Des Autom Embed Syst (2011) 15:191–224
DOI 10.1007/s10617-011-9071-9

FTL algorithms for NAND-type flash memories

Se Jin Kwon · Arun Ranjitkar · Young-Bae Ko · Tae-Sun Chung

Received: 8 October 2010 / Accepted: 18 February 2011 / Published online: 9 March 2011

Abstract Flash memory is being rapidly deployed as data storage for embedded devices such as PDAs, MP3 players, mobile phones and digital cameras due to its low electronic power, non-volatile storage, high performance, physical stability and portability. The most prominent characteristic of flash memory is that prewritten data can only be dynamically updated via the time consuming erase operation. Furthermore, every block in flash memory has a limited program/erase cycle. In order to manage these issues, the flash memory controller can be integrated with a software module called the flash translation layer (FTL). This paper surveys the state-of-art FTL algorithms. The FTL algorithms can be classified by the complexity of the algorithms: basic and advance. Furthermore, they can be classified by their corresponding tasks: performance enhancement and durability enhancement. The FTL algorithms corresponding to each classification are further broken down into various schemes depending on the methods they adopt. This paper also provides the information of hardware features of flash memory for FTL programmers.

Keywords NAND flash memory · Embedded system · FTL

1 Introduction

Flash memory, a version of EEPROM, has characteristics such as random access, low power consumption, resistance to shocks and convenient portability. Furthermore, it retains data even after the power has been turned off. Traditionally, floppy disks, ZIP disks, optical discs,

S.J. Kwon
Computer Engineering, Ajou University, Suwon, South Korea
e-mail: sejin1109@ajou.ac.kr

A. Ranjitkar
Information & Communication, Ajou University, Suwon, South Korea

Y.-B. Ko · T.-S. Chung (✉)
Information & Computer Engineering, Ajou University, Suwon, South Korea
e-mail: tschung@ajou.ac.kr

Table 1 A comparison between hard disk and nand flash-based solid state driver [2]

	2.5″ SATA 3.0 Gbps SSD	2.5″ SATA 3.0 Gbps HDD
Mechanism type	Solid NAND-type flash memory	Magnetic rotating platters
Density	64 GB	80 GB
Weight	73 g	365 g
Performance	Read: 100 MB/s, write: 80 MB/s	Read: 59 MB/s, write: 60 MB/s
Active power consumption	1 W	3.86 W
Operation vibration	20 G (10 2000 Hz)	0.5 G (22 350 Hz)
Shock resistance	1,500 G for 0.5 ms	170 G for 0.5 ms
Operating temperature	0°C to 70°C	5°C to 55°C
Acoustic noise	None	0.3 dB
Endurance	MTBF >2 M hours	MTBF >0.7 M hours

Table 2 Comparison between NOR-type and NAND-type [3, 4]

	NOR-type	NAND-type
Read	150 ns (1 bytes) 14.4 μs (512 bytes)	10.2 ns (1 bytes) 35.9 μs (512 bytes)
Write	211 μs (1 bytes) 3.53 ms (512 bytes)	201 μs (1 bytes) 226 μs (512 bytes)
Erase	1.2s (128 Kbytes)	2 ms (16 Kbytes)

mini-discs, etc., were used as data storage devices. The hard disk is one of the popular devices among others. However, current solid state memories, such as flash memory, are used as data storage devices with large capacity and fast access capability as shown in Table 1. Flash memory is a high reliable semi-conductor device, which does not have any moving parts, like platters in hard disks. Therefore, it can be used in note books or tablet PCs as a substitute to classical storage devices.

There are two types of flash memory: NOR-type and NAND-type. Both memories were first invented by Dr. Fujio Masuoka in 1980 [1]. Seeing the great potential in flash memory, Intel made the first commercial NOR-type flash memory in 1988. The major advantage of NOR-type flash memory lies in its high-speed read operation. It is read and written in the unit of one byte, and is randomly accessed due to parallel interconnection of its individual memory cells. Therefore, NOR-type flash memory is widely used in code storage applications where only read operations are mainly required. On the other hand, NAND-type flash memory (introduced in 1987 by Toshiba) is optimized for mass (data) storage. A general comparison between NOR-type and NAND-type flash memory is shown in Table 2. Many companies such as M-Systems, Sandisk, and Samsung are developing products using NAND-type flash memories such as mini SD cards, memory sticks, iPods, USB flash drives, flash drivers, GPS devices, Bluetooth products, Netbooks and other mobile PCs.

To replace the hard disk, we need to consider two characteristics of flash memory. The most important characteristic is erase-before-write. Unlike the hard disk, the memory unit in flash memory requires erase operation before updating data. Another characteristic is the damage of memory caused by excessive use of memory units. When a certain portion of memory is written and erased several times exceeding the program/erase cycle limit, that portion of memory can be damaged and the data integrity cannot be guaranteed. Considering these characteristics, flash memory is supported by a software layer called Flash Translation Layer (FTL) for efficient usage of flash memory.

This paper surveys the state-of-art FTL algorithms from U.S. patents and various academic journals. Many U.S. patents from M-systems and Sandisk were already covered in E. Gal et al. 2005 [5], since the basic FTL algorithms were proposed from early flash-related-patents. However, E. Gal et al. mainly concentrated their research in U.S. patents, and there have been much development of FTL in academic studies. Another survey by Chung et al. 2009 [6] lists and compares FTL algorithms. The research discusses the different techniques used in FTLs, and compares some of the prominent FTL algorithms by conducting experimental evaluations. The main concern of the research by Chung et al. is that the scope is not diverse, and only performance is taken into the consideration. On the other hand, our work aims at broadening the scope to include performance and durability, to categorize the diverse algorithms according to a new method, and to compare and analyze FTL algorithms via intuitive demonstrations. We are focusing our survey on the FTLs which are related to NAND-type flash memory since both research and market interests are heading toward to NAND-type flash memory. We introduce the basic algorithms of early U.S. patents, and chronologically categorize complex and combined algorithms discussed in various academic papers.

