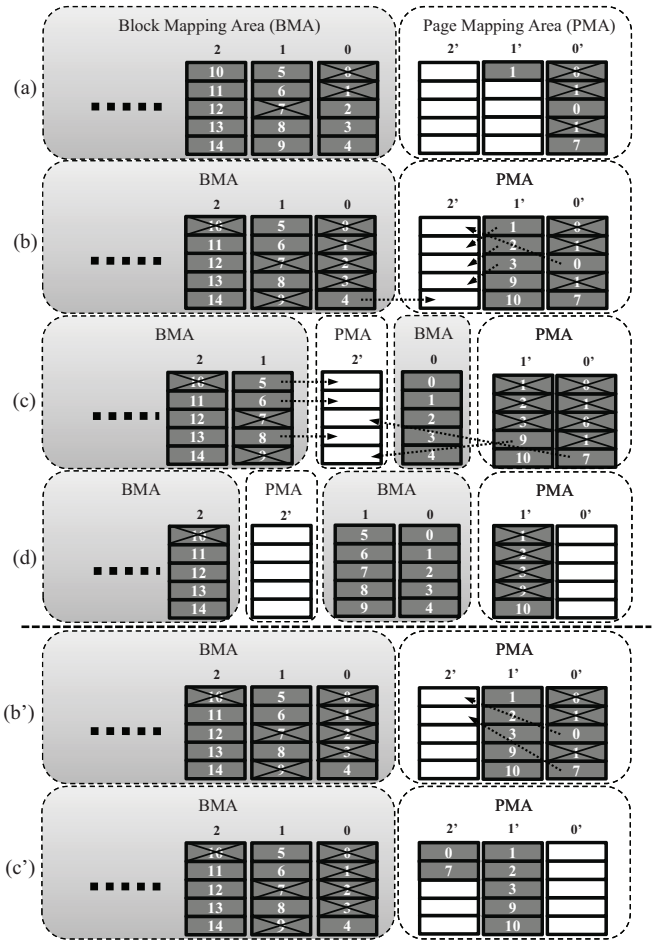


Figure 1: Procedures of typical merge and garbage collection

1' resulting in Figure 1(b). In all the figures, all invalid pages are marked by X. Now there is only one empty log block 2' and vFTL must maintain at least one empty log block for the merge operation. Let us assume that vFTL chooses log block 0' for the merge operation. As this block has two valid sectors 0 and 7 that belong to two different logical blocks 0 and 1, respectively, vFTL merges two logical blocks to make one more empty log block. To merge logical block 0, it erases the last empty log block 2' and copies all the valid sectors 0∼4 from the log blocks and logical block 0 to the empty log block 2' (depicted by the arrows in Figure 1(b)). Then, the empty log block 2' becomes logical block 0 and the previous logical block 0 becomes the new empty log block 2' resulting in Figure 1(c). Now vFTL merges logical block 1 following the same steps (the arrows in Figure 1(c)), resulting in two empty log blocks 0' and 2' as shown in Figure 1(d). For further write requests, one of the two empty blocks would now be designated as the new active log block.

The thing to note here is that vFTL could have provided clean pages more efficiently by performing garbage collection rather than merging multiple blocks. For example, as depicted in Figure 1(b'), assume that vFTL performs a garbage collection instead of a merge, erasing log block 2' and copying valid sectors 0 and 7 from log block 0' to log block 2'. Then, we have three clean pages in log block 2' and an empty log block 0' as shown in Figure 1(c'). Now, incoming write requests may be served with clean pages in log block 2'.

In this manner, a cost based selection between merge and garbage collection makes it possible for vFTL to provide clean pages more efficiently. If utilization of log blocks is low, then consequently, garbage collection cost will be low, and garbage collection will be better than merge; otherwise merge will be better. From a broad perspective, the merge operation evicts some sectors from log blocks so as to lower the utilization of log blocks and, consequently, garbage collection cost thereafter. Therefore, it could be better to perform merge first, then garbage collection rather than to perform garbage collection alone.

Now the question becomes: Can we find an optimal sequence of merges and garbage collections? This question can be converted to finding the optimal utilization of log blocks that minimizes overall page write cost for a given workload. If utilization is higher than the optimal value, the FTL performs a merge to lower the utilization. Otherwise, the FTL performs garbage collection. We will pose and answer this question in Section 4, where we show the existence of the optimal utilization of PMA for a given workload.

Let us now discuss an important difference between Janus-FTL and other hybrid mapping FTLs. In Figure 1(b), logical blocks 0∼2 have some invalid sectors, and space occupied by these sectors are not utilized for any purpose. Specifically, logical block 0 in Figure 1(b) has one valid page and four invalid pages. In many cases, data blocks (logical blocks in our example) may have only a small number of valid pages, sometimes, in fact, no valid pages at all, when all their sectors have been updated in log blocks. In this way, a significant portion of the data blocks is wasted, and the size of wasted space, in the worst case, may be up to the size of the initial over-provisioned space. For example, if an FTL reserved 3% of the total space as over-provisioned blocks, then, in the worst case, roughly another 3% of the space in data blocks may be wasted due to invalid sectors. At this point, we may pose another question: Can we utilize the wasted space occupied by invalid sectors in data blocks? To the best of our knowledge, there is no block mapping or hybrid mapping FTLs that try to utilize this wasted space. Later on, we will show how Janus-FTL utilizes this wasted space.

We posed two questions, and the answers to these two questions are correlated. As we show next, Janus-FTL eliminates the wasted space by dynamically changing the sizes of the page mapping and block mapping areas. Furthermore, the answer to the first question, that is, finding the optimal utilization of the page mapping area, provides a solution for finding the optimal sizes of both areas.

