Improving Performance and Capacity of Flash Storage Devices by Exploiting Heterogeneity of MLC Flash Memory

Sungjin Lee and Jihong Kim, *Member, IEEE*

Abstract—The multi-level cell (MLC) NAND flash memory technology enables multiple bits of information to be stored in a memory cell, thus making it possible to increase the density of flash memory without increasing the die size. In MLC NAND flash memory, each memory cell can be programmed as a single-level cell or a multi-level cell at runtime because of its performance/capacity asymmetric programming property, which is called flexible programming in this paper. Therefore, MLC flash memory has a potential to achieve the high performance of SLC flash memory while preserving its maximum capacity. In this paper, we present a flexible flash file system, called FlexFS, which takes advantage of flexible programming. FlexFS divides a flash memory medium into SLC and MLC regions, and then dynamically changes two different types of regions to provide an optimal storage solution to end-users in terms of performance and capacity. FlexFS also provides a reasonable storage lifetime by managing the wearing rate of NAND flash memory, which is accelerated by the use of flexible programming. Our implementation of FlexFS in the Linux 2.6 kernel shows that it achieves the I/O performance comparable to SLC flash memory while guaranteeing the capacity of MLC flash memory in various real-world workloads.

Index Terms—NAND flash memory, file system, storage system, operating system

1 Introduction

As NAND flash memory technologies rapidly advance, flash-based storage devices are becoming an attractive storage solution for various applications from mobile consumer devices to high-end server systems. This rapid growth of the NAND flash memory market is largely driven by the desirable characteristics of NAND flash memory, such as high performance, low-power consumption, and high mobility.

There are two types of NAND flash memory in the market: single-level cell (SLC) and multi-level cell (MLC) flash memory. They are distinctive in terms of capacity, performance, and endurance. For example, the capacity of MLC flash memory is larger than that of SLC flash memory. By storing two (or more) bits of information in a memory cell, MLC flash memory achieves significant density increases while lowering the cost per bit over SLC flash memory, which only stores a single bit in a cell. However, SLC flash memory has higher performance and longer cell lifetime than MLC flash memory. In particular, its write (or programming) performance is much higher than that of MLC flash memory.

As the demand for a high capacity storage device is rapidly increasing, MLC flash memory is being widely adopted in many mobile devices, such as mobile phones, digital cameras, laptops, and other portable devices. However, because of its poor performance, it is a challenge to satisfy users' requirements for a high performance storage device with a large capacity.

In order to overcome this poor performance problem, we exploit the performance/capacity asymmetric programming property of MLC NAND flash memory, which we call *flexible programming*. Flexible programming enables each memory cell to be programmed as a single-level cell (SLC-mode programming) or as a multi-level cell (MLC-mode programming). If SLC-mode programming is used to write data to a particular cell, the effective properties of that cell become similar to those of an SLC flash memory cell. Conversely, MLC-mode programming allows us to make use of the high capacity of an MLC flash memory cell. The most attractive aspect of flexible programming is that fine-grained storage optimizations are possible at the system software level, such as a file system, during runtime. By efficiently exploiting flexible programing, it is possible to optimize both the performance and capacity aspects of storage solutions, so that NAND storage solutions can achieve both the SLC performance and the MLC capacity.

In order to realize a high-performance and high-capacity flash-based storage device with flexible programming, however, several technical challenges need to be addressed properly. First, flexible programming allows two different types of memory cells (programmed by either SLC-mode or MLC-mode programming) to coexist in the same flash memory simultaneously. These heterogeneous cells should be managed effectively as if they appear to be homogeneous cells to end-users. Second, with flexible programming, there is a strong trade-off between performance and capacity, which should be carefully managed. For example, if too many

- S. Lee is with the Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139.
- J. Kim is with the Department of Computer Science and Engineering, Seoul National University, Gwanak-ro, Gwanak-gu, Seoul 151-742, Korea. E-mail: jihong@davinci.snu.ac.kr.

*Manuscript received 04 Dec. 2012; revised 17 Apr. 2013; accepted 12 May 2013. Date of publication 26 May 2013; date of current version 12 Sep. 2014. Recommended for acceptance by E.-Y. Chung.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2013.120*

memory cells are programmed as single-level cells, the capacity of flash memory is significantly reduced even though its I/O performance is improved. It is thus important to decide the number of single-level and multi-level cells so that both performance and capacity would be optimally supported. Third, flexible programming requires frequent data movements from single-level cells to multi-level cells for performance and capacity reasons. These data movements incur a large number of extra block erasures, shortening the lifetime of a flash-based storage device. Therefore, this lifetime problem must be properly addressed so that a reasonable storage lifetime can be supported.

In this paper, we propose a flexible flash file system, called *FlexFS*, for MLC NAND flash memory that efficiently addresses those technical challenges. FlexFS provides end-users with a homogeneous view of storage, while internally managing two different types of cells. FlexFS guarantees the maximum capacity of MLC flash memory while achieving the I/O performance close to SLC flash memory. To provide high performance and high capacity simultaneously, FlexFS employs a dynamic free-space management (DFM) technique. FlexFS also adopts a novel dynamic lifetime management (DLM) technique, which manages the storage lifetime by controlling the use of SLC-mode programming.

We implemented FlexFS in the Linux 2.6 kernel on top of our in-house flash storage prototype and evaluated its effectiveness using real-world applications from various consumer devices, including mobile phones, laptops, and desktop PCs. To validate the long-term behaviors of FlexFS, we conducted a simulation study with a trace-driven simulator using block I/O traces collected for several days from various consumer devices. Experimental results showed that FlexFS achieves the performance close to SLC flash memory, while offering the capacity of MLC flash memory.

The rest of this paper is organized as follows. In Section 2, we briefly review NAND flash memory and explain some unique properties of MLC flash memory, which enable flexible programming. In Section 3, we present an overview of FlexFS and describe the proposed dynamic free-space and lifetime management techniques in detail. Our experimental results are given in Section 4, and related work is summarized in Section 5. In Section 6, we conclude with a summary.

2 Background

2.1 NAND Flash Memory

NAND flash memory consists of multiple blocks, each of which is composed of several pages. In many NAND flash memories, the size of a page is between 512 B and 8 KB, and one block consists of between 4 and 128 pages. NAND flash memory does not support an overwrite operation because of its write-once nature. Therefore, before writing new data into a block, the previous block must be erased. Furthermore, the total number of erase operations allowed for a block is typically limited to between 5,000 and 100,000 cycles.

Like SRAM and DRAM, NAND flash memory stores bits in a memory cell, which consists of a transistor with a floating gate that can store electrons. The number of electrons stored on the floating gate determines the threshold voltage, V_t, which represents the state of the cell. In case of SLC flash memory, each cell has two states, and thus only a single bit can

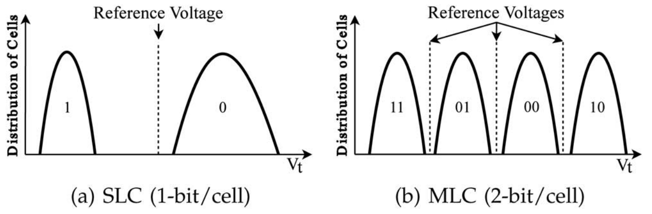


Fig. 1. Threshold voltage V_t distributions of SLC and MLC flash memories.

be stored in that cell. Fig. 1(a) shows how the value of a bit is determined by the threshold voltage. If the threshold voltage is greater than a reference voltage, it is interpreted as a logical '0'; otherwise, it is regarded as a logical '1'. In general, the write operation moves the state of a cell from '1' to '0', while the erase operation changes '0' to '1'.

If flash memory is composed of memory cells which have more than two states, it is called MLC flash memory, and two or more bits of information can be stored on each cell, as shown in Fig. 1(b). Even though the density of MLC flash memory is higher than that of SLC flash memory, it requires more precise charge placement and charge sensing because of narrower voltage ranges for each cell state, which in turn reduces performance and lifetime of MLC flash memory.

2.2 MLC NAND Flash Memory Array

In MLC flash memory, it is possible to use SLC-mode programming, allowing a multi-level cell to be used as a single-level cell. To understand the implications of SLC programming, it is necessary to know the overall architecture of a flash memory array. Fig. 2 illustrates the array of flash memory cells which forms a flash memory block. We assume that each cell is capable of holding two bits. For a description purpose, this figure does not show all the elements, such as source and drain select gates, which are required in a memory array. (For a more detailed description, see references [2], [3].)

As shown in Fig. 2, the memory cells are arranged in an array of rows and columns. The cells in each row are connected to a word line (e.g., $WL(0)$), while the cells in each column are coupled to a bit line (e.g., $BL(0)$). These word and bit lines are used for read and write operations. During a write operation, the data to be written ('1' or '0') is provided at the bit line while the word line is asserted. During a read operation, the word line is again asserted, and the threshold voltage of each cell can then be acquired from the bit line.

Fig. 2 also shows the conceptual structure of a flash block corresponding to an MLC NAND flash memory array. The size of a page is determined by the number of bit lines in the memory array, while the number of pages in each flash block is twice the number of word lines because two different pages share the memory cells that belong to the same word line. These two pages are respectively called the least significant bit (LSB) page and the most significant bit (MSB) page. As these names imply, each page only uses its own bit position of a bit pattern stored in a cell. This is possible because each memory cell stores two bits, for example, one bit for the LSB page and the other for the MSB page. Thus, if a block has 128 pages, there are 64 LSB and 64 MSB pages. A memory cell in an erased state is interpreted as a logical '11'. When a logical '0' is programmed to the LSB position of the cell, the cell will then have a bit pattern of '10', which is interpreted as a logical '0' for

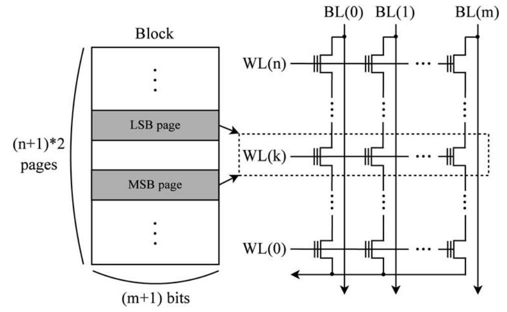


Fig. 2. An organization of an MLC flash memory array.

the LSB page. If the MSB position is then programmed as '0', the bit pattern will change to '00'.