The remainder of this paper is organized as follows. Section 2 introduces important features of NAND-type flash memory for FTL programmers. Section 2.3 explains the overall role of FTL. Section 3 classifies various FTL algorithms according to their methods, and describes each of them in detail. Finally Sect. 4 concludes this paper.

2 NAND-type flash memory

NAND-type flash memory is a new type of flash memory which is characterized by high density data storage, small chip size, and low cost-per-bit memory. It requires an intermediate software layer called FTL for managing and controlling data. With skillful management of FTL, the overall performance of flash memory can be enhanced. However, in order to understand and develop FTLs, FTL programmers must recognize the hardware organization and characteristics of flash memory. Some of the prominent features of NAND-type flash memory are discussed below.

2.1 Basic features in NAND-type flash memory

A NAND-type flash memory chip consists of a fixed number of blocks, and each block is composed of thirty-two to sixty-four pages depending on the product. A page refers to the unit of read and write operation, and a block, which consists of fixed number of pages, is referred to as the unit of the erase operation. The amount of time taken for each read/write/erase operation can differ depending on the product; however, the erase operation is the most time-consuming operation. For example, one read operation in Samsung Electronics' NAND-type flash memory [4] requires 15 μs, and one write operation requires 200 μs. An erase operation, which is a more protracted operation, takes 2 ms.

There are small block and large block NAND-type flash memories depending on the size of page and block. The small block flash memories' page size is identical to the file system's data sector size, but current large block flash memories such as SSDs and fusion memories contain pages that are four to eight times larger than file system's data sector. Each sector within a page is composed of 512 bytes of data area and 16 bytes of spare area. The spare area is an additional part of a sector that is used to indicate the meta data of each sector or block. The meta data includes the error correction code (ECC), the validity of each sector's

Table 3 Comparison between SLC and MLC [7, 8]

	SLC	MLC
Page size (data area)	2 Kbytes	4 Kbytes
Page size (spare area)	64 bytes	128 bytes
Block size	64 pages	128 pages
Read	129 μs	165 μs
Write	298 μs	950 μs
Erase	1.5 ms	1.9 ms
Erase/program cycle limit	1,000,000	10,000

data, the statement of block, the number of erase operations performed on the corresponding block, and any other information used by the FTL algorithm. Therefore, in case of small block flash memory, a page would contain 512 bytes of data area and 16 bytes of spare area, since the size of a page is one sector. On the other hand, in case of large block flash memory, a page with four sectors would consist of 2 Kbytes of data area and 64 bytes of spare area.

NAND-type flash memory provides partial programming for each page, since it restricts to update data without having an erase operation (erase-before-write) as mentioned in Sect. 1. The partial programming is additional byte-unit-programming before having an erase operation. It enables to re-access the written page and write data within the portion that has not been written yet. Thus, FTL designers can indicate the meta data in the spare area even after the write operation had been performed on the data area. The number of partial programming (NOP) for data area and spare area varies depending on the small block and large block flash memory. In case of small block flash memory, the NOP for the data area is usually one to two and the NOP for spare area is three to four. On the other hand, the NOP for data and spare area in the large block flash memory is only one or two.

2.2 Features in large block flash memory

Beside the basic features of NAND-type flash memory, large block flash memory has some unique features which require to be informed to the FTL programmers. Currently many FTLs are based on small block flash memory; therefore, they are modified by considering the features of large block flash memory discussed below.

2.2.1 MLC/SLC

The large block flash memory can be categorized into single-level cell (SLC) flash memory and multi-level cell (MLC) flash memory. Each cell in SLC can represent only one bit. On the other hand, each cell in MLC can represent more than one bit, since the voltage level of a single cell in MLC has been divided into four or more levels [9]. This MLC technology allows the semiconductor manufacturers to produce higher capacity with lower cost. For example, in 2 bit-MLC, each page consists of eight sectors and each block consists of 128 pages, while the size of page and block in SLC is four sectors per page and 64 pages per block. However, as a trade-off to the capacity, there are few additional limitations in MLC as shown in Table 3. In MLC, the cost of read and write operation have been increased, and the bit error rate has increased by two orders of magnitude due to the reduced distance between adjacent voltage levels [9]. Furthermore, the program/erase cycle limit has been reduced. The program/erase cycle limit refers to the maximum number of writes/erases allowed for each block. If a block is above the threshold, the block may not function correctly thus causing data loss.

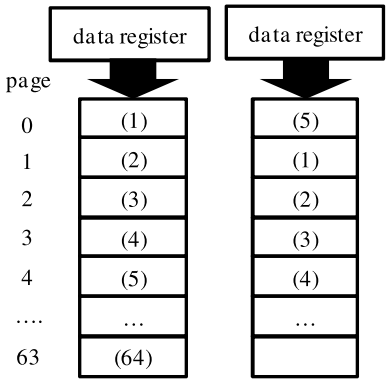

Fig. 1 Sequential write restriction

2.2.2 Sequential write restriction and NOP restriction

The small block flash memory does not have any restriction of writing data within a block. On the other hand, the large block flash memory requires the pages within a block to be written from least significant bit (LSB) page to most significant bit (MSB) page, to avoid program-disturb write errors [10]. Figure 1 shows an example. We assume page 0 and page 63 are LSB page and MSB page, respectively. The left side of the figure shows the sequential write operations, and the right side of the figure shows the random write operations which are prohibited in the large block flash memory. The left side of the figure is allowed in the large block flash memory, since the write operations are performed sequentially from page 0 to 63. On the other hand, the right side of the figure is prohibited, since the page 0 is written after page 4.

Another significant restriction of the large block flash memory is the NOP restriction. Even though a page consists of multiple sectors, only one NOP is given in data and spare area [8]. Therefore, this NOP restriction does not allow to re-access the page to indicate additional information.