3.2 Design of Janus-FTL

In this section, we will explain how Janus-FTL recycles the wasted space in data blocks. In the explanation, we will focus on the two basic operations of Janus-FTL, that is, *Fusion* and *Defusion*, and omit descriptions of already known implementation techniques for page and block mapping FTLs (See [4, 9, 12] for details). For brevity, we will skip discussions on additional features such as bad block

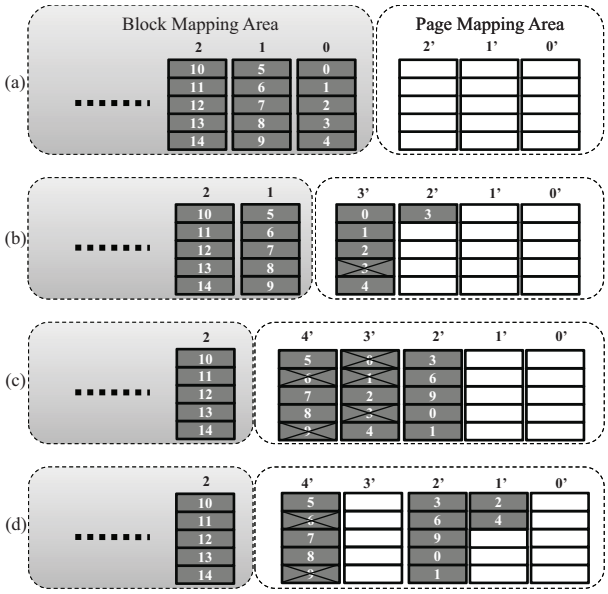


Figure 2: Fusion and garbage collection

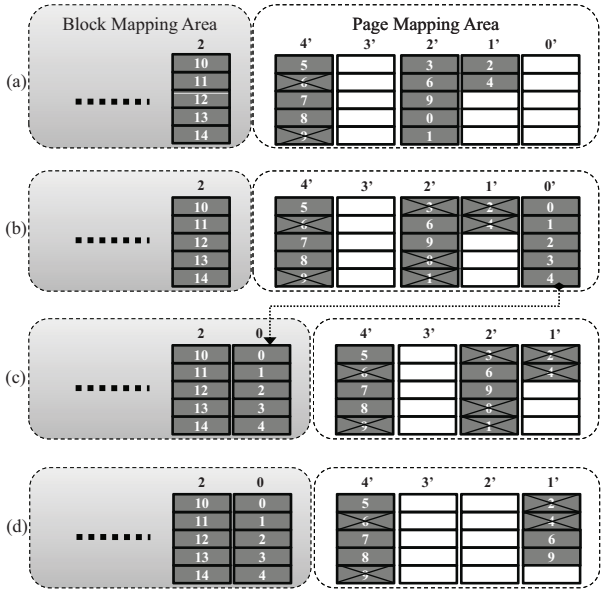


Figure 3: Defusion

management and wear-leveling. We assume that a page keeps only one sector, where in reality multiple sectors are kept. However, the cost model that we will introduce later is not affected by this assumption because it uses the "number of pages in a block" rather than the "number of sectors in a block". Also, in our previous discussion, we mentioned utilization; formally, we define utilization u as $u = d/s$, where d and s refers to the size of data and total available space, respectively.

Figure 2(a) shows an example of the initial state of flash memory storage, where three over-provisioned blocks are reserved to comprise the Page Mapping Area (PMA), to which the page mapping scheme is applied. Also, all other blocks comprise the Block Mapping Area (BMA), where each block stores a group of consecutive sectors and the block mapping scheme is applied. Like the vFTL, a block map translates the logical block number to the physical block number in the BMA and a page map translates the logical sector number to the physical location in the PMA. Also, the page map maintains a list of logical block numbers in the PMA.

As we mentioned earlier, the average garbage collection cost depends on utilization, and this may be unreasonably high if the initial over-provisioned space is too small for the total data size. However, we may keep the garbage collection cost low by setting a limit to the number of data blocks that share the over-provisioned blocks at the same time. To control the utilization of the PMA, we introduce two operations that move data blocks between the BMA and PMA, namely, Fusion and Defusion. Moreover, wasted space can be utilized by fusing data blocks with invalid page(s) into the over-provisioned blocks in the PMA. In Figure 2(b), block 2' was erased and is designated as the active log block. Assume that a write request to sector 3 in logical block 0 has just arrived. Then, the modified sector 3 is written to the first page of log block 2' as shown in Figure 2(b). Now, logical block 0 has an invalid page and Janus-FTL migrates logical block 0 to the PMA (Figure 2(b)), and we call this the *Fusion operation*. The Fusion operation updates the page map that keeps a list of logical blocks and also the locations of all sectors in the PMA. It should be noted that logical block 0 does not exist any more in the BMA and the physical block occupied by logical block 0 is now log block 3'.

Now, let us assume that a write request to sector 6 in logical block 1 arrived and logical block 1 is moved to the PMA. Then, modified sectors 6, 9, 0, and 1 are written to active log block 2', the result being as shown in Figure 2(c). Now, there is no available page in active log block 2' and one of the two empty blocks must now become the new active log block. However, as we shall see later, Janus-FTL must maintain at least two empty blocks all the time, and using one of the two empty blocks violates this rule. Hence, Janus-FTL executes a garbage collection to make a new active log block. For the first step of garbage collection, Janus-FTL (randomly) erases, say, empty log block 1'. In the next step, it chooses a block with the smallest number of valid sectors, in this case block 3', and copies all valid sectors to block 1' as depicted in Figure 2(d). After these copies, block 1' is designated as the active log block and, from then on, all modified sectors will be written to the available pages of this new active log block. In Figure 2(d), it should be noted that there are two empty blocks, block 0' and block 3', after garbage collection.