2.3 SLC-Mode and MLC-Mode Programming

In MLC NAND flash memory, LSB pages must be programmed before their corresponding MSB pages.[1] When data are written to a certain LSB page, the distribution of the threshold voltages of cells becomes similar to that of SLC flash memory as illustrated in Fig. 1(a). Then, when data are written to the corresponding MSB page, all available four states in cells are fully utilized, so the threshold voltage distribution is changed to that shown in Fig. 1(b). Because of this programming mechanism of MLC flash memory [4], if LSB pages in a block are only used for writing data, that block becomes similar to a block in SLC flash memory in terms of performance and capacity. For example, the performance characteristic of the block is improved, but its capacity is reduced by half in comparison with MLC flash memory. On the other hand, if both LSB and MSB pages in a block are used for writing data, its physical characteristics remain the same as those of MLC flash memory. Note that recent 3-bit MLC flash memory also supports SLC-mode programming [5], which is similar to that of 2-bit MLC flash memory.

In flexible programming, SLC-mode programming is realized by writing data to only LSB pages in a block. Conversely, MLC-mode programing is made by using both LSB and MSB pages for writing data. Furthermore, the datasheets of NAND flash chips specify the offsets of LSB and MSB pages in a block, and thus two different types of pages can be accessed by the primitive I/O operations of NAND flash memory, such as a page read or page write operation. Therefore, SLC-mode and MLC-mode programming is easily controlled by system software such as a file system at runtime.

Table 1 compares the performances of two different types of programming modes. We used two different MLC NAND flash chips: Samsung's KFXXGH6X4M [6] and Micron's MT29F8G08AAA [7]. The $MLC_{\rm LSB}$ column shows the read and write response times when only LSB pages are used, whereas the $MLC_{\rm BOTH}$ column shows the performances when both LSB and MSB pages are used. All the data were measured in a block device driver. As shown in Table 1, there were no significant performance differences between page read and block erase operations. However, the write performance was greatly improved with $MLC_{\rm LSB}$, which was almost equal to that of pure SLC flash memory fabricated at

1. The programming order of LSB and MSB pages can be exchanged, depending on flash memory manufacturers.

TABLE 1
Performance Comparisons of Different Types of Programming Modes (Unit: μsec)

	KFXXGH6X4M [6]		MT29F8G08AAA [7]	
	$MLC_{\rm LSB}$	$MLC_{\rm BOTH}$	$MLC_{\rm LSB}$	$MLC_{\rm BOTH}$
Read (page)	409	403	375	405
Write (page)	431	994	421	834
Erase (block)	872	872	1581	1581

the same process. This high write performance of SLC-mode programming is the main motivation of FlexFS. Our primary goal is to improve the write performance using SLC-mode programming, while maintaining the capacity of MLC flash memory using MLC-mode programming.

3 Design and Implementation of FlexFS

In order to achieve the SLC performance, FlexFS writes as many data as possible to flash memory using fast SLC-mode programming. However, the excessive use of SLC-mode programming rapidly exhausts free space available for writing data because it wastes MSB pages in blocks, leaving them unused. This wasted storage space can be reclaimed by moving valid data in blocks (which were programmed by SLC-mode programming) to other blocks using MLC-mode programming. In this paper, this process is called *free-space reclamation*. Free-space reclamation requires many page migrations as well as block erase operations, but it can be done with a low performance penalty by leveraging storage idle time. Usually, most consumer devices, such as mobile phones, laptops, and desktop PCs, have a considerable amount of idle time (e.g., average idle time is 83%-98%). Thus, FlexFS can continuously create sufficient free space so that all of the requested data can be written using SLC-mode programming.

The approach mentioned above seems to be effective, but it poses several technical issues that must be properly handled. First, heterogeneous memory cells must be managed appropriately at the level of a file system. In FlexFS, the number of different types of memory cells is changed at runtime, and furthermore they are scattered across a storage medium. Second, the amount of free space available for writing must be properly managed. If free-space reclamation performs too frequently, sufficient free space can be maintained for SLC-mode programming. However, this frequent reclamation often incurs useless data migrations (which turn out to be unnecessary later), thereby shortening the storage lifetime uselessly. On the other hand, if free-space reclamation is conducted too infrequently, available free space is quickly exhausted. Thus, FlexFS cannot write incoming data to flash memory, even though a storage capacity offered to end-users is not fully utilized. Third, FlexFS writes incoming data to flash memory using SLC-mode programming and then moves them to other locations using MLC-mode programming for free-space reclamation, which requires lots of block erasures. For this reason, the storage lifetime could be seriously limited because flash memory is more rapidly worn out.

FlexFS is designed to properly deal with such technical issues. FlexFS efficiently manages heterogeneous memory cells, giving an illusion to end-users that they are using a homogeneous storage device with high performance and high capacity as well as a reasonable lifetime. These benefits of FlexFS can be realized by adopting two novel techniques:

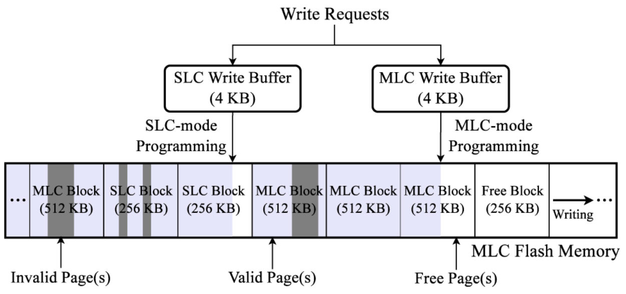


Fig. 3. A layout of flash blocks in FlexFS.

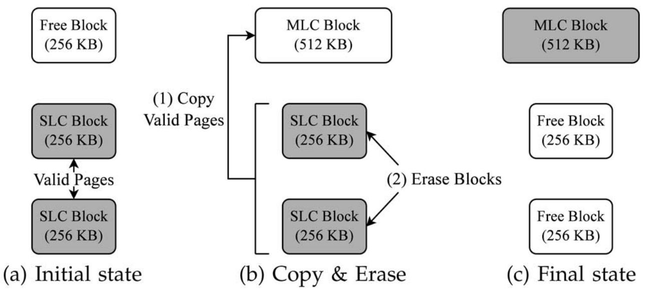


Fig. 4. Free-space reclamation.

dynamic free-space and lifetime management techniques. The dynamic free-space management (DFM) technique resolves the problems caused by improper management of free space by maintaining minimal but sufficient free space. The dynamic lifetime management (DLM) technique adaptively controls the wearing rate of flash memory (which is accelerated by free-space reclamation) so that a reasonable storage lifetime can be provided.

In the following subsection, we present the overall architecture of FlexFS, focusing on its heterogeneous cell management. The detailed descriptions of the dynamic free-space and lifetime management techniques are given in Sections 3.2 and 3.3, respectively.

3.1 Management of Heterogeneous Cells

FlexFS is based on a JFFS2 file system [8], so its architecture is similar to JFFS2, except for some features required to manage heterogeneous cells. Thus, we focus on how FlexFS deals with different types of memory cells.[2]

Fig. 3 shows the layout of flash blocks in FlexFS and how it handles write requests.[3] We assume that the number of pages in a block is 128, and a page size is 4 KB. These values will be used throughout the rest of this paper. FlexFS logically divides the flash memory medium into two regions: an SLC region and an MLC region. The SLC region is composed of SLC blocks programmed by SLC-mode programming, and the MLC region consists of MLC blocks programmed by MLC-mode programming. If a block does not contain any data, it is called a free block. In FlexFS, a free block is neither an SLC block nor an MLC block; its type is determined when data are written into it later. For simplicity's sake, we assume that the size of a free block is the same as that of an SLC block. Regardless of the number of SLC, MLC, or free blocks, FlexFS provides the maximum capacity of MLC flash memory to end-users.

When a write request arrives, FlexFS decides the type of a region to which data are to be written and stores requested data temporarily in a proper write buffer, which is separately managed for two different regions. This temporary buffering is necessary because the unit of read and write operations is a page in flash memory. FlexFS performs write operations in a similar fashion to other log-structured file systems [8]–[10], except that two log blocks (one for the SLC region and the other for the MLC region) are reserved for writing. When data are evicted from the write buffer, FlexFS writes them to the log block of a corresponding region using a proper programming mode. If existing data are updated in flash memory, the old version of the data is first invalidated, while the new data are appended to the free space of a log block. The space occupied by the invalid data is reclaimed by garbage collection later [1].

FlexFS has a special operation, called free-space reclamation, which moves valid pages in SLC blocks to MLC blocks to expand available free space in flash memory. Fig. 4 shows how free-space reclamation increases available free space. Initially, there are two SLC blocks and one free block. We assume that SLC blocks contain only valid pages. To create free space, FlexFS copies 128 pages in two SLC blocks to the free block using MLC-mode programming. The free block accordingly becomes a new MLC block. Then, two SLC blocks are erased and become free blocks. As a result, free-space reclamation frees up one block, increasing available free space.

Free-space reclamation incurs lots of extra I/O operations. To prevent user I/O requests from being delayed by such extra I/Os, free-space reclamation is performed only when there are no user I/O activities and on-demand free-space reclamation is not required (see Section 3.2). If an observed idle period is longer than a certain threshold value, FlexFS triggers free-space reclamation, expecting that there will be a long idle period. This threshold value must be carefully decided. If a threshold value is too short, a great performance penalty caused by background free-space reclamation cannot be avoided. On the other hand, if a threshold value is too long, the amount of idle time that can be exploited for free-space reclamation is reduced, lowering the efficiency of free-space reclamation. Currently, FlexFS uses a fixed threshold value which is set to 50 ms (see Section 4.1.1). In our observation, a threshold value of 50 ms is long enough not to incur a significant I/O performance penalty even with I/O intensive workloads. Even though a relatively long threshold value is chosen, it is short enough to create sufficient free space in real-world workloads without losing lots of chances of utilizing idle time for free-space reclamation.