2.2.3 Random data out/in

The large block flash memory provides random data out (in the reading case) and random data in (in the writing case), since the size of a page is composed of more than one sector. When the large block flash memory reads data, it may output random data in a page instead of the consecutive sequential read [7]. The read operation is performed in large block flash memory according to the following steps. We assumed the size of a page is 2 Kbytes for the following example.

1. Write the read command to the command register along with an address from which data is read.
2. Transfer the 2 Kbytes of data within the selected page to the data register.
3. Read out data in the data register which is 30 ns cycle time per byte [7].

Thus, the time for the read operation is calculated as follows. We assume that the times required in each step above are T_c, T_d and To, respectively.

$$T_{read} = T_c + T_d + 2\ \text{Kbytes} \times T_o \tag{1}$$

However, if we use the random out command, the first and second steps in the above are the same, and the third step is changed as follows.

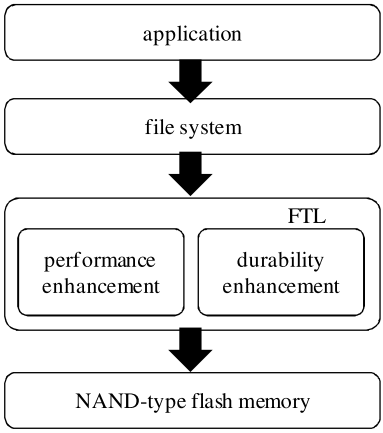

Fig. 2 Organization of NAND-type flash memory system

1. Write the random data output command to the command register along with a column address.
2. Read out data from the data register in 30 ns per byte.

If we assume that the time required for writing the random data out command to the command register (step 1) is T_{ro} and that the size of random out data is c, then the time for the random out operation is calculated as follows.

$$T_{random\_out} = T_c + T_d + T_{ro} + c \times T_o \tag{2}$$

Compared to T_{read} in (1), T_{random_out} has a burden of T_{ro}, but T_{ro} is practically very small. Therefore, since T_{ro} is negligible and the constant c is usually 512 bytes (one sector), the random out operation may increase performance by about four times compared to that of the general 2 Kbytes read operation. On the other hand, if a whole page is being read at once, then it is appropriate to use the read operation without the random out command. In the writing case, data loading time can be decreased by using the random data in operation. Since its procedures are similar to the reading case, we omit in this paper.

2.3 The flash translation layer

Among the total capacity of NAND-type flash memory, less than four percent of memory is hidden from the consumers for implementing the FTL algorithm [6]. This hidden capacity includes the spare area and some spare blocks. The spare blocks are additional empty blocks given to the FTL programmers to use accordingly. Thus, the number of spare blocks needed for the FTL algorithm is different, and the performance evaluation can also be dramatically affected on the number of spare blocks.

The methods of utilizing the spare blocks for enhancing the performance and durability of flash memory is decided by a software layer called FTL. The basic role of FTL is explained by the general organization of NAND-type flash memory system, as is shown in Fig. 2. The host system views the flash memory as a hard disk-like block device, and thus the file system issues read or write commands with logical addresses to read from, or to write data to, specific addresses of flash memory. The physical addresses corresponding to the logical addresses are determined by a mapping algorithm of FTL. During the address translation, FTL looks up the address mapping table. The mapping table is stored in SRAM, a fast but

high-priced memory, which is used for mapping logical addresses to physical addresses in the unit of sector or block. When issuing overwrites, FTL redirects the physical address to an empty location (free space), thus avoiding the erase operations. After managing an overwrite, FTL changes the address mapping information in SRAM. The outdated block can be erased later. The techniques of adding functions for flash memory [11–13] in the file system have been used in the past. However, for compatibility with the existing file system, the tendency is moving in the direction of having a device driver as a separate layer.

FTL carries out performance and durability enhancement in addition to the basic address translation. The performance enhancement refers to the issues of reducing the number of operations in read, write and erase. Among three operations, reducing the number of erase operations is the most critical issue since the cost of an erase operation is very high compared to that of the read or write operation. Durability enhancement refers to erasing every physical block as evenly as possible without causing performance degradation. If a block is above the program/erase cycle limit, the block may not function correctly thus causing data loss.

3 FTL algorithms

The FTL algorithm has basically three components coordinating with each other. They are basic mapping, performance enhancement and durability enhancement. In Sect. 3, we will introduce and analyze the diverse algorithms for each component, and we organize them according to a new dimension. Implementing each FTL algorithm can be slightly different depending on the small block and large block flash memory; however, the main idea of the FTL algorithms remains same. For the sake of simplicity, we will explain the FTL algorithms based on the characteristics on small block flash memory.

3.1 Basic mapping algorithms

The mapping technique is the process of mapping the physical blocks to the logical blocks for achieving easy and efficient data access to the block. In this section, we will discuss and evaluate some basic mapping techniques.

3.1.1 The sector mapping algorithm

The sector mapping algorithm is a naive and intuitive FTL algorithm [14]. In the sector mapping, every logical sector is mapped to a corresponding physical sector. Therefore, if there are n logical sectors seen by the file system, then the row size of the logical to physical mapping table is n. Figure 3 shows an example of sector mapping. In this example, we assume that a block is composed of four pages, where the size of each page is one sector. If we assume that there are 16 logical sectors, then the row size of the mapping table is 16.

When the file system issues a command “w 9 A: write data A to logical sector number (LSN) 9”, the FTL writes the data to physical sector number (PSN) 9 according to the mapping table. When another write operation occurs in PSN 9, an erase operation is inevitable due to erase-before-write, since the PSN 9 is already occupied with the data “A”. In this case, FTL determines the location of an empty sector, writes data, and adjusts the mapping table. If an empty sector does not exist, then FTL will select a victim block from flash memory, copy the valid data to a spare free block (empty block), and update the mapping table. Finally, it will erase the victim block and reserve as a spare block for future use.