Again, we need to mention an important difference between Janus-FTL and other FTLs that do not recycle invalid space in data blocks. In particular, in the FAST scheme, data blocks with invalid pages never join the PMA when updated sectors are written to pages in the PMA. If 10 blocks were reserved as over-provisioned space, then utilization of the over-provisioned space is 0.8 (= 8/10) in the FAST scheme when all sectors of the 8 data blocks are updated in the over-provisioned blocks. In contrast, utilization of the over-provisioned space in Janus-FTL is 0.44 (= $8/(10+8)$) for the same case because data blocks with invalid pages

lend their space to the over-provisioned space. Even when sectors of 40 data blocks are modified, the utilization of the over-provisioned space becomes 0.8 (= 40/(10 + 40)). Like this, recycling wasted space in data blocks can lower utilization of over-provisioned space, consequently leading to lower garbage collection cost. This is one reason behind the superior write performance of Janus-FTL.

Now we will describe the *Defusion operation* of Janus-FTL that is similar to the merge operation in Figure 1. Assume that Janus-FTL decides to evict logical block 0 from the PMA. We start from the situation depicted in Figure 3(a). Note that sectors 0∼4 of logical block 0 are dispersed in blocks 1' and 2'. For the first step, an empty log block , say block 0', is (randomly) chosen to be erased. Then, it copies all sectors of logical block 0, sectors 0∼4, to the empty log block 0' as depicted in Figure 3(b). After these copies, logical block 0 is removed from the PMA via a map update resulting in Figure 3(c). We now see that logical block 0 appears in the BMA. As a result of these steps, we observe that the PMA has only one empty log block, and this is against the aforementioned rule that there must be at least two empty blocks in the PMA. Now, Janus-FTL has to make another empty log block. In this example, as shown in Figure 3(d), we can make the second empty log block by just copying sectors of block 2' to the active log block 1'. However, in some cases, this may require an additional garbage collection using the last empty log block 3', and this is why Janus-FTL must maintain at least two empty blocks at all times. (If Janus-FTL executes garbage collection before the Defusion operation, then it must maintain at least one empty block at all times.)

With the Fusion and Defusion operations, Janus-FTL can dynamically change the sizes of PMA and BMA. If only a few logical blocks temporarily join the PMA to update their sectors and are merged out quickly, the behavior of Janus-FTL is that of the block mapping FTL. As more logical blocks join the PMA, the behavior of Janus-FTL gradually changes to that of page mapping FTL. When all logical blocks join the PMA, Janus-FTL becomes the pure page mapping FTL. Therefore, Janus-FTL can be regarded as a spectrum between two extremes, that is, the block mapping FTL and the page mapping FTL. Let us now find the optimal point on the spectrum for a given workload.

4. THE COST MODEL

In this section, we will answer the question of finding the optimal utilization of the PMA that will indicate the optimal point on the spectrum between the page mapping and block mapping schemes. It is important to find the optimal value because maintaining utilization at the optimal value can lead to minimal write cost for a given workload. To figure out the optimal utilization, first, we will derive the cost models for Defusion and garbage collection of the PMA. Then, we will give a discussion on the existence of the optimal utilization for a given workload in Section 4.2. In our cost models, we do not consider the map update cost because it is comparatively much smaller than the costs of other operations in Janus-FTL. Moreover, with various optimization techniques, for example, such as using non-volatile RAMs, the map can be updated with almost negligible cost.

4.1 Garbage Collection and Defusion

When the initial over-provisioned blocks are given, utilization of the PMA increases as logical blocks start joining it. Garbage collection cost increases along with utilization. Due to the similarity between the page mapping scheme and LFS, garbage collection cost of the PMA can be derived from the LFS cost model.

Let us assume that N_P is the number of pages in a block, C_{PROG} is the page write cost (time), C_{CP} is the page copy cost (time), and C_E is the block erase cost (time). Utilization of the PMA is u_d. We assume that, for garbage collection, Janus-FTL selects a block that has the least valid pages. In general, utilization of the selected block, u, is typically smaller than the utilization of the PMA, u_d. (We will present the relation between u_d and u later.) Then, the average garbage collection cost of the PMA, C_{GC}, depends on utilization, u, and can be defined as follows:

$$C_{GC} = u \cdot N_P \cdot C_{CP} + C_E \tag{1}$$

From the utilization of the selected block, u, we can expect $u \times N_P$ valid pages in the selected block. Then, garbage collection erases an empty block with cost C_E and copies valid pages to the empty block with cost $u \times N_P \times C_{CP}$. After garbage collection, the previously empty block becomes the new active log block, where modified sectors will be written to. Right after garbage collection, we can expect $(1-u) \times N_P$ available pages in the new active log block and, thus, $(1-u) \times N_P$ write requests can be served with each garbage collection. Counting that $(1-u) \times N_P$ write requests share the garbage collection cost, we can derive the average page write cost of the PMA as follows:

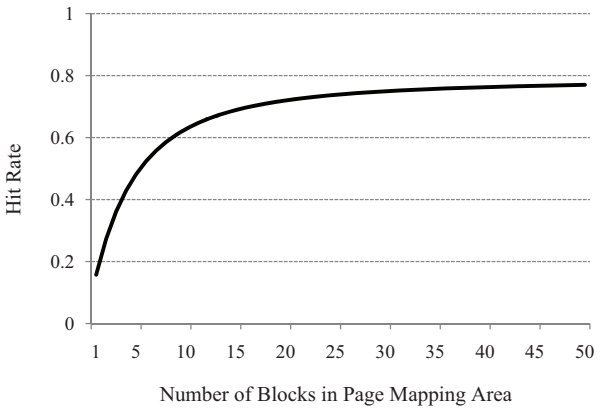
$$C_{PW} = \frac{C_{GC}}{(1-u) \cdot N_P} + C_{PROG} \tag{2}$$