Before finishing this subsection, we summarize some terms that represent free space in FlexFS. In this paper, the amount of free space currently available for writing data to the SLC region is called *currently available free space* or simply called *free space*. On the other hand, from end-users' perspective,

2. A detailed explanation of traditional file-system design issues (e.g., the management of files/directories and the process of file-system mount/unmount) is omitted in this paper because they are almost the same as those of JFFS2. Note that FlexFS shares the same limitations as JFFS2 like slow mount time. However, these inherited limitations can be overcome by implementing the main features of FlexFS in a recent file system (e.g., UBIFS) because FlexFS is designed to work independently of JFFS2 in principle.

3. For a more detailed description of how FlexFS handles read requests, please refer to our previous work [1].

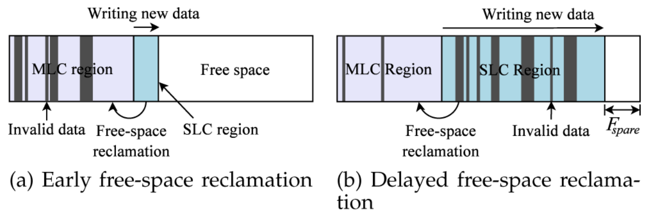


Fig. 5. A comparison of early free-space reclamation and delayed free-space reclamation.

available free space is the amount of empty space not occupied by user data, and it is offered to end-users based on the capacity of MLC flash memory. To differentiate it from the former one, this empty space is called *unused storage space*. In the example of Fig. 4, free space is changed from 256 KB to 512 KB, but unused storage space is fixed to 1 MB.

3.2 Dynamic Free-Space Management

FlexFS has to maintain sufficient free space to write requested data to the SLC region. If available free space is exhausted, FlexFS must perform free-space reclamation *on demand* to make free space, suspending user I/O requests. The requested data then can be written to the newly reclaimed free space. This *on-demand free-space reclamation* incurs a great performance penalty. As shown in Fig. 4, for the free space of 256 KB to be reclaimed, FlexFS needs to move the data of 512 KB to the MLC region. Due to this high migration cost, I/O performance becomes lower than that of MLC flash memory.

The simplest approach that maintains sufficient free space is to reclaim as much free space as possible during idle time. This approach is called *early free-space reclamation* in which it creates free space as early as possible whenever available idle time is observed. Considering plenty of idle time available in consumer devices, FlexFS creates sufficient free space for SLC-mode programming. Note that the case where the amount of idle time is not enough is discussed in Section 3.2.3.

Even though early free-space reclamation helps us to keep a large amount of free space, it frequently incurs useless free-space reclamation that moves the data to be invalidated soon to the MLC region. As shown in Fig. 5(a), early free-space reclamation moves data to the MLC region right after they are written to the SLC region if idle time is observed. However, many of data are updated or deleted in the MLC region and become invalid soon. Therefore, the migration of those data turns out to be useless. Free-space reclamation requires many block erase operations, so useless free-space reclamation adversely affects the lifetime of a flash device.

To resolve such a lifetime problem, FlexFS employs a *delayed free-space reclamation* policy that delays free-space reclamation as long as possible so that many data are invalidated in the SLC region. As shown in Fig. 5(b), delayed free-space reclamation does not trigger free-space reclamation unless available free space is smaller than F_{spare}. Here, F_{spare} is the amount of spare free space that must be maintained in flash memory, so as to prevent on-demand free-space reclamation. If available free space becomes smaller than F_{spare}, free-space reclamation is invoked and then is conducted like early free-space reclamation until the free space of F_{spare} is to be reclaimed. FlexFS chooses infrequently updated data as victim data for free-space reclamation. This helps us to further reduce useless data migration.

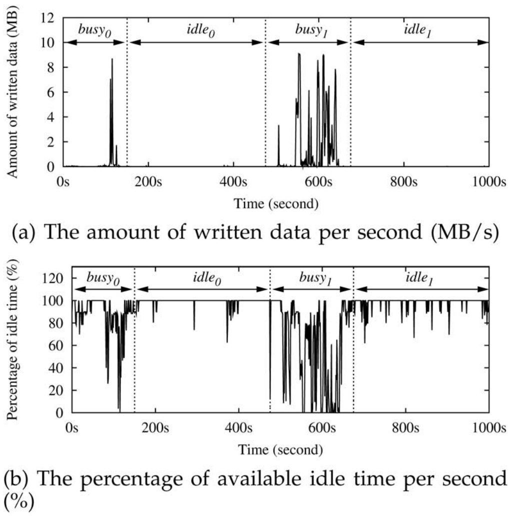


Fig. 6. Characteristics of I/O traffic in a laptop PC.

As astute readers may notice, FlexFS uses unused storage space as the temporal SLC buffer for storing incoming data rapidly. According to [15]–[17], a storage device is not always full and a large amount of storage space remains unused. Thus, this SLC buffering is useful in most cases except when a storage device is nearly full. In Section 3.2.2, we discuss how FlexFS handles the case where unused storage space is almost exhausted.

In FlexFS, F_{spare} must be carefully decided because it greatly affects not only storage performance but also its lifetime. If F_{spare} is too large, useless free-space reclamation is frequently conducted like early free-space reclamation. If F_{spare} is too small, performance degradation caused by on-demand free-space reclamation is frequently observed. Thus, our goal is to maintain F_{spare} as small as possible so long as free space exhaustion does not occur. In the following subsection, we describe how FlexFS determines F_{spare}.

3.2.1 Determination of F_{spare}

We first analyze two important characteristics of I/O traffic, including the amount of written data and the amount of available idle time, which have a great effect on the determination of F_{spare}. Fig. 6 displays the amount of data written per second and the percentage of available idle time per second, both of which are observed in a laptop PC. As shown in Fig. 6, a storage device is usually in an idle state for a significant amount of time, which is denoted by $idle_0$ and $idle_1$. Conversely, a large amount of data is written to a storage device during a short time period, which is denoted by $busy_0$ and $busy_1$. These idle and busy periods are irregularly repeated over time.

The observation in Fig. 6 gives an important clue to deciding F_{spare}. Suppose that the amount of free space required for storing the data issued during $busy_1$ is F_1. Given sufficient idle time, FlexFS can reclaim the free space F_1 required for $busy_1$ during available idle periods (e.g., $idle_0$ in Fig. 6). Once sufficient free space is obtained, it is not necessary to further perform free-space reclamation because more free space will be not required. Suppose further that there are $(n+1)$ busy/idle periods, $busy_0, idle_0, \ldots, busy_n, idle_n$, and FlexFS performs free-space reclamation to create the exact

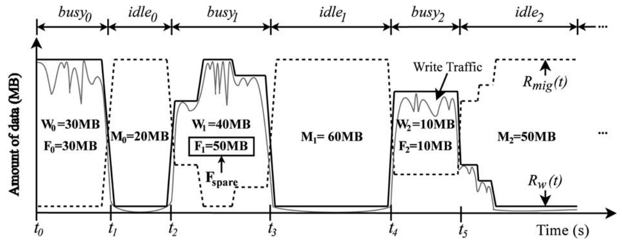


Fig. 7. Determination of F_{spare} by monitoring busy and idle periods.

amount of free space $F_0, \ldots, F_n$ for the respective busy periods $busy_0, \ldots, busy_n$. All the incoming data can be written to the SLC region without free-space exhaustion and useless free-space reclamation can be minimized.

If the amount of free space for future busy periods is known in advance, FlexFS can control F_{spare} so that the exact amount of free space is prepared for the next busy period. However, it is difficult to predict free space for future busy periods because write traffic and idle time vary greatly over time. For this reason, FlexFS determines F_{spare} in a conservative manner; it chooses the largest amount of free space required for past busy periods as F_{spare}, expecting that the free space required for future busy periods would be not larger than the largest one previously observed.

Fig. 7 illustrates how FlexFS determines F_{spare} by referring to the history of I/O traffic. Assume that the unit time interval is one second. At a certain time t, $R_w(t)$ is the amount of written data to the SLC region and $R_{mig}(t)$ is the amount of free space that *can be* reclaimed during available idle time. FlexFS divides the time into pairs of two consecutive periods, a busy period and an idle period. The time intervals during which $R_{mig}(t) \leq R_w(t)$ are regarded as a busy period. The time intervals during which $R_{mig}(t) > R_w(t)$ are considered as an idle period. Suppose that there are $(n+1)$ pairs of busy/idle periods, and $busy_k$ and $idle_k$ for $k = 0, 1, \ldots, n$ denote individual busy and idle periods. The amount of data written in the SLC region during a busy period $busy_k$ is denoted by W_k. The amount of free space that can be created during an idle period $idle_k$ is denoted by M_k. For example, in Fig. 7, W_1 is $\sum_{t=t_2}^{t_3}(R_w(t) - R_{mig}(t)) = 40$ MB, and M_1 is $\sum_{t=t_3}^{t_4}(R_{mig}(t) - R_w(t)) = 60$ MB.

The amount of free space F_k required for a busy period $busy_k$ can be estimated using the equation below:

$$\begin{gathered} F_0 = W_0 \ (k = 0) \\ F_1 = \max(0, F_0 - M_0) + W_1 \\ F_2 = \max(0, F_1 - M_1) + W_2 \\ \vdots \\ F_n = \max(0, F_{n-1} - M_{n-1}) + W_n \ (k > 0). \end{gathered} \tag{1}$$

In Eq. (1), F_0 is the amount of free space required to store data issued during $busy_0$, whereas M_0 is the amount of free space that can be reclaimed during the idle period $idle_0$. If $F_0 > M_0$, the data of $(F_0 - M_0)$ still remain in the SLC region at the beginning of $busy_1$ because of the insufficient idle time. Since the data of W_1 are requested for writing during $busy_1$, the free space of $(F_0 - M_0) + W_1$ is required for $busy_1$. If $F_0 \leq M_0$, the free space required for $busy_1$ is W_1. For example, F_0, F_1, and F_2 in Fig. 7 are 30 MB, 50 MB, and 10 MB, respectively.