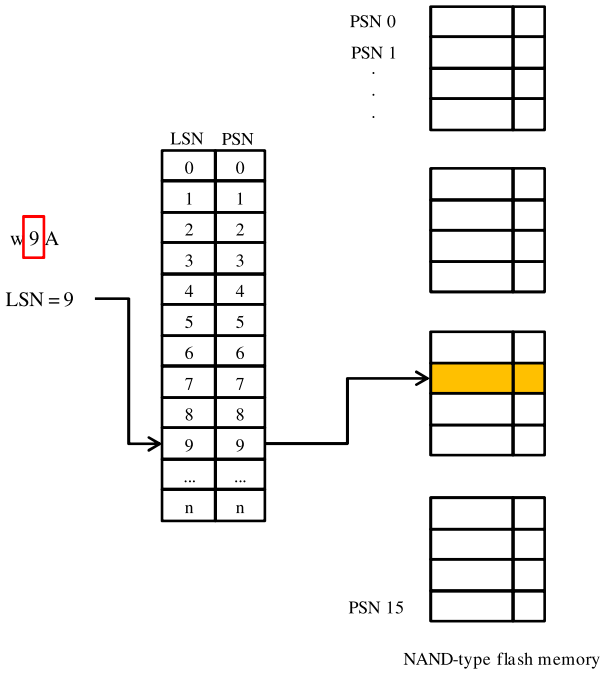

Fig. 3 Sector mapping algorithm

3.1.2 The block mapping algorithm

As the sector mapping algorithm requires a large mount of RAM, it is hardly suitable for small embedded systems. To overcome this memory issue, Takayuki Shinohara [16] proposed the block mapping algorithm. The basic idea of block mapping is to contain the mapping table in the unit of block.

In the block mapping algorithm, the row size of the logical to physical mapping table corresponds to the number of blocks in flash memory. Figure 4 shows an example of the block mapping algorithm. We assume the row size of the mapping table is four. When the file system issues the command "w 9 A", the logical block number (LBN) is first calculated by dividing LSN via the number of sectors per block. The corresponding physical block number (PBN) is retrieved from the block mapping table. Next, the offset of the retrieved physical block is determined by the remainder of division. In Fig. 4, LBN 2 is calculated by 9/4, thus PBN 2 is retrieved from the block mapping table. The write operation is performed on the second sector of PBN 2 since the remainder is 1 (9%4).

Many performance enhancing algorithms use the block mapping algorithm as their basic mapping technique, since it requires small size mapping information [17]. However, if the file system issues many write commands with identical LSNs, then this will lead to severe performance degradation due to the write and erase operations.

3.1.3 The hybrid mapping algorithm

The hybrid mapping is introduced due to the disadvantages of both sector and block mapping algorithms mentioned in the previous two subsections [15]. The hybrid mapping, as its name suggests, uses both block mapping and sector mapping. First, it uses block mapping

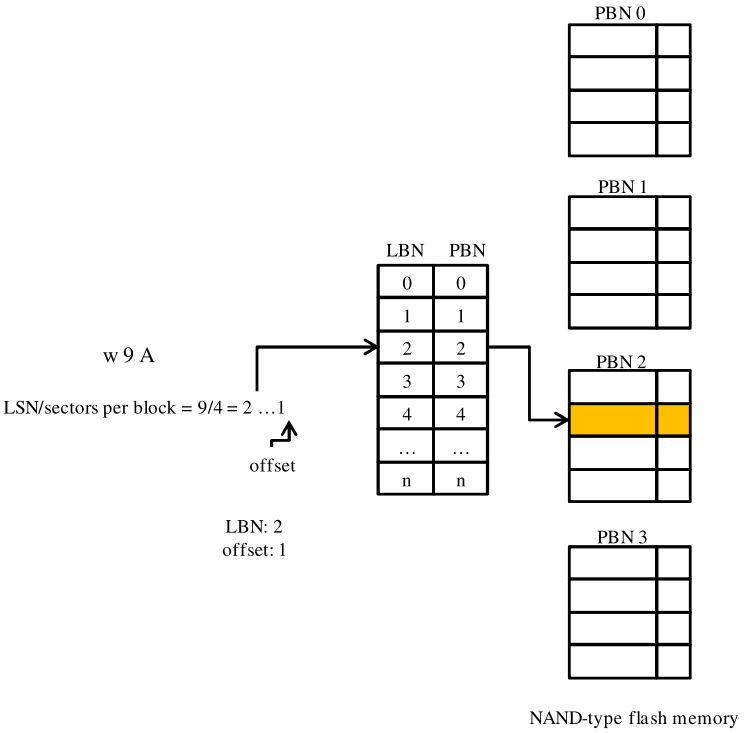

Fig. 4 Block mapping algorithm

to achieve the corresponding physical block, and it then uses a sector mapping to locate an available empty sector within the physical block.

Figure 5 shows an example of the hybrid mapping algorithm. When the file system issues the command, "w 9 A", FTL calculates the LBN 2(9/4) and retrieves PBN 2 from the block mapping table. Next, an empty sector is searched from the corresponding PBN's sector mapping table. In the example, since the first sector of PBN 2 is empty, the data is written to the first sector. Since the data is written in the location where the logical and physical sector offsets (e.g., 1 and 0, respectively) are different, the corresponding logical sector number of data, LSN 9, is indicated to the first sector spare area of PBN 2. The technique of writing data in the same offset as seen in the block mapping algorithm is called in-place, and the technique of determining the location in the different offset, as in the hybrid mapping algorithm, is called out-of-place. Finally, the sector mapping table of the corresponding physical block is updated. In Fig. 5, the physical sector offset 0 corresponds to the logical sector offset 1 of PBN 2's sector mapping table, since the data of LSN 9 is written in the first sector of PBN 2. When reading data from flash memory, FTL locates the physical block number from the block mapping table and finds the corresponding physical sector offset in the corresponding sector mapping table of PBN.