Before deriving the Defusion cost, we should note that the Fusion cost can be ignored because the Fusion operation only requires an update of the map that keeps a list of the fused logical blocks. Let us now focus on the Defusion cost. From Figure 3(b), we saw that Defusion erases an empty block with cost C_E and copies all sectors of a logical block to the empty block with cost $N_P \times C_{CP}$. In steady state, the Defusion operation consumes an empty log block and thus, it has to make another empty log block. To make an empty block, it selects a block with the least valid pages in the PMA. With utilization, u, we can expect $u \times N_P$ valid pages in the block. Also, copying those pages to the active log block may cause garbage collections when there are no available pages in the active log block. As a result, we must consider a possible garbage collection cost during page copies. Interestingly, C_{PW} in Equation (2) defines the page write cost considering a possible garbage collection and thus, page copy cost after Defusion can be calculated by multiplying $u \times N_P$ to C_{PW}. Hence, the Defusion cost can be derived as follows:

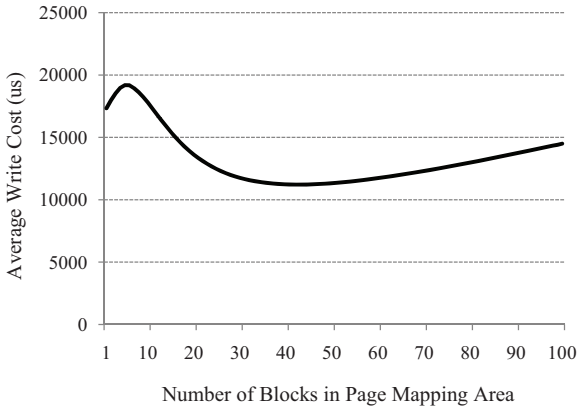
$$C_{Defusion} = N_P \cdot C_{CP} + C_E + u \cdot N_P \cdot C_{PW} \tag{3}$$

4.2 Existence of PMA Optimal Utilization

Using the above equations, we can derive the overall write cost model of Janus-FTL that shows the existence of the optimal utilization for a given workload. From many studies on cache replacement policies, we were able to observe that performance of a storage device depends on the characteristics of workloads, specifically the locality of reference pattern. To derive the overall write cost model, we need to



(a) Hit rate vs. number of fused blocks



(b) Average write cost vs. number of fused blocks

Figure 4: Average write cost and existence of optimal utilization of Page Mapping Area in Janus-FTL
We assume that the number of blocks and initial over-provisioned blocks is 1024 and 20, respectively.
Also, we set N_P=64, C_E=2ms, C_{CP}=225μs, and C_{PROG}=200μs.

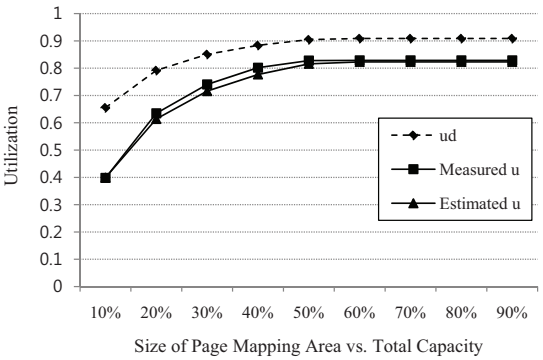


Figure 5: The measured and estimated values of u and its relation with u_d for the OLTP Financial Trace

characterize a workload and, for that purpose, we will use the hit rate curve of a given workload. Hereafter, we will use the term *miss* when a write is requested to a sector in the BMA and *hit* when a write is requested to a sector in the PMA. Also, we assume that Janus-FTL uses the LRU (Least Recently Used) policy to select a block to be evicted from the PMA.

Let us assume that r is the hit rate of the PMA at a certain time. As blocks with recently updated sectors reside in the PMA, the PMA can be regarded as a cache with hit rate r. Now, with probability r, a sector write can be done with cost C_{PW}. On the other hand, with probability of $(1-r)$, a sector write pays a cost of "$C_{Defusion} + C_{Fusion} + C_{PW}$" because it incurs a Defusion to evict the LRU block from the PMA, a Fusion to migrate the newly referenced block to the PMA, and a write of the requested sector to the PMA. Since Fusion cost is negligible, the overall sector write cost of Janus-FTL can be derived as follows:

$$C_{Avgw} = (1-r) \cdot C_{Defusion} + C_{PW} \quad (4)$$

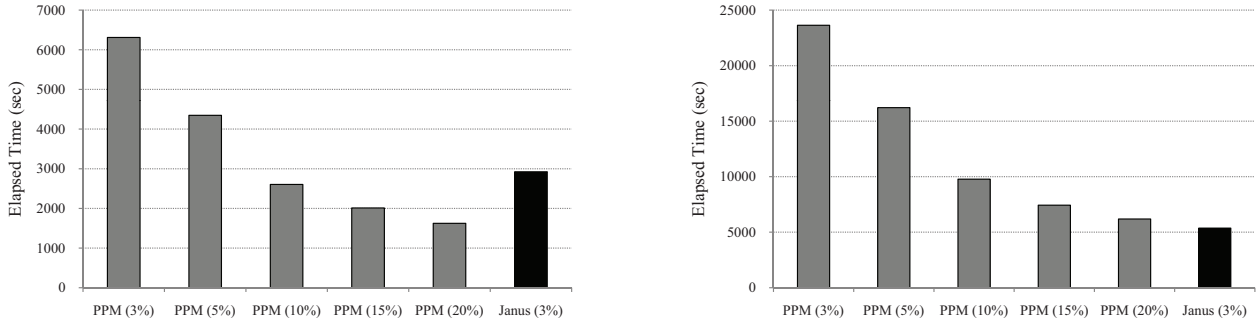
Now we can use this equation to find the optimal utilization of the PMA and, in turn, the optimal point on the spectrum for a given workload. Let us take two examples for finding the optimal value; one for the workload with strong locality and one with a random write pattern. Assume that the workload has strong locality and the hit rate varies according to the number of fused blocks in the PMA as shown in Figure 4(a). In particular, the hit rate soars as recently referenced blocks join the PMA. However, as more blocks join, the increase in hit rate drops quickly. Now we can redraw Figure 4(a) with u_d in the x-axis because we can calculate u_d with the number of initial over-provisioned blocks (N_O) and the number of fused blocks (N_F) such that $u_d = \frac{N_F}{N_O+N_F}$. However, u, the utilization of the selected block for garbage collection, is needed to calculate the average write cost with Equation (4).

To find u, we take from the following interesting relation between u_d and u that has been proposed with some justifications by previous LFS studies [19, 24]:

$$u_d = \frac{(u-1)}{\ln u} \quad (5)$$

To confirm that Equation (5), which was proposed for the LFS, is still valid for flash memory storage, we measured u, the average utilization of the victim block for garbage collection with the OLTP Financial Trace [2] in our simulation environment that will be described later, and compared it with the estimated value obtained from Equation (5) as shown Figure 5. In this figure, we can observe that, at least for database workloads used in our experiments, the estimated u from Equation (5) matches well with the measured u even in flash memory storage and, hence, we use this equation in our examples and experiments. (The Postmark benchmark also shows similar results. However, in the case of the PC Trace [15], the estimated u differs somewhat from the measured u. We believe this is because the length of the trace is quite short and that most requests are sequential.)

With u derived from Equation (5), we can obtain the average write cost graph as seen in Figure 4(b). In the figure, the average write cost increases at first but then plunges as



(a) PC Trace

(b) OLTP Financial Trace

Figure 6: Comparison of Pure Page Mapping FTL and Janus-FTL
The percent in the parenthesis refers to the ratio of over-provisioned blocks to the total number of blocks in flash memory.

recently referenced blocks join the PMA and the hit rate of the PMA increases sharply, culminating at the minimal average write cost point. Beyond the minimal point, it again increases because garbage collection cost rises, while the hit rate increase slows down. These two contradicting factors, that is, the gain from hit rate increase and the loss from the increase in garbage collection cost, result in the optimal utilization of the PMA that minimizes the overall average write cost of Janus-FTL. This optimal utilization indicates the optimal point on the spectrum between the block mapping and page mapping schemes.

Let us now consider the other random write pattern example. In this case, the hit rate increases monotonically as more logical blocks join the PMA. Then, the average write cost becomes minimal at the page mapping extreme of the spectrum. (We do not present a graph of this result in this paper.) Generally, the optimal point moves toward the page mapping extreme if the workload becomes more random and moves toward the block mapping extreme as the locality gets stronger. Like this, there exists an optimal point for each workload on the spectrum and Janus-FTL can dynamically move along the spectrum between the block and page mapping schemes.

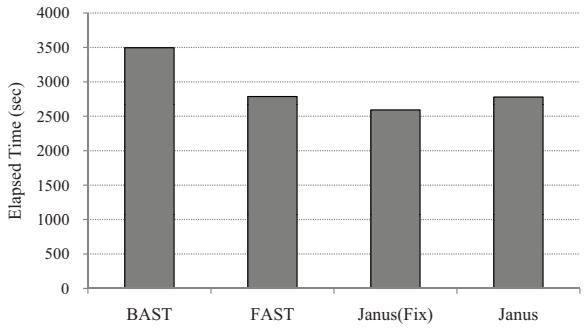
5. IMPLEMENTATION

Our implementation of Janus-FTL maintains a pool of more than two empty blocks that can be used for garbage collection, Defusion, and sequential buffering. In the previous sections, we assumed that a block in the BMA always moves to the PMA to modify its sectors. However, our implementation of Janus-FTL uses some optimization techniques for the sequential write pattern. In more detail, if a write is requested to a sector in the BMA, Janus-FTL allocates a *sequential log block* for the logical block where the requested sector resides. Then all write-requested sectors in the logical block will be written to the sequential log block. When Janus-FTL needs to reclaim an allocated sequential log block, it evaluates the write pattern of the sequential log block. According to the evaluation result, Janus-FTL selectively applies a switch merge [12], a partial merge, or a migration [14] to reclaim the sequential log block and the logical data block. In random write cases, it fuses the logical data block and the sequential log block to the PMA. Then, all unused pages in the sequential log block are used for incoming write requests in the PMA. With these optimizations, sequential write requests are processed efficiently in sequential log blocks and only seemingly random write requests enter the PMA. It should be noted that those optimizations for sequential writes do not affect the Janus-FTL cost model if we filter out the sequential references processed by the optimization techniques. To keep utilization of the PMA at a certain level determined by an adaptation scheme, some logical blocks may be evicted from the PMA before fusing new logical blocks into the PMA.