Among all the values of F_k for $k = 0, 1, \ldots, n$, the largest one is the maximum free space that was required by previously observed busy periods. Suppose that the similar write traffic will be requested in the future and the amount of free space required by future busy periods is not larger than the largest one previously observed. In that case, FlexFS does not need to perform free-space reclamation if more free space than the largest one exists in flash memory. In that sense, FlexFS chooses the largest one of all the values of F_k as F_{spare}, which is formally expressed as follows:

$$F_{spare} = \max_{k=0}^{n}(F_k). \tag{2}$$

If an unexpectedly large amount of data is requested, (e.g., more free space than 50 MB are requested in the example of Fig. 7), on-demand free-space reclamation inevitably occurs. In practice, free-space exhaustion is not frequently observed because F_{spare} is decided in a very conservative way. Furthermore, since a storage device is initially empty, it would take a long time until available free space is reduced to F_{spare}. Thus, FlexFS is trained sufficiently by monitoring I/O traffic for a long time before free space becomes smaller than F_{spare}. Note that, in this paper, this monitoring period is called a *training period*.

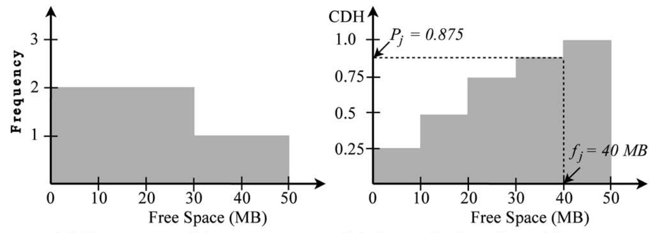
The value of F_{spare} is obtained with small memory and computational overheads. To calculate F_k, it is only necessary to keep F_{k-1} and M_{k-1} for the previous busy/idle period, and W_k for the current busy period. If $F_k > F_{spare}$, F_k becomes new F_{spare}; otherwise, it is discarded. The computational cost for profiling the amount of written data W_k and the amount of data reclaimed during idle time M_k was very low.

3.2.2 Management of Insufficient Free Space

FlexFS uses available free space as temporary SLC buffer to speed up write performance. If an unused storage space is large enough, FlexFS achieves high performance by writing all the data to the SLC region while reclaiming free space for future use. However, if a storage device is almost full (i.e., only a small number of free blocks remain) or if free space required for busy periods is too large (i.e., F_{spare} is too large), it is hard to provide SLC performance because FlexFS cannot create sufficient free space for SLC-mode programming.

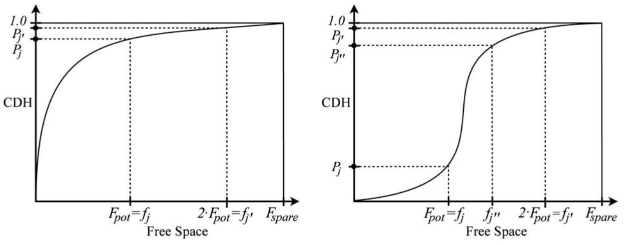
FlexFS judges that there is insufficient free space when *potential free space* F_{pot} *is smaller than* F_{spare}. F_{pot} is similar to currently available free space as mentioned in Section 3.1, but it includes the space occupied by invalid and wasted pages as free space, which can be free after garbage collection or free-space reclamation. That is, F_{pot} indicates the maximum amount of data that can be stored in the SLC region. Note that F_{pot} is also the same as half of an unused storage capacity, which is available storage space seen by end-users. If $F_{pot} < F_{spare}$, it means that FlexFS cannot create more free space than F_{spare}, regardless of the amount of idle time. If SLC-mode programming is used for writing data even when $F_{pot} < F_{spare}$, free space could be exhausted before all the requested data are written to the SLC region. In that case, FlexFS must trigger on-demand free-space reclamation to create more free space, which incurs a great performance penalty.

One of the feasible approaches that avoid on-demand free-space reclamation while providing relatively high



(a) Free-space histogram (b) Cumulative data histogram

Fig. 8. A free-space histogram and a cumulative data histogram (CDH) corresponding to the example in Fig. 7.



(a) P_j is high (b) P_j is low

Fig. 9. Two representative examples of CDHs.

performance is to adaptively use two different types of programming modes according to the distribution of free space required for busy periods. For example, if almost all busy periods require smaller free space than F_{pot} except only a few busy periods requiring larger free space than F_{pot}, it might be better to keep writing data using SLC-mode programming. This is because almost all data can be written to the SLC region without incurring on-demand free-space reclamation. On the other hand, if most of the busy periods require larger free space than F_{pot}, using MLC-mode programming is a better choice because it avoids a performance penalty caused by free space exhaustion.

In order to figure out the distribution of required free space, FlexFS keeps the free-space histogram using the values of F_{k-1}, M_{k-1}, and W_k for individual busy periods $busy_k$. Fig. 8(a) shows how FlexFS composes the free-space histogram. For $busy_0$ in Fig. 7, W_0 are 30 MB. (Note that F_{k-1} and M_{k-1} are not available). It means that $busy_0$ requires the free space of 30 MB. In Fig. 8(a), a bin width is assumed to be 10 MB. Thus, the frequencies of 3 histogram bins ranging from 0 MB to 30 MB are increased by 1. For $busy_1$, the values of F_0, M_0, and W_1 are 30 MB, 20 MB, and 40 MB, respectively. As explained in Section 3.2.1, $(F_{n-1} - M_{n-1})$ is the amount of free space consumed by previous busy periods (i.e., $busy_0$ in Fig. 7, and, for $busy_1$, it is 10 MB. Since the data of 40 MB are written during $busy_1$, the frequencies of 4 histogram bins ranging from 10 MB to 50 MB are increased by 1. In this manner, $busy_2$ can be added to the histogram.

Using the free-space histogram, FlexFS constructs a cumulative data histogram (CDH). The CDH consists of a list of (f_j, P_j) pairs. A finite number of (f_j, P_j) pairs are indexed by the histogram bin j, where f_j is the smallest free space that falls into the j-th histogram bin and P_j is the responding empirical cumulative probability of occurrence $P_j = Pr(\textit{free space} \leq f_j)$. Fig. 8(b) shows the CDH corresponding to Fig. 8(a). Using the CDH, it is possible to estimate how many data can be written to flash memory without requiring more free space than a given one. For example, if f_j is 40 MB, $P_j = Pr(\textit{free space} \leq$ 40 MB) is 0.875. That is, 87.5% of future write requests are likely to require less free space than 40 MB.

Fig. 9 shows two representative examples of CDHs. Fig. 9(a) is the CDH of I/O traffic where a majority of busy periods require a small amount of free space, whereas Fig. 9(b) is the CDH where almost all busy periods require a large amount of free space. Suppose that F_{pot} is denoted by f_j in the CDHs of Fig. 9. Even for the same f_j, P_j of Fig. 9(a) is close to 1.0, but P_j of Fig. 9(b) is close to 0.0. As the value of P_j increases, more data can be written to the same free space without incurring on-demand free-space reclamation. For this reason, for I/O traffic with a similar CDH to Fig. 9(a), it would be better to use SLC-mode programming. On the other hand, MLC-mode programming would be preferred for I/O traffic with the CDH of Fig. 9(b).

In order to choose a proper programming mode, FlexFS needs to estimate the expected write performance depending on the programing mode using the CDH of given I/O traffic. First, if MLC-mode programming is used, the amount of data that can be stored in the potential free space F_{pot} is increased to $2 \cdot F_{pot}$, which is denoted by $f_{j'}$ in Fig. 9. Since the unused storage space offered to end-users is $2 \cdot F_{pot}$ as well, more data than $f_{j'}$ cannot be requested. Thus, there will be no free-space exhaustion, but write performance is reduced to that of MLC flash memory. If the time taken to write a page to the MLC region is denoted by T_{mlc}, the expected page write performance E_{mlc} with MLC-mode programming is defined as follows:

$$E_{mlc} = T_{mlc}. \tag{3}$$

Second, if SLC-mode programming is used, the probability of writing data without free-space exhaustion is $(P_j/P_{j'})$ according to the CDH shown in Fig. 9. The probability that free-space exhaustion occurs (i.e., more data than f_j are to be written) is $(1 - P_j/P_{j'})$. When free-space exhaustion occurs, FlexFS moves data in the SLC region to the MLC region, and then writes pending data to the MLC region.[4] If the time taken to move a single page for free-space reclamation is T_{mig} and the time spent to write a page to the SLC region is T_{slc}, the expected write performance E_{slc} with SLC-mode programming is defined as follows:

$$E_{slc} = \frac{P_j}{P_{j'}} \cdot T_{slc} + \left(1 - \frac{P_j}{P_{j'}}\right) \cdot (T_{mig} + T_{mlc}). \tag{4}$$

In Eq. (4), as the value of $(P_j/P_{j'})$ increases, E_{slc} decreases because more data can be written to the SLC region with less free-space reclamation. If $E_{slc} < E_{mlc}$, it is better to use SLC-mode programming; otherwise, MLC-mode programming is preferred. In our measurement, T_{slc}, T_{mlc}, and T_{mig} are 421 μsec, 834 μsec, and 1.5 $m$sec, respectively. According to Eqs. (3) and (4), $E_{slc} < E_{mlc}$ when $(P_j/P_{j'})$ is higher than 0.77.

By choosing the most appropriate programming mode, FlexFS provides higher performance than MLC flash memory even when available free space is smaller than F_{spare}. However, when $(P_j/P_{j'})$ is low (i.e., lower than 0.77) as depicted in

4. If the pending data are still written to the SLC region, they could cause additional free-space reclamation. This is the reason why FlexFS writes them to the MLC region when free-space exhaustion occurs.

Fig. 9(b), the performance is inevitably reduced to that of MLC flash memory.

To further improve performance when $(P_j/P_{j'})$ is low, FlexFS employs a more sophisticated writing strategy that writes part of the incoming data to the MLC region. If some of the incoming data are written to the MLC region while sending the rest of them to the SLC region, the exhaustion of free space can be delayed because more data can be stored in the same free space. As shown in Fig. 9(b), the amount of data that can be written to flash memory without on-demand free-space reclamation is increased to $f_{j''}$. As expected, $f_j \leq f_{j''} \leq f_{j'}$, and thus $P_j \leq P_{j''} \leq P_{j'}$. The probability that free-space exhaustion occurs becomes $(1 - P_{j''}/P_{j'})$, which is lower than the probability $(1 - P_j/P_{j'})$ when only SLC-mode programming is used. In the example of Fig. 9(b), $P_{j''}$ is close to P_j, and $(1 - P_{j''}/P_{j'}) \approx 0.0$. As a result, by writing part of the incoming data to the MLC region, lots of on-demand free-space reclamation can be avoided even if write performance is reduced to that between SLC and MLC performances.