3.2 Evaluations of mapping algorithms

The FTL algorithms are compared in terms of file system-issued read/write performance and memory requirement for storing mapping information. In this section, we will analyze and

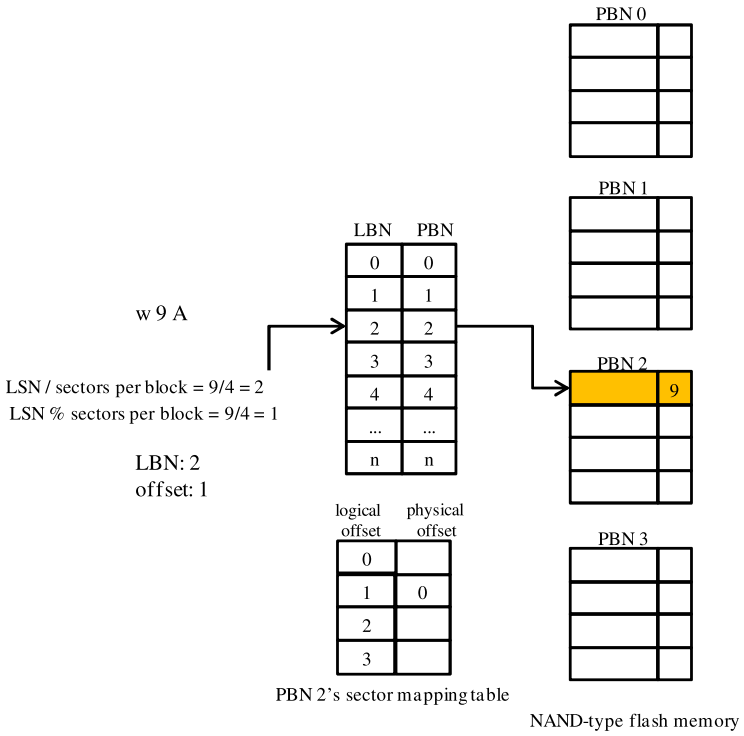

Fig. 5 Hybrid mapping algorithm

compare the basic mapping algorithms as an example, because the performance enhancing and wear-leveling algorithms are based on one of these basic mapping algorithms.

3.2.1 Performance evaluation

The read/write performance of an FTL algorithm can be measured according to the number of flash I/O operations (read, write and erase), as the read/write performance is I/O-bounded. We assumed the mapping table is maintained in SRAM, and the cost of accessing the mapping table in SRAM is always zero. The read and write costs can be computed by the following two equations. Though the following equations are general, they can be a starting point in designing and analyzing FTL algorithms.

$$C_{read} = \chi T_r \tag{3}$$

$$C_{write} = p_i T_w + p_o(\chi T_r + T_w) + p_e(T_e + T_w + T_c) \tag{4}$$

C_{read} and C_{write} denote the costs of read and write commands issued from the file system layer, respectively. The variable χ represents the number of read operations. T_r, T_w and T_e are the costs of read, write, and erase operations processed in the flash memory layer. T_c is the cost of copies needed for transferring the valid data to another free block before erasing. p_i and p_o are the probability for in-place and out-of-place technique, and p_e is the probability that a write command incurs any erase operations.

Table 4 Bytes for addressing

Mapping table	Bytes per entry	Total cost of RAM
Sector mapping table	3 bytes	3 bytes × 32 × 8192 = 768 Kbytes
Block mapping table	2 bytes	2 bytes × 8192 = 16 Kbytes
Hybrid mapping table	(2 + 1) bytes	2 bytes × 8192 + 1 bytes × 32 × 8192 = 272 Kbytes

The variable χ is 1 in the sector and block mapping techniques, as the sector to be read can be directly found from the mapping table. However, in the hybrid mapping algorithm, the value of the variable χ is in the range of $1 \leq \chi \leq n$, where n is the number of sectors within a block. The requested data can be read only after scanning the logical sector numbers stored in the spare areas of a physical block. Thus, the hybrid mapping scheme has higher read cost compared to the sector and the block mapping techniques.

There are three scenarios for the write commands as shown in (4). The first scenario, $p_i T_w$, occurs when the write operation is performed in the in-place location. The second situation, $p_o(\chi T_r + T_w)$, occurs when the write operation should be performed after scanning an empty sector of a block. This case can occur in the hybrid mapping algorithm, and additional read operations might be required. The last scenario, $p_e(T_e + T_w + T_c)$, occurs when the write operation incurs an erase operation. In this case, the basic mapping algorithms perform as follows: valid data is copied to a new spare free block (T_c), and the write operation is performed (T_w). The mapping table is changed accordingly; and finally, the source block is erased (T_e).

Given that T_e and T_c are high cost operations relative to T_r and T_w, the variable p_e is a key point when computing the write cost. In the sector mapping algorithm, the probability of requiring an erase operation per write is relatively low, while in block mapping, conversely, is relatively high.

3.2.2 SRAM calculation

Another measure of comparison is the memory requirement for storing the mapping information. The mapping information should be stored in a persistent storage, and it can be cached in SRAM for achieving better performance. Table 4 shows the memory requirements for the three address mapping techniques. For convenient understanding, we have used 128 Mbytes small block flash memory for calculating the total cost of RAM; however, the method of calculation remains same. 128 Mbytes of the small block flash memory is composed of 8,192 blocks, and each block is composed of 32 sectors [4]. In the sector mapping, three bytes are needed to represent a sector number, whereas in block mapping, only two bytes are needed to represent a block number. Hybrid mapping requires two bytes for representing a block number and one byte for indicating the offset. According to Table 4, it is clear that block mapping requires the smallest amount of SRAM.

3.2.3 Static and dynamic allocation

Each address mapping table of the basic mapping algorithms can be built by static or dynamic allocation. The static allocation maps the physical sectors/blocks to the logical sectors/blocks beforehand. Figures 3, 4, and 5 are examples of basic mapping algorithms using the static allocation.