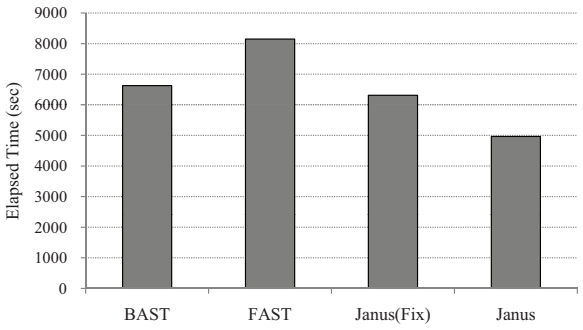
In order to adapt to the workload, a deliberate adaptation scheme is required to determine the optimal utilization of the PMA. Our implementation of Janus-FTL uses ghost buffers to estimate the hit rate curve by simultaneously simulating various cache sizes (that is, various number of blocks in the PMA). Based on the estimated hit rates, Janus-FTL periodically (after every N_D write requests in the current implementation, where N_D is the number of initial data blocks) makes a decision to inflate or deflate the PMA and gradually increases/decreases the size of the PMA.

We implemented two versions of Janus-FTL, one for our simulation environment and the other for the Linux MTD (Memory Technology Device) subsystem. Also, we implemented a pure page mapping FTL that uses a greedy policy for garbage collection and modified existing code of the FAST scheme [16] to port it to our environments. (We implemented BS-FTL [17], Reconfigurable-FTL [20], A-FTL [25], and SB-FTL [10] in the simulation environment. However, we do not present their performance results as they are more-or-less similar to the FAST scheme on various workloads.)

All FTLs in the simulation environment, including Janus-FTL, assume that maps are kept in non-volatile RAM such that we can focus only on the performance of basic FTL algorithms. In contrast, all FTLs running in the Linux MTD subsystem write maps to flash memory for consistency recover upon sudden power-failures since they are without non-volatile RAM. Our implementation of map management is similar to the i-node map management of LFS, which uses two techniques, namely, check-point and roll-forward, and has little overhead for map writes while providing power-failure recovery.



(a) PC Trace



(b) OLTP Financial Trace

Figure 7: Comparison of BAST, FAST, Janus-FTL with fixed utilization, and adaptive Janus-FTL

6. EXPERIMENTS

In the first and second sets of experiments, we used our simulation environment that mimics flash memory operations and measured their execution times. (Number of blocks is set to 8192 and other parameters of the simulator are given in Figure 4.) For the experiments, we used two traces, namely, the PC Trace [15] and OLTP Financial Trace [2]. Specifically, the PC Trace was collected from various applications such as a word processor, music player, web browser, and game running on Windows XP. The OLTP Financial Trace is a write-intensive trace collected from banking facilities. Comparatively speaking, the PC Trace has mostly sequential writes while the OLTP Financial Trace has random and locality-based writes.

In the first set of experiments, we compare Janus-FTL with pure page mapping FTL. The experimental results also show the trade-off between performance and the amount of initial over-provisioned space. In general, performance of the pure page mapping (PPM) FTL is known to increase with more over-provisioned blocks, and we confirm this in Figure 6. For the experiments, we put aside 3%, 5%, 10%, 15%, and 20% of the total blocks in flash memory as over-provisioned blocks. Though performance of the PPM-FTL depends on how hot and cold data are mixed during the aging process, we just placed all sectors sequentially in the PPM-FTL before the experiments. This is because the aging process can be controversial and also, we wanted to give favor to the PPM-FTL in our experiments. However, it should be noted that Janus-FTL shows almost consistent performance regardless of storage aging.

In Figure 6, we observe that for PPM-FTL execution time decreases with more over-provisioned blocks. Janus-FTL with 3% over-provisioned space performs much better (about four times under the OLTP Financial Trace) than PPM-FTL with the same over-provisioned space. PPM-FTL outperforms Janus-FTL only when over-provisioned space is larger than or equal to 10% of the total space in the experiments with the PC Trace and fails to outperform Janus-FTL in the experiments with the OLTP Financial Trace. Overall, Janus-FTL outperforms PPM-FTL with even much smaller over-provisioned space.

One disadvantage of the PPM scheme is that it cannot exploit the various optimization techniques for sequential write patterns. As recycling blocks with the switch merge is the best way to write a given amount of data to flash memory storage and software has been optimized to generate sequential write patterns, optimizations for sequential write have much potential to improve performance of storage devices. Unfortunately, those optimizations are feasible only when sectors are stored consecutively in a block, which is the case of the BMA in Janus-FTL. If the storage device is aged, it is difficult for PPM-FTL to achieve the efficiency of the switch merge, and even of the partial merge, during garbage collection. In contrast, as sectors are placed sequentially in the BMA, Janus-FTL can apply various optimization techniques for sequential writes. Also, Janus-FTL controls the utilization of the PMA to keep the garbage collection cost low and this is another reason for its superior write performance.

The second set of experiments compares Janus-FTL with block mapping and hybrid mapping FTLs. Figure 7 shows the experimental results of the four FTLs, namely BAST, FAST, Janus-FTL with fixed utilization (denoted as "Janus(Fix)"), and Janus-FTL with adaptive scheme (denoted as "Janus"). In particular, Janus(Fix) tries to maintain the utilization of the PMA to 75%.

In Figure 7(a), BAST and Janus(Fix) perform the worst and the best, respectively. Interestingly, the adaptive Janus-FTL performs worse than the Janus(Fix) because PC Trace is too short and the size of the working set (unique sectors referenced within a time span) changes too quickly. However, the opposite is true for the OLTP Financial Trace, which is long enough for adaptation, as seen in Figure 7(b). Specifically, the adaptive Janus-FTL performs 20% better than Janus(Fix). In the experiments with the OLTP Financial Trace, FAST performs worse than BAST, and the reason is that FAST uses only one sequential log block while BAST exploits several log blocks for sequential references. As a result, we observe more switch or partial merge cases in BAST than FAST.