The proportion of the incoming data sending to the SLC region is denoted by α, where $0 \leq \alpha \leq 1$. As the value of α approaches 1.0, a large amount of data is written to the SLC region. To offer the optimal I/O performance, the value of α must be carefully decided because it determines (1) the value of $f_{j''}$, (2) the value of $P_{j''}$, and (3) the amount of the data written to the SLC region. The following equation formalizes the relationship between $f_{j''}$ and α.

$$f_{j''}(\alpha) = f_j \cdot \alpha + f_{j'} \cdot (1 - \alpha). \tag{5}$$

Using $f_{j''}(\alpha)$ in Eq. (5), the value of $P_{j''}$ depending on α is obtained using the CDH as follows:

$$P_{j''}(\alpha) = Pr(free\,space < f_{j''}(\alpha)). \tag{6}$$

The time taken to write a page to flash memory depending on the value of α is formalized as follows:

$$T_{sm}(\alpha) = T_{slc} \cdot \alpha + T_{mlc} \cdot (1 - \alpha). \tag{7}$$

Based on Eqs. (4)–(7), the expected write performance $E_{sm}(\alpha)$ according to the value of α is defined as follows:

$$\begin{aligned} E_{sm}(\alpha) = \; & \tfrac{P_{j''}(\alpha)}{P_{j'}} \cdot T_{sm}(\alpha) \\ & + (1 - \tfrac{P_j'(\alpha)}{P_{j'}}) \cdot (T_{mig} + T_{mlc}). \end{aligned} \tag{8}$$

To determine the optimal value of α, FlexFS calculates the expected performance $E_{sm}(\alpha)$ while increasing α from 0.0 to 1.0 by 0.1, and chooses α that minimizes $E_{sm}(\alpha)$. Note that Eqs. (3) and (4) are the special cases of Eq. (8) where the values of α are 0.0 and 1.0, respectively.

Finally, we discuss memory and computational overhead issues in obtaining the value of α. FlexFS maintains the free-space histogram and updates the frequencies of histogram bins. To compute the new value of α, FlexFS constructs the CDH using the free-space histogram. Building the CDH requires extra computational cost. Thus, FlexFS computes the new value of α every 1,024 page writes. The bin width of a histogram is also set to 4 MB. For a 32 GB storage device, there are 8,192 histogram bins. Since F_{spare} is much smaller than a storage capacity, the number of the histogram bins actually used in building the CDH is smaller than 8,192. This helps us to further reduce the cost of obtaining the value of α. According to our implementation study, the time taken to build the CDH is 194 μsec, on average, in an embedded processor running at 400 MHz. Regarding the memory overhead, only two plain arrays are necessary to maintain the free-space histogram and the CDH. Therefore, the memory space overhead is 64 KB for a 32 GB flash device.

3.2.3 Free Space Management under Insufficient Idle Times

If available idle time is very short, FlexFS cannot create sufficient free space for SLC-mode programming. In that case, the data requested for writing are accumulated in the SLC region over time, and consequently they incur on-demand free-space reclamation. FlexFS resolves this problem by reducing the amount of data written to the SLC region smaller than the amount of free space reclaimed during available idle time.

Suppose that there are $(\mathrm{n}+1)$ busy and idle periods. According to Eq. (1), the average amount of data newly written is $W_{avg}(= \sum_{k=0}^{n+1} W_k/(n+1))$ and the average amount of free space to be reclaimed during idle time is $M_{avg}(= \sum_{k=0}^{n+1} M_k/(n+1))$. If $W_{avg} > M_{avg}$, it means that there is not sufficient idle time. In that case, FlexFS changes the maximum value of α to (M_{avg}/W_{avg}) to limit the data written to the SLC region smaller than the data to be reclaimed during idle time. To prevent a premature decision and change the value of α in response to a changing workload, FlexFS computes the new value of α whenever the amount of newly written data exceeds half of the total storage capacity.

Note that most consumer devices exhibit sufficient idle time. Thus, in our evaluation, a performance penalty caused by the lack of idle time is not observed.

3.3 Dynamic Lifetime Management

FlexFS prolongs the storage lifetime by employing delayed free-space reclamation. However, it does not mean that FlexFS always provides a reasonable storage lifetime. In FlexFS, each block undergoes more P/E cycles because a lot of data are temporarily written to the SLC region, waiting to be moved to the MLC region during idle periods. In order to provide a longer storage lifetime, it would be better to write data to the MLC region directly. However, this reduces the overall performance. To efficiently deal with such a performance/lifetime trade-off, we propose a novel dynamic lifetime management (DLM) technique that controls the amount of data to be written to the SLC region, so as to achieve a reasonable storage lifetime.

3.3.1 Minimum Lifetime Metric

We start by introducing a new lifetime metric which is designed to express the trade-off between lifetime and performance. The maximum lifetime L_{max} of flash memory depends on the storage capacity and the number of P/E cycles as well as the amount of written data [11], and is expressed as follows:

$$L_{max} = \frac{C_{total} \cdot \; E_{cycles}}{R_{write}}, \tag{9}$$

where C_{total} is the total capacity of flash memory, and E_{cycles} is the number of P/E cycles allowed for each block. The writing rate R_{write} indicates the amount of data written in a unit time period (e.g., day).

L_{max} is not appropriate to express the trade-off between lifetime and performance because it just shows an expected storage lifetime. Therefore, we use an explicit lifetime L_{min}, which represents the minimum lifetime that must be ensured by a file system. L_{min} is specified by flash storage manufacturers, and is usually set to 2-5 years. FlexFS can control the writing rate R_{write} by adjusting the amount of write traffic sent to the SLC region. Thus, the trade-off between performance and lifetime can be expressed as follows:

$$\begin{aligned}&\text{Control } R_{write} \text{ by changing } \delta\\&\text{Subject to}\\&L_{min} \approx \frac{C_{total} \cdot \ E_{cycles}}{R_{write}},\end{aligned} \tag{10}$$

where δ is a write-acceleration index, which represents the ratio of the data destined for the SLC region to the total incoming data. If δ is close to 1.0, FlexFS writes all the data to the SLC region, and this increases R_{write} because of frequent free-space reclamation. If δ is close to 0.0, R_{write} is decreased. FlexFS controls δ so that the lifetime specified by L_{min} is to be satisfied.

3.3.2 Assignment of Writing Budget

FlexFS divides the lifetime L_{min} into n time windows w_i for $i = 0, \ldots, n-1$, and the length of a time window is T_w. Each time window is a unit time period for managing a storage lifetime, so the length of T_w must be properly decided. We discuss this issue in Section 3.3.3. Suppose that $WB(w_i)$ is the writing budget assigned to w_i, which represents the amount of flash space allowed to be used for writing data during w_i. The writing rate $R_{write}(w_i)$ for w_i can be expressed as $WB(w_i)/T_w$.

The assignment of writing budget to each window impacts greatly on both the performance and the rate at which flash memory wears out because it determines the amount of data to be written to the SLC or MLC region. In this work, FlexFS equally distributes available writing budget to all time windows. Therefore, $WB(w_i)$ can be expressed as follows:

$$WB(w_i) = \frac{(C_{total} \cdot E_{cycles}) - W(w_{i-1})}{n-i}, \tag{11}$$

where $W(w_{i-1})$ indicates the amount of writing budget that has actually been used by w_{i-1}. If $(i-1) <= 0$, $W(w_{i-1}) = 0$. The remaining writing budget is $(C_{total} \cdot E_{cycles}) - W(w_{i-1})$, and the number of the remaining windows is $(n-i)$. Thus, the remaining writing budget is shared equally by the remaining windows. Note that $WB(w_i)$ is computed at the beginning of w_i.

3.3.3 Determination of a Write-Acceleration Index

Once writing budget $WB(w_i)$ has been assigned to a time window w_i, FlexFS adjusts the write-acceleration index δ so that writing budget actually used in the future time window w_i is smaller than or is equal to $WB(w_i)$.

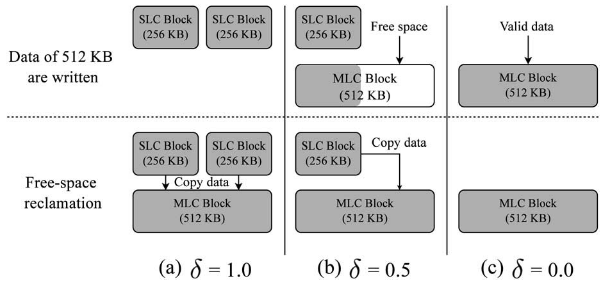


Fig. 10. The amount of writing budget used depending on the value of δ.

Fig. 10 shows how the amount of writing budget used for writing data is changed depending on the value of δ. Suppose that 512 KB data are requested for writing. If δ is 1.0, 512 KB data are written to two SLC blocks. If the data in two SLC blocks are not invalidated, 512 KB data will be moved to one MLC block for free-space reclamation. The total amount of writing budget used is thus 1.5 MB because three blocks have been used for writing data. If δ is 0.5, the writing budget of 1 MB is used because it requires two blocks for writing data. Finally, if δ is 0.0, 512 KB data are written to one MLC block. Thus, the writing budget of 512 KB is used.

This simple example suggests that we can generalize the relationship among the write-acceleration index, the amount of incoming data, and the amount of writing budget actually used in the following way:

$$OW(w_i) = IW(w_i) \cdot (\delta + 1) + IW(w_i) \cdot \delta \cdot \mu, \tag{12}$$

where $IW(w_i)$ is the amount of the data that will arrive during the future time window w_i, and $OW(w_i)$ is the amount of the writing budget used depending on δ. More specifically, $IW(w_i) \cdot (\delta + 1)$ is the writing budget used for writing incoming data to flash memory. For example, in Fig. 10(a), $IW(w_i) \cdot (\delta + 1)$ is 1 MB. On the other hand, $IW(w_i) \cdot \delta \cdot \mu$ is the writing budget which is used for free-space reclamation. Some of the data stored in the SLC region could be invalidated before being moved to the MLC region. The value of μ is the ratio of the data moved for free-space reclamation to the entire data written to the SLC region. If μ is 1.0, it means that all the written data are moved to the MLC region. On the other hand, if μ is 0.0, none of the data are moved. For instance, in Fig. 10(a), δ is 1.0 and μ is 1.0, and thus $IW(w_i) \cdot \delta \cdot \mu = 512$ KB. As a result, $OW(w_i)$ is 1.5 MB. If μ is 0.0, $OW(w_i)$ is 1.0 MB. Since the value of μ is changed according to a workload, the average amount of data moved to the MLC region for free-space reclamation is used for estimating μ.