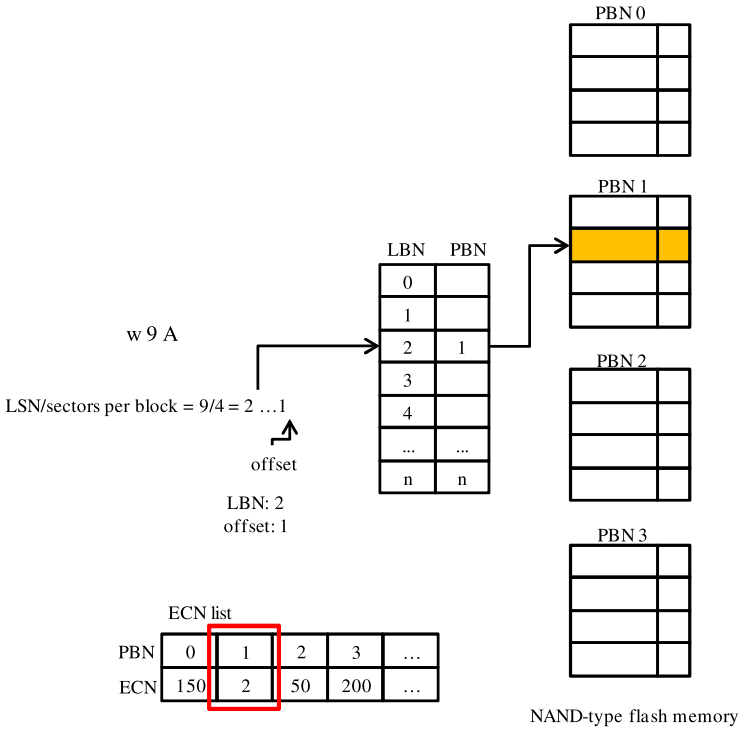

Fig. 6 Dynamic allocation

The dynamic allocation, on the other hand, builds the address mapping table as the write operation performs. When the file system issues a write command, it searches for an empty sector/block to be allocated to LSN/LBN, respectively. An empty sector/block can be chosen differently depending on the FTL algorithms; however, many algorithms use the techniques of selecting the victim block by comparing each block's erase count number (ECN: the number of erase operations performed in a block) or by simply using the round-robin policy. Figure 6 shows an example of dynamic allocation with the block mapping algorithm. FTL calculates LBN from the command "w 9 A", and it checks for existing corresponding PBN from the block mapping table. Since LBN 2 has no corresponding PBN, FTL selects an empty block according to its algorithm. In this figure, we have selected an empty block with the least ECN, PBN 1, by comparison with all the spare blocks. We can keep the records of ECN by having a list in SRAM or by just indicating and reading ECN from the spare sector areas. Thus, data is written to the second sector of PBN 1.

3.3 Performance enhancing algorithms

The unit of mapping algorithm can be defined in term of sectors, pages, and blocks. As the granularity of the mapping table gets finer, the performance increases while the cost of RAM also increases as mentioned in Sects. 3.2.1 and 3.2.2. On the other hand, the coarser granularity gives lower cost of RAM but degraded performance. It is the duty of performance enhancing algorithms to identify the methods to increase the performance while decreasing the cost of RAM. Therefore, the basic mapping algorithms are modified to efficiently utilize

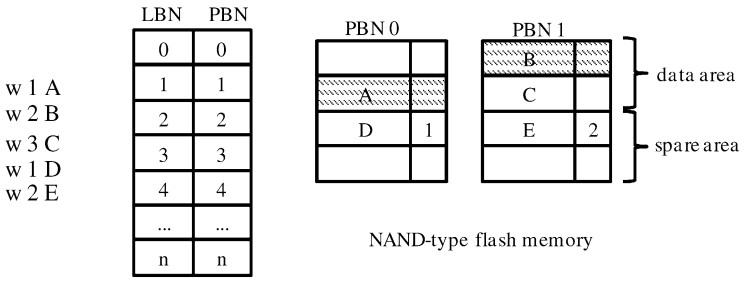

Fig. 7 Mitsubishi

the spare blocks in order to reduce overall number of read/write/erase operations. Previous performance enhancing algorithms include log-sector scheme, log-block scheme and state transition scheme. They are based on the partial programming, basic mapping algorithms, and static/dynamic allocation, which were all discussed in Sect. 3.2. In this section, we classify the performance enhancing algorithms according to new concept, and each FTL algorithm is explained by intuitive demonstration.

3.3.1 The log-sector scheme [16–19]

The main purpose of the log-sector scheme is reducing the number of erase operations occurring from the repetition of identical logical sector numbers. The concept of "log-sector" is to provide one or several sectors as a buffer for each sector. Thus, in the log-sector scheme, updated data will be written to the log-sector instead of immediately erasing the original block and writing the updated data to the new block as in the block mapping algorithm. The log-sectors can be allocated within a different block from the original data or within the same block.

Mitsubishi Mitsubishi [16] is one of the representative algorithms which allocates log-sectors within the same block from the original data. It is based on the block mapping algorithm, in order to have benefits in small SRAM requirement and small overhead in mapping table construction. Mitsubishi proposed "space sectors" as another term for log-sectors. The space sectors are empty sectors that are allocated within a physical block to be used as a buffer for the rest of the sectors. Therefore, one physical block in Mitsubishi is composed of a general sector area and a space sector area.

Figure 7 shows an example of the Mitsubishi algorithm. Figure 7 assumes that a physical block is composed of two general sectors and two space sectors. When the file system issues the first write request, "w 1 A", LBN 0 (1/2) and the logical sector offset 1 (1%2) are calculated and the corresponding PBN is obtained from the block mapping table. As the physical sector offset 1 of PBN 0 is initially empty, the data is written to the second sector of PBN 0. Additional write operations are performed in the same way until "w 1 D". When the corresponding physical sector is already occupied with data, Mitsubishi searches an empty sector starting from the first sector of the space sector area and writes data in that sector. Thus, data "D" is written in the first sector of the space sectors, and "1" is indicated in the spare area for re-calling its logical sector number. When a read operation, "r LSN: read data from LSN", occurs, the most up-to-date can be found by scanning the spare areas from the bottom.