In the last set of experiments, we compare the performance of Janus-FTL and other FTLs with the Postmark benchmark [11] running on Linux. As we mentioned previously, all FTLs in these experiments are embedded in the Linux MTD subsystem. They write modified maps to flash memory storage for consistency recovery upon sudden power-failure. In the experiments, we reserved 10% of the total flash memory capacity (1GB, 8192 blocks of 128KB) as over-provisioned space in all FTLs.

The Postmark benchmark makes random read and write requests to data files. If the total size of data files is small,

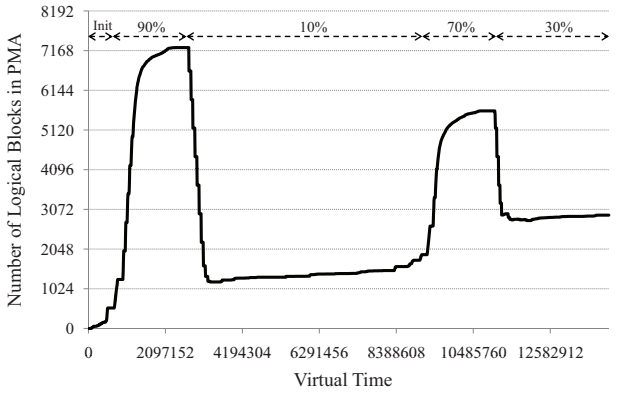


Figure 8: Workload adaptation of Janus-FTL during Postmark benchmark execution

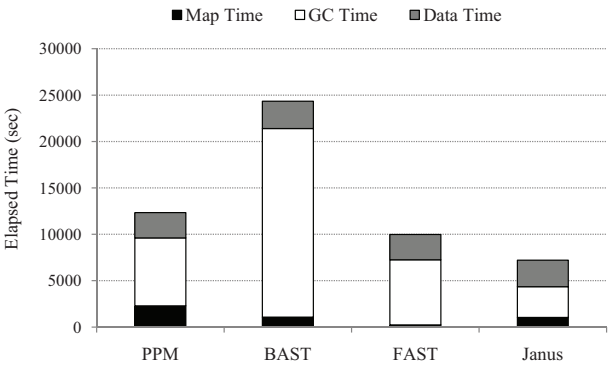


Figure 9: Comparison of FTLs with Postmark benchmark

then the optimal point may exist somewhere in the middle of the page mapping and block mapping extremes. The optimal point moves toward the page mapping extreme as the total size becomes larger. If the total size is close to the storage capacity, then the Postmark benchmark randomly accesses almost the entire storage device area and, in this case, the optimal point will be around the page mapping extreme. To investigate the benefit of the workload adaptation feature of Janus-FTL, we ran the Postmark benchmark four times with each iteration of the benchmark having a different total data file size. Specifically, we set the total data file size to 90% and 10% of the storage capacity for the first and the second iterations, respectively. Then, we set it to 70% and 30% for the third and the fourth iterations, respectively.

In Figure 8, we can see that Janus-FTL adapts well to the workload changes during the Postmark benchmark. In particular, about 7,200 data blocks were fused to the PMA at the first iteration, and the number of fused blocks changed to roughly 1,300, 5,500, and 2,800 at the second, the third, and the fourth iterations, respectively. From these experiments, we can conjecture that the performance difference between PPM-FTL and Janus-FTL is mainly due to the capability of Janus-FTL to adapt to the evolving workload.

In the experimental results of Figure 9, "Map Time" accounts for the elapsed time for map read/write, " GC Time" accounts for block recycling time such as garbage collection, merge, Fusion, and Defusion, while "Data Time" refers to the elapsed time for reading and writing requested sectors. Note that the "Data Time" is exactly the same for all FTLs and the differences among FTLs are only caused by block recycling and map read/write.

As expected, PPM-FTL has the largest map read/write overhead because its map size is the largest. The map read/write overhead of BAST, which has the smallest map, seems to be larger than we expected at first glance. However, the large map overhead of BAST is mainly due to its inferior performance and frequent occurrence of block recycling operations that incur map writes. In the experiments, FAST shows the lowest map read/write overhead because it performs moderately well and its map size is slightly larger than that of BAST. (FAST has a small page map for log blocks in addition to a block map for data blocks.) The current implementation of Janus-FTL uses a page map for both the BMA and the PMA and, thus, it has the same map overhead as PPM-FTL. In Figure 9, however, Janus-FTL seems to incur less overhead than the PPM-FTL due to its superior performance and infrequent occurrence of block recycling operations. We should note that the "Map Time" of Janus-FTL may be reduced with a sophisticated implementation that uses a block map for the BMA.

Figure 9 shows that Janus-FTL, which adapts to the evolving workload, performs the best among the four FTLs. Specifically, it performs more than two times better than PPM-FTL if we exclude the "Data Time". As expected, BAST performs the worst due to the dominant random writes of the Postmark benchmark. In the experiments, FAST performs well but still worse than Janus-FTL. FAST is similar to the non-adaptive Janus-FTL in that both FTLs apply page mapping to a limited area, though FAST does not control the utilization of the log blocks and also does not recycle wasted space in data blocks, and these differences result in the performance differences. In particular, Janus-FTL performs 1.6 times better than FAST if we exclude the "Data Time". Furthermore, extra overhead ("Map Time" and "GC Time") for writing sector data in Janus-FTL can be reduced to less than half that of FAST in a system using non-volatile RAM to update a map with negligible "Map Time".