FlexFS writes as many hot data as possible to the SLC region if it is necessary to regulate write traffic for a lifetime guarantee. If hot data are selectively written to the SLC region, the value of μ can be reduced because they will be invalidated before being moved to the MLC region. This helps us to improve both storage performance and lifetime. Regarding the detection of hot/cold data in FlexFS, please see our previous study [1].

The value of δ must be chosen so that $OW(w_i) = WB(w_i)$. Thus, it can be expressed as follows:

$$\delta = \begin{cases} \frac{WB(w_i) - IW(w_i)}{IW(w_i) \cdot (1+\mu)} & \text{if } WB(w_i) > IW(w_i) \\ 0 & \text{otherwise.} \end{cases} \tag{13}$$

In Eq. (13), δ is decided at the beginning of w_i when the exact value of $IW(w_i)$ is unknown. According to our previous study [12], the future write traffic is accurately estimated by monitoring input write traffic for a relatively long time (e.g., 30 minutes or several hours). For this reason, we set T_w to 30 minutes and estimate $IW(w_i)$ to be the moving average of the past four time windows. The new value of δ is used if $\delta < \alpha$. Thus, the write performance is throttled only when a preset lifetime is unlikely to be achieved.

Note that FlexFS cannot guarantee a preset storage lifetime in some special cases. For example, if write traffic is so heavy (i.e., $IW(w_i)$ is always larger than $WB(w_i)$), it is hard to guarantee the lifetime L_{min}. In that case, FlexFS writes all the incoming data to the MLC region, limiting the value of δ to 0.0. Thus, FlexFS can provide a lifetime close to that of JFFS2.

4 Experimental Results

We implemented FlexFS in a custom FPGA-based flash storage prototype and then conducted performance and lifetime evaluations, using various application usage scenarios. To validate the feasibility of FlexFS with long-term scenarios, we also conducted a simulation study, using a trace-driven simulator with the I/O traces collected for a long time. We first present our performance evaluation results in Section 4.1, and then show our lifetime evaluation results in Section 4.2.[5]

4.1 Performance Evaluation

4.1.1 Evaluation with Linux Implementation

FlexFS was implemented in the Linux 2.6.25 kernel and was evaluated in a custom FPGA-based flash storage prototype called BlueSSD [13]. Our flash storage prototype was equipped with a flash array board, which was composed of 32 flash chips. Each flash chip was Micron's 1 GB MLC NAND flash memory [7]. The page size was 4 KB and there were 128 pages in a block.

We evaluated FlexFS in our storage prototype by replaying I/O traces gathered from a variety of systems with six different scenarios, including a Qtopia-based mobile phone [14], laptops, and desktop PCs. A detailed description of the respective I/O traces is summarized in Table 2. All the I/O traces were a set of I/O system calls sent to a file system, such as fopen(), fread(), and fwrite(). A strace tool was used for collecting system-call traces from a Qtopia mobile phone, which was based on a Linux operating system. To gather I/O activities from Windows XP-based laptops and desktop PCs, we used a Process Monitor tool offered by Microsoft Corp. The collected I/O traces were replayed on our flash storage prototype board as if actual applications were running on top of a file system. This I/O trace playback not only helped us to repeat the same I/O traffic under various file system configurations, but also allowed us to quickly evaluate a variety of applications (which run on different platforms such as Qtopia or Windows) without much effort to port them to our storage platform.

TABLE 2
Descriptions of Benchmark Programs

Trace	Description	Length (min)	Idle (%)
`Qtopia`	This trace was collected from a Qtopia-based mobile phone. Several mobile apps, an SMS, an address book, a memo, games, an MP3 player, and a camera, were used.	30	98%
`Office`	This trace was collected while editing several document files, e.g., PPT, DOC, XLS, using Microsoft Office programs.	60	95%
`Web`	This trace was collected while navigating web sites, using internet web browsers.	30	97%
`Devel`	This trace was collected while developing software. I/O activities involved in coding, compiling, and debugging were collected.	60	96%
`General(H)`	These traces captured I/O activities of personal computer users. A heavy user trace is denoted by `General(H)`. A light user trace is denoted by `General(L)`.	30	90%
`General(L)`		120	98%

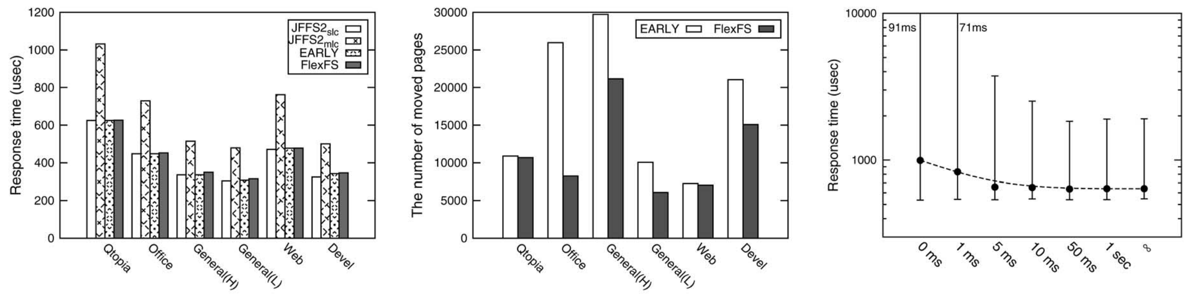
For fair comparisons, we performed our evaluations with the following file system configurations: $JFFS2_{\text{SLC}}$, $JFFS2_{\text{MLC}}$, *EARLY*, and *FlexFS*. $JFFS2_{\text{SLC}}$ was a JFFS2 file system that used SLC-mode programming, and $JFFS2_{\text{MLC}}$ was JFFS2 that used MLC-mode programming. *EARLY* was the FlexFS file system with early free-space reclamation. *FlexFS* was the proposed FlexFS with both dynamic free-space management and dynamic lifetime management techniques. The total storage capacity of $JFFS2_{\text{SLC}}$ was 16 GB. For $JFFS2_{\text{MLC}}$, *EARLY*, and *FlexFS*, the total capacity was 32 GB. The number of P/E cycles for each block was set to 3 K. A threshold value for triggering free-space reclamation was set to 50 ms.

The space utilization of a storage device has a great effect on the performance of FlexFS. According to Agrawal et al.'s study [15] that analyzed the storage space usage of 60,000 personal computers, the average fullness was about 41%. In our evaluation, therefore, 59% of the total storage space (i.e., 18.8 GB) was set to free space. In Section 4.1.2, we evaluate the performance of FlexFS in detail when the storage space is nearly full.

Due to a relatively short length of I/O system-call traces (e.g., 30-120 minutes), it was hard to decide the meaningful value of F_{spare} in our Linux implementation study. Therefore, for respective traces, the values of F_{spare} were set to the largest free space which was obtained by an offline analysis of system-call traces under the assumption that F_{spare} was properly decided by monitoring I/O traffic for a long time. Instead, in Section 4.1.2, we investigate how well FlexFS decides the value of F_{spare} in detail using a trace-driven simulator with long-term I/O traces.

Write Performance: Fig. 11(a) shows the average write response time of four file system configurations. $JFFS2_{\text{SLC}}$ exhibits the best write performance because all the requested data are written to the SLC region. Conversely, $JFFS2_{\text{MLC}}$ shows the worst performance among all the configurations. *EARLY* achieves the same performance as $JFFS2_{\text{SLC}}$. As shown in Table 2, idle periods account for 90%-98% of the total trace execution time across all the traces. Thus, *EARLY* creates sufficient free space for SLC-mode programming by moving all the data written in the SLC region to the MLC region during idle periods. Similarly, *FlexFS* also offers the performance close to $JFFS2_{\text{SLC}}$ by exploiting plenty of idle

5. We present additional experimental results in Appendix (available on-line in the Computer Society Digital Library at https://doi.ieeecomputersociety.org/10.1109/TC.2013.120) which illustrate the changes of major parameters used in FlexFS according to the characteristics of a workload.



(a) Average write response time
(b) Number of move pages during free-space reclamation
(c) Max/avg/min response times with varying threshold values

Fig. 11. Performance evaluation results with Linux implementation.

time. In our observation, the storage lifetime is not a problem in our current setting (i.e., the storage capacity of 32 GB with 3 K P/E cycles), so the performance degradation for the lifetime guarantee is not observed. We perform more detailed analysis on the storage lifetime with various storage device settings in Section 4.2.

The write performance is somewhat different depending on I/O traces. This is mainly due to the effect of a write buffer employed in both JFFS2 and FlexFS. If small-size data (e.g., smaller than 4 KB) are requested to the file system, those data are temporarily stored in a write buffer, avoiding actual writes to flash memory. Thus, the overall write latency is reduced accordingly. On the other hand, the benefit of using a write buffer is not observed in some I/O traces like `Qtopia`, which usually write a bulk of data to flash memory. Regardless of the effect of a write buffer, the performance benefit of *FlexFS* is maintained.

Free-Space Reclamation Overhead: We evaluated the migration overhead caused by free-space reclamation. For this purpose, we analyzed the amount of data that were moved from the SLC region to the MLC region for two configurations, *EARLY* and *FlexFS*, which used different free-space reclamation policies.

Fig. 11(b) shows the amount of data moved by free-space reclamation. *FlexFS* moves a smaller amount of data to the MLC region in comparison with *EARLY* because it delays free-space reclamation as long as possible so that many data are to be invalidated in the SLC region. For `Office`, `General` (H), `General` (L), and `Devel`, the reclamation overheads are reduced by 69%, 29%, 39%, and 28% over *EARLY*, respectively. In those traces, many files are removed or updated while they stay in the SLC region. In cases of `Web` and `Qtopia`, an update on the previously written data or the deletion of the existing files is rarely observed, so the benefit of delayed free-space reclamation is limited to 2-4%.