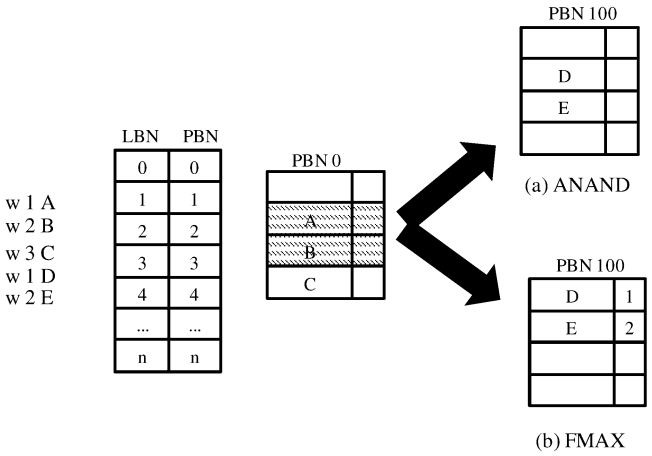

Fig. 8 M-systems

Mitsubishi reorganizes the corresponding physical block when there is no empty sector left in the space sector area. The reorganization process is performed as following. The FTL algorithm obtains a free block and copies valid sectors from the old physical block to the chosen free block. Next, the mapping information is updated. Finally the old physical block is erased.

If the write operations of Fig. 7 were performed in the pure block mapping algorithm, then one erase operation would have required at the point of "w 1 D" and another erase at "w 2 E". On the other hand, Mitsubishi has no erase operations due to the space sectors. The total number of erase operations is reduced as the number of space sectors increases; however, the total capacity of flash memory will be extremely sacrificed, because every physical block is occupied with the same number of space sectors. Therefore, one additional space sector results in capacity reduction of 512 bytes $\times$ number of physical blocks.

M-Systems M-Systems [17] propose FTL algorithms known as ANAND and FMAX. These algorithms are examples of log-sector scheme which allocate log-sectors within a different block from the original data. Their techniques are based on the block mapping algorithm; however, in their scheme, one logical block is mapped to two physical blocks giving the ratio of 1:2. The write operation is similarly performed to the block mapping technique, but when an overwrite is inevitable due to the occurrence of identical logical sector numbers, the data is written to "replacement block".

Figure 8 shows the write operation of M-Systems with an example. When a write operation, "w 1 A", is issued, LBN and logical sector offset are calculated and PBN is retrieved from the block mapping table. However, when the targeted sector is already filled with data, the write operation is performed differently in ANAND and FMAX. In ANAND, the overwrite is written in the replacement block where the logical and physical sector offsets are identical (in-place). In Fig. 8(a), "w 1 D" is written in the replacement block, PBN 100, since the physical sector offset 1 of PBN 0 is already filled with data "A". When the third identical logical sector number ("w 1 F") occurs, valid data of both data block and replacement block are merged into a free block. The free block then becomes a new data block, and old data block and the replacement block are erased. Gathering valid data from more than two blocks into one block is called "merge operation".

The classical problem of ANAND is the requirement of merge operation in every third occurrence of the same logical sector number. In order to reduce the number of merge operations, FMAX writes the data according to the out-of-place technique that enables the utilization of all unused sectors of the replacement block before having a merge operation. Figure 8(b) shows the result of FMAX. "w 1 D" is written in the first sector of PBN 100 since FMAX writes data according to out-of-place in the replacement block. For "w 2 E", data is written in the replacement block below data "D". When there is no empty sector left in the replacement block, the merge operation is performed.

AFTL Wu and Kuo [18] have proposed An Adaptive Two-Level Management for the Flash Translation Layer (AFTL), in which its mapping algorithm is very similar to FMAX. They have used ANAND as their targeting algorithm and pointed out the overhead of inefficient usage of all the sectors within the replacement block. As solution, AFTL writes the updates in the replacement block in out-of-place technique; furthermore, it uses a sector mapping table for the replacement block. As AFTL uses both block mapping and sector mapping algorithms, it can be viewed as the log-sector scheme using the hybrid mapping technique.

RNFTL Wang et al. propose A Reuse-Aware NAND Flash Translation Layer (RNFTL) [19], in which its mapping algorithm is mainly based on FMAX and ANAND. The main motivation of RNFTL comes from the space utilization of merge operation. According to their real-time experiments from various traces, only 42% of the original block is occupied with data when the original block and log-sectors are forced to perform a merge operation. In order to enhance the space utilization, Wang et al. propose to preserve these original blocks from being erased if there are many empty sectors in the original block.

RNFTL contains a preserved block table (dirty block table [19]), which contains PBNs of the preserved blocks. When an original block and its log-sectors are having a merge operation, RNFTL checks whether the space utilization of the original block is below the predetermined threshold "a". In case the space utilization is below "a", the original block's PBN is inserted to the preserved block table, and its empty sectors are reused as the log-sectors for other original blocks. On the other hand, if the space utilization is above "a", the merge operation is performed as same as FMAX and ANAND. In their performance evaluation, they have assumed "a" is set to 20% and every original block contains a block amount of log-sectors. They have significantly reduced overall number of erase operations; however, RNFTL still suffers from the excessive requirement of log-sectors, which is the primary problem of the log-sector scheme. We will discuss this problem in detail in Sect. 3.3.2.