7. CONCLUSION

In this paper we proposed the Janus-FTL that provides a spectrum between page and block mapping schemes. With adaptation for workloads on the spectrum, Janus-FTL can provide superior long-term performance for both sequential and random writes. Also, by recycling wasted space and controlling the utilization of the page mapping area, it can maintain garbage collection overhead low even with a small amount of over-provisioned space.

Janus-FTL can be enhanced even further. To estimate u from u_d, our current implementation uses only one equation regardless of the workload. Deriving and applying different equations for various workloads will be our next research topic. We expect that distinguishing hot and cold data will further lower the garbage collection overhead of Janus-FTL. Also, the map writing overhead of Janus-FTL can be reduced further through implementation optimizations. Fi-

nally, developing a prudent replacement policy that determines the blocks to be evicted from the PMA may be a promising research topic.

8. ACKNOWLEDGMENTS

This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. R0A-2007-000-20071-0) and by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MEST) (No. 2009-0085883).

9. REFERENCES

[1] *Flash-memory Translation Layer for NAND Flash (NFTL)*. M-Systems, 1998.

[2] *Storage Performance Council I/O Traces*. http://traces.cs.umass.edu/storage/Financial1.spc.bz2.

[3] *Understanding the Flash Translation Layer (FTL) Specification*. Intel Corporation, 1998.

[4] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX 2008 Annual Technical Conference*, pages 57–70, 2008.

[5] T. Blackwell, J. Harris, and M. I. Seltzer. Heuristic Cleaning Algorithms in Log-Structured File Systems. In *Proceedings of the USENIX 1995 Annual Technical Conference*, pages 277–288, 1995.

[6] F. Chen, D. A. Koufaty, and X. Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives. In *Proceedings of the 2009 Joint SIGMETRICS/Performance Conference*, pages 181–192, 2009.

[7] M.-L. Chiang, P. C. H. Lee, and R.-C. Chang. Managing Flash Memory in Personal Communication Devices. In *Proceedings of the 1997 International Symposium on Consumer Electronics*, pages 177–182, 1997.

[8] M.-L. Chiang, P. C. H. Lee, and R.-C. Chang. Using Data Clustering to Improve Cleaning Performance for Flash Memory. *Software: Practice and Experience*, 29(3):267–290, 1999.

[9] 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 2009 ASPLOS Conference*, pages 229–240, 2009.

[10] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee. A Superblock-based Flash Translation Layer for NAND Flash Memory. In *Proceedings of the 6th ACM/IEEE International Conference on Embedded Software*, pages 161–170, 2006.

[11] J. Katcher. PostMark: A New File System Benchmark. Technical Report TR3022, Network Appliance Inc., 1997.

[12] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho. A Space-efficient Flash Translation Layer for CompactFlash Systems. *IEEE Transactions on Consumer Electronics*, 28(2):366–375, 2002.

[13] J.-H. Kim, D. Jung, J.-S. Kim, and J. Huh. A Methodology for Extracting Performance Parameters in Solid State Disks (SSDs). In *Proceedings of the 2009 MASCOTS Symposium*, pages 1–10, 2009.

[14] J. Lee, S. Kim, H. Kwon, C. Hyun, S. Ahn, J. Choi, D. Lee, and S. H. Noh. Block Recycling Schemes and Their Cost-Based Optimization in NAND Flash Memory Based Storage System. In *7th ACM/IEEE International Conference on Embedded Software*, pages 174 – 182, Salzburg, Austria, 2007.

[15] S. Lee, D. Shin, Y.-J. Kim, and J. Kim. LAST: Locality-Aware Sector Translation for NAND Flash Memory-based Storage Systems. *ACM Operating Systems Review*, 42(6):36–42, 2008.

[16] 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*, 6(1), 2007.

[17] Z. Liu, L. Yue, P. Wei, P. Jin, and X. Xiang. An Adaptive Block-Set based Management for Large-Scale Flash Memory. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, pages 1621–1625, 2009.

[18] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the Performance of Log-structured File Systems with Adaptive Methods. *ACM Operating Systems Review*, 31(5):238–251, 1997.

[19] J. Menon. A Performance Comparison of RAID-5 and Log-Structured Arrays. In *Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing*, pages 167–178, 1995.

[20] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, and J.-S. Kim. A Re-configurable FTL (Flash Translation Layer) Architecture for NAND Flash-based Applications. *ACM Transactions on Embedded Computing Systems*, 7(4), 2008.

[21] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[22] E. Seo, S. Y. Park, and B. Urgaonkar. Empirical Analysis on Energy Efficiency of Flash-based SSDs. In *Proceedings of the Workshop on Power Aware Computing and Systems*, 2008.

[23] J.-Y. Shin, Z.-L. Xia, N.-Y. Xu, R. Gao, X.-F. Cai, S. Maeng, and F.-H. Hsu. FTL Design Exploration in Reconfigurable High-Performance SSD for Server Applications. In *Proceedings of the 23rd International Conference on Supercomputing*, pages 338–349, 2009.

[24] W. Wang, Y. Zhao, and R. Bunt. HyLog: A High Performance Approach to Managing Disk Layout. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 145–158, 2004.

[25] C.-H. Wu and T.-W. Kuo. An Adaptive Two-level Management for the Flash Translation Layer in Embedded Systems. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design*, pages 601–606, 2006.

[26] M. Wu and W. Zwaenepoel. eNVy: A Non-Volatile, Main Memory Storage System. In *Proceedings of the 1994 ASPLOS Conference*, pages 86–97, 1994.