By eliminating useless page migrations, *FlexFS* also reduces the number of P/E cycles performed on flash memory. For `Office`, `General` (H), `General` (L), and `Devel` traces, the number of P/E cycles is reduced by 28%, 12%, 15%, and 11% over *EARLY*, respectively. On the other hand, for `Web`, only the reduction of 4% is obtained, and no reduction on P/E cycles is observed in `Qtopia`.

Effect of Free-Space Reclamation on I/O Latency: To understand the effect of free-space reclamation on performance, we measured the write latencies of FlexFS while changing a threshold value from 0 to ∞ m sec. If a threshold value was 0, FlexFS triggers free-space reclamation whenever there were data in the SLC region regardless of the existence of idle time. Conversely, if a threshold value was ∞, free-space reclamation was never triggered. We executed a workload generator that issued 4 KB writes according to the Pareto distribution, which was usually used to model I/O traffic. To simulate a write-intensive workload, the average inter-arrival time of requests was set to $10\ m\mathrm{sec}$, and the standard deviation was $110\ m\mathrm{sec}$.

Fig. 11(c) shows our evaluation results, which summarizes the maximum, the minimum, and the average I/O response times according to different threshold values. When a threshold value is 0, the average write response time is 1.5 times longer than that without free-space reclamation (i.e., ∞). In particular, the maximum response time is increased to 91.7 ms. However, the write response times become similar to those without free-space reclamation when a threshold value exceeds 50 ms (i.e., our default threshold value). Even though our default threshold value is conservatively decided, sufficient free space is created in realistic workloads due to plenty of idle time available in consumer devices. As depicted in Fig. 11(a), FlexFS exhibits the write performance close to $JFFS2_{\mathrm{SLC}}$ because on-demand free-space reclamation never occurs.

4.1.2 Detailed Analysis with a Trace-Driven Simulator

The evaluation with the real file system prototype enables us to assess the performance of FlexFS accurately. However, it is not appropriate to evaluate the feasibility of the dynamic free-space management algorithm that requires a long-term history of I/O traffic to make a decision. For more detailed analysis of our free-space management technique, we conducted a simulation study with a trace-driven simulator using block I/O traces which were collected for a long time.

The trace-driven simulator used for our evaluation modeled the primitive I/O operations of MLC flash memory. It also supported FlexFS-specific features, including SLC-mode/MLC-mode programming, free-space reclamation, and heterogeneous block management. To improve simulation accuracy, we modeled the write buffer mechanism and included software overheads in the simulator. According to our performance comparison study, the performance difference between the simulator and the storage prototype was about 14%, which was accurate enough to show the benefits of FlexFS over other file system configurations.

Block-level I/O traces collected at a block device driver were used as an input for the simulator. Table 3 shows the detailed descriptions of block I/O traces used for our

TABLE 3
Descriptions of Block I/O Traces

Trace	Description	Length (hours)	Idle (%)	Written Data	F_{spare}
Desktop	Software development (Linux 2.6, EXT3)	166	94%	10.7 GB	0.46 GB
Laptop1	General laptop usage (Windows XP, NTFS)	146	98%	7.8 GB	2.54 GB
Laptop2	General laptop usage (Windows XP, FAT32)	122	98%	5.3 GB	1.98 GB
Mobile	Portal multimedia player (Windows XP, FAT32)	4.4	83%	29.9 GB	7.1 GB

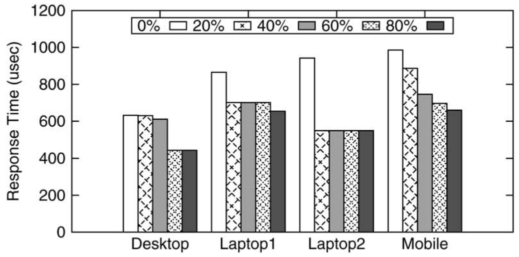


Fig. 12. Write response times with five different training periods, ranging from 0% to 80% of the total trace length.

evaluation. A block-level I/O trace did not include file-system-specific operations, such as file creation and file deletion. However, it provided sufficient information, including the length of idle periods and the amount of write traffic, which were required for our DFM technique to make a decision. The traces were collected using a Diskmon tool for Laptop1, Laptop2, and Mobile and were gathered using a blktrace tool for Desktop.

As shown in Table 3, all the I/O traces exhibited very long idle periods, which facilitated the creation of sufficient free space for SLC-mode programming. F_{spare} was very different according to the characteristics of I/O traffic. For example, Desktop, Laptop1, and Laptop2 required a relatively small amount of free space for busy periods. Considering that many flash devices usually provided several ten gigabytes of a storage capacity or more, the amount of free space that must be reserved for busy periods was not so large. Thus, it is expected that FlexFS achieves performance close to SLC flash memory even when the storage utilization is relatively high. However, for the Mobile trace that often writes many large-size multimedia files, a large amount of free space needs to be maintained in flash memory.

First of all, we evaluated how the value of F_{spare} is properly decided. As pointed out in Section 3.2.1, FlexFS uses the largest amount of free space (which was required by previous busy periods) as F_{spare}, expecting that there will be no busy periods requiring more free space than F_{spare}. To demonstrate the validity of this assumption, we assessed the write performance of FlexFS while varying the length of a training period, which indicates the length of the time until available free space is nearly exhausted and FlexFS starts free-space reclamation to maintain free space F_{spare}.

As depicted in Fig. 12, five different training periods ranging from 0% to 80% of the total length of the I/O trace are used for the evaluation. For example, in the case of Desktop, the length of the 20% training period is 33.2 hours. After 33.2 hours, free-space reclamation is started with F_{spare}, which is the largest amount of free space observed during a

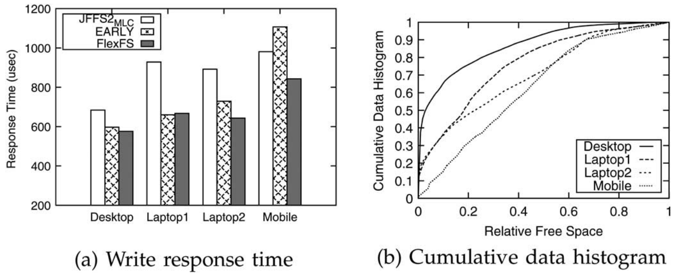


Fig. 13. Evaluation results when the space utilization of a flash device is very high.

training period. As expected, the write performance of FlexFS is improved greatly as the length of a training period increases. In particular, the performances of Desktop and Laptop2 become the same as that of SLC flash memory when the percentages of training periods are higher than 60% and 20%, respectively. Laptop1 and Mobile exhibit longer write response times than that of SLC flash memory even with the 80% training period. However, they exhibit 41% and 48% higher performance than MLC flash memory, respectively.

We then evaluated how FlexFS works when the storage space utilization is very high; that is, potential free space F_{pot} is smaller than F_{spare}. To simulate such a situation, the amount of available free space was initially set to F_{spare}, which was previously obtained by analyzing the I/O trace. After the execution of the I/O trace, flash memory was completely filled with user data, and only MLC blocks remained in flash memory. The empirical distribution of I/O traffic (obtained at offline) was initially given to FlexFS, which was exploited to decide the amount of data to be written to the SLC or MLC region.

We evaluated three different file system configurations: $JFFS2_{MLC}$, *EARLY*, and *FlexFS*. $JFFS2_{MLC}$ wrote all the incoming data to the MLC region, and thus the overhead caused by on-demand free-space reclamation never occurred. *EARLY* wrote requested data to the SLC region, regardless of the remaining space in a storage device. *FlexFS* was the proposed FlexFS that decided the amount of data to be written to the SLC or MLC region by exploiting the CDH of required free space. Fig. 13(a) shows the write response times of three configurations and Fig. 13(b) is the CDHs observed in four different I/O traces.

As shown in Fig. 13(b), in Desktop and Laptop1, a lot of busy periods require small free space for writing data. On the other hand, in Mobile, busy periods requiring large free space are frequently observed because large multimedia files are often written. In Laptop2, busy periods that require large free space or small free space are frequently observed. By adaptively distributing incoming data to the SLC or MLC region according to the characteristic of I/O traffic, *FlexFS* outperforms *EARLY* and $JFFS2_{MLC}$ by 13% and 38%, respectively. *EARLY* exhibits good performance in Desktop and Laptop1, but its performance greatly deteriorates in Mobile because of free-space reclamation overheads caused by the excessive use of SLC-mode programming.

In summary, *FlexFS* improves write performance by 28%, on average, over $JFFS2_{MLC}$. Note that this performance improvement, 28%, is achieved when a storage device is nearly full. Thus, it shows the write performance that FlexFS

TABLE 4
Evaluation Results with or without DLM

	P/E cycles (avg)	Write response time (avg)
without DLM	47.2	468 μsec
with DLM	37.3	711 μsec

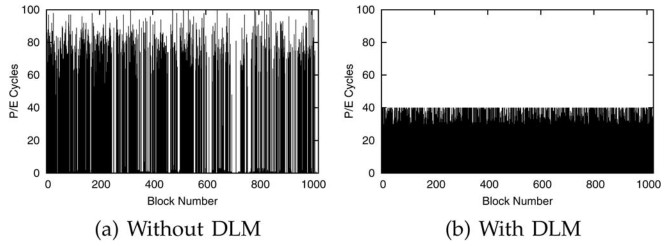


Fig. 14. Distribution of P/E cycles with or without DLM.

can achieve in the worst-case scenario. Considering that the amount of free space is usually sufficient enough for SLC buffering [15], it is expected that FlexFS yields much better write performance in most cases.

4.2 Lifetime Evaluation

We evaluated the proposed dynamic lifetime management (DLM) technique. For our evaluation, the number of P/E cycles allowed to each block was set to 10 K and the storage lifetime was set to 3 years. Since it is impossible to perform an evaluation for 3 years, we scaled down the number of P/E cycles and the target lifetime to 40 and 4 days, respectively. The capacity of a storage device was 512 MB and the I/O traces listed in Table 2 were replayed for 4 days.

Table 4 shows the average P/E cycles and the average write response time when the DLM technique is used or not. Without DLM, FlexFS writes all the requested data to the SLC region, so it exhibits the performance close to that of SLC flash memory. However, the average number of P/E cycles exceeds 40 cycles because of its high data migration requirement. Thus, the required storage lifetime cannot be ensured. When DLM is used, FlexFS writes part of the incoming data to the MLC region so that the average number of P/E cycles becomes smaller than 40. Even though this reduces the overall write performance, the required storage lifetime can be guaranteed. Fig. 14 shows the distribution of P/E cycles on all blocks with or without DLM. The maximum number of P/E cycles with DLM is limited to less than or equal to 40.

Finally, we analyzed the expected storage lifetime while varying some important parameters that affect the lifetime of flash-based storage devices, so as to understand the effect of FlexFS on lifetime in various storage configurations. As noted in Eqs. (9) and (10), the lifetime of flash devices is dependent upon a storage capacity, the number of P/E cycles allowed for a block, and the amount of data written by a workload. For this reason, our analysis was conducted with different storage capacities, 8 GB (low) and 32 GB (high), and with different P/E cycles, 1 K, 3 K, and 5 K. The required storage lifetime was assumed to be 3 years.

Fig. 15 shows our analysis results using the I/O traces `General (H)` and `Office` collected from desktop applications. We also compared the expected storage lifetimes of *EARLY* and *FlexFS*. In general, the 3-year lifetime can be guaranteed when the capacity of flash memory is large (e.g., 32 GB) and

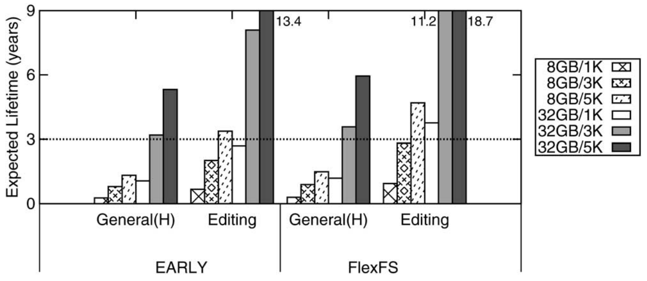


Fig. 15. Expected lifetimes with various storage settings.

the number of P/E cycles is high enough (e.g., 3 K-5 K). However, the specified storage lifetime cannot be ensured when the storage capacity is small (e.g., 8 GB) or the number of P/E cycles is relatively low. Moreover, when write traffic is heavy (e.g., `General (H)`), the expected storage lifetime is further reduced. In those cases, it is inevitable that some of the requested data are written to the MLC region directly even though this causes performance degradation. *FlexFS* achieves longer storage lifetime over *EARLY* by reducing useless free-space reclamation. Therefore, it is expected that *FlexFS* requires less write performance throttling, providing higher performance than *EARLY*.

As depicted in Fig. 15, when the capacity of a storage device is 32 GB, the expected lifetime is usually longer than our target storage lifetime (i.e., 3 years). Considering that recent consumer devices employ a storage device larger than 32 GB for mobile phones and 128 GB for laptops/desktop PCs, it is expected that a storage lifetime would not be a serious problem in real-world products. Therefore, it may be safely assumed that a storage lifetime problem would be mitigated without significant performance degradation. Even if a storage lifetime is limited by heavy write traffic with small P/E cycles, FlexFS helps to provide a reasonable lifetime by regulating the amount of write traffic sent to the SLC region.

5 Related Work

There have been several efforts to combine both SLC and MLC flash memory. Chang et al. suggest a solid-state disk which is composed of a single SLC flash chip and many MLC flash chips [18], while Park et al. and Im et al. present a flash translation layer for SLC-MLC combined storage devices [19], [20]. The basic idea of these approaches is to store small and frequently updated data in a small SLC flash chip while using large MLC flash chips for storing bulk data. This improves the I/O performance for small writes greatly, while providing a large storage capacity with relatively low-cost per byte. In these approaches, however, it is difficult to provide performance close to pure SLC flash memory because they only use small SLC flash memory to achieve a cost benefit. For example, when a large amount of data are issued for writing to a storage device, the I/O performance is limited to that of MLC flash memory because all of the requested data must be sent to MLC flash chips due to the limited capacity of an SLC flash chip. FlexFS can handle this case efficiently by flexibly increasing the size of the SLC region. Furthermore, FlexFS offers a more cost-effective storage solution to end-users over existing approaches because it is based on pure MLC NAND flash memory. Consequently, a more efficient storage device, in terms of performance and capacity, can be realized with FlexFS, in comparison with existing SLC/MLC hybrid approaches.

6 Conclusions

FlexFS is a file system that takes advantage of the flexible programming of MLC flash memory. The novel features of FlexFS are the dynamic free-space and lifetime management techniques, which effectively deal with performance/capacity/lifetime issues raised by the use of flexible programming at the file-system level. Experimental results show that FlexFS achieves the performance of SLC flash memory and the capacity of MLC flash memory at the same time while providing a reasonable storage lifetime in various mobile and PC workloads.

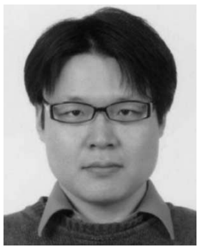
The performance and lifetime of FlexFS can be further improved by exploiting file-system-level information. For example, FlexFS can make a better decision in choosing data for free-space reclamation by taking into account the different characteristics of user data and metadata as well as the types of files. As future work, we are planning to develop an improved version of FlexFS that exploits file-system-level information for more efficient performance and lifetime management.

Acknowledgments

An earlier version of this paper was presented at the USENIX Annual Technical Conference, June 17, 2009 [1]. This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2013R1A6A3A03063762). This work was supported by NRF grant funded by the Ministry of Science, ICT and Future Planning (NRF-2013R1A2A2A01068260). This research was supported by Next-Generation Information Computing Development Program through NRF funded by the Ministry of Science, ICT and Future Planning (No. 2010-0020724). The ICT at Seoul National University and IDEC provided research facilities for this study. Jihong Kim is the corresponding author of this paper.

References

[1] S. Lee, K. Ha, K. Zhang, J. Kim, and J. Kim, "FlexFS: A flexible flash file system for MLC NAND flash memory," in *Proc. USENIX Annu. Tech. Conf.*, Jun. 2009, pp. 115–128.
[2] M. Bauer, "A multilevel-cell 32 MB flash memory," in *Proc. Solid-State Circuits Conf.*, Feb. 1995, pp. 132–133.
[3] P. Pavan, R. Bez, P. Olivo, and E. Zanoni, "Flash memory cells—An overview," *Proc. IEEE*, vol. 85, no. 8, pp. 1248–1271, Aug. 1997.
[4] F. Roohparvar, "Single level cell programming in a multiple level cell non-volatile memory device," U.S. Patent 7,366,013, Apr. 2008.
[5] S.-H. Shin, et al., "A new 3-bit programming algorithm using SLC-to-TLC migration for 8 MB/s high performance TLC NAND flash memory," in *Proc. Symp. VLSI Technol. Circuits*, Jun. 2012, pp. 132–133.
[6] Samsung Corp., "KFXXGH6X4M Flex-OneNAND specification," 2008.
[7] Micron Technology, Inc., "MT29F8G08AAAWP NAND flash memory specification," 2012.
[8] D. Woodhouse. (Jul. 2001). *JFFS: The Journalling Flash File System*, Red Hat Inc. [Online]. Available: http://sources.redhat.com/jffs2/jffs2.pdf.
[9] Aleph One. (2002). *YAFFS: Yet Another Flash File System*, [Online]. Available: http://www.aleph1.co.uk/yaffs.
[10] M. Rosenblum and J. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, Feb. 1992.
[11] SanDisk. (Oct. 2008). "Longterm data endurance (LDE) for client SSD," *SanDisk White Paper* [Online]. Available: http://www.sandisk.com/media/65675/LDE_White_Paper.pdf
[12] S. Lee, T. Kim, K. Kim, and J. Kim, "Lifetime management of flash-based SSDs using recovery-aware dynamic throttling," in *Proc. USENIX Conf. File Storage Technol.*, Feb. 2012, pp. 327–340.
[13] S. Lee, K. Fleming, J. Park, K. Ha, A. Caulfield, S. Swanson, Arvind, and J. Kim, "BlueSSD: An open platform for cross-layer experiments for NAND flash-based SSDs," in *Proc. Int. Workshop Archit. Res. Prototyping*, Jun. 2010.
[14] Nokia Corp. (2012). *Qtopia 4.1.2* [Online]. Available: http://qt.nokia.com/
[15] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch, "A five-year study of file-system metadata," in *Proc. USENIX Conf. File Storage Technol.*, Feb. 2007, pp. 31–45.
[16] J. R. Douceur and W. J. Bolosky, "A large-scale study of file system contents," in *Proc. Int. Conf. Measur. Model. Comput. Syst.*, Jun. 1999, pp. 59–70.
[17] H. Huang, W. Hung, and K. G. Shin, "FS2: Dynamic data replication in free disk space for improving disk performance and energy consumption," in *Proc. Symp. Operating Syst. Principles*, Oct. 2005, pp. 263–276.
[18] L. P. Chang, "Hybrid solid-state disks: Combining heterogeneous NAND flash in large SSDs," in *Proc. Conf. Asia South Pacific Des. Autom. (ASP-DAC)*, Mar. 2008, pp. 428–433.
[19] S. Park, J. Park, J. Jeong, J. Kim, and S. Kim, "A mixed flash translation layer structure for SLC-MLC combined flash memory system," in *Proc. Workshop Storage I/O Virtual. Perform. Energy Eval. Dependability (SPEED)*, Feb. 2008.
[20] S. Im and D. Shin, "ComboFTL: Improving performance and lifespan of MLC flash memory using SLC flash buffer," *J. Syst. Archit.*, vol. 56, no. 12, pp. 641–653, Dec. 2010.



Sungjin Lee received the BE degree in electrical engineering from Korea University, Seoul, in 2005, and the MS and PhD degrees in computer science and engineering from the Seoul National University, South Korea, in 2007 and 2013, respectively. He is currently working as a postdoctoral associate with the Computer Science and Artificial Intelligence Laboratory, the Massachusetts Institute of Technology, Cambridge. His research interests include storage systems, operating systems, and embedded software.

Jihong Kim received the BS degree in computer science and statistics from Seoul National University, Seoul, Korea, in 1986, and the MS 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.