3.3.2 The log-block scheme [20–23]

The main concept of the log-block scheme is to have a fixed percentage of the log-blocks as temporary storage for the write operations of identical LSNs. The log-block scheme reserves a certain portion of flash memory to use as a buffer similar to the replacement blocks of M-Systems. However, in the log-block scheme, data in the buffer is written sequentially from the first sector of every log-block to the bottom. The main difference between the log-block scheme and the log-sector scheme is the capacity they use for the log-sector/log-block area. M-systems' algorithms (ANAND and FMAX) assume that every data block has one dedicated replacement block, whereas the log-block scheme shares few log-blocks within all the data blocks.

In the log-sector scheme, every data block has an additional replacement block as in the M-Systems or replacement sectors as in Mitsubishi. For example, in Mitsubishi, if a block

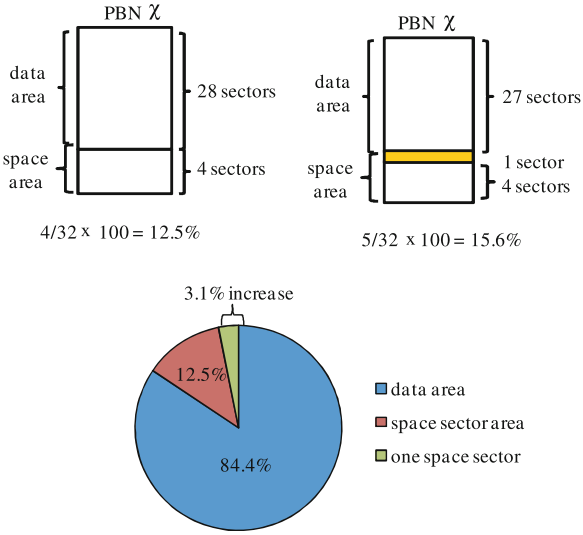

Fig. 9 Space degradation of log-sector scheme

consists of 32 sectors and four sectors are used as a log-sectors, then $4/32 \times 100 = 12.5\%$ space is reserved for log-sectors, as is shown in Fig. 9. Every increment of the log-sector decreases the capacity of data sectors by 3.1%.

The scientific community felt an urge for a technique where the ratio of the log-block to the data block would be as minimum as possible. The log-block scheme provides a solution by utilizing a small amount of log-blocks for the entire flash memory. In the experiment of [21, 22], they have varied the number of log-blocks from two to 64 blocks for 256 Mbytes flash memory. 64 blocks are only 0.3% of entire flash memory. If the log block is doubled to 128 blocks, then the percentage increases to 0.6% which is far less than that of the log-sector scheme.

BAST Kim et al. [20] first proposed a log-block scheme. Their algorithm, BAST, reduces the number of erase operations by writing to temporary storage, log-blocks. BAST is very similar to FMAX in terms of mapping one logical block to two physical blocks. However, the main difference between BAST and FMAX is the number of blocks used as a buffer. Instead of having one block as a buffer for every data block as in FMAX, BAST contains below 0.3% of log-blocks to cover the entire flash memory [21]. Since there are few log-blocks, it is important to efficiently reuse the log-blocks. BAST contains its basic mapping information in a block mapping table, and contains a separate sector mapping table for the log-blocks. Further explanation of BAST is described in Fig. 10.

In Fig. 10, it is assumed that the number of sectors per block is four and the total number of log-blocks is two. When the first write operation, "w 1 A", is executed, PBN 0 is retrieved from the block mapping table since the result of 1/4 is LBN 0. "A" is stored at offset 1 (1%4) of PBN 0. In the case of the third write operation, "w 1 C", BAST finds the designated sector already occupied with data in PBN 0; therefore, BAST writes data at the first sector of the log-block, PBN 100. LSN 1 is indicated in the spare area since it is written in out-of-place. Fourth write operation, "w 2 D", is directed below data "C" since the log-block is filled sequentially. The following write operations will allocate PBN 101 as a log-block for PBN 1 and fill data sequentially.

Let us assume another write command, "w 8 J", has occurred in Fig. 10. In this case, BAST has to select a victim log-block between PBN 100 and PBN 101 for the merge op-

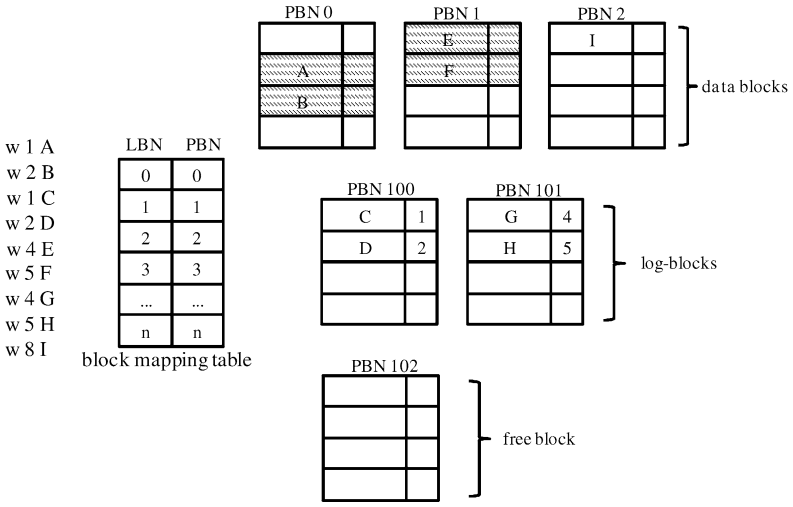

Fig. 10 BAST

eration, since there is no more log-block to be allocated for the data block, PBN 2. The victim can be selected by simply using a round of robin policy or selecting the block with least ECN; however, the method can be diverse depending on the environment and the FTL programmer. When a victim block is selected, BAST copies the up-to-date data from the victim log-block and its associated data block to a free block. Then, BAST updates the free block as a new data block in the block mapping table. Finally, the old data block and victim log-block are erased, and one of the erased blocks is chosen as a new log-block for PBN 2.

As explained above, a merge operation in BAST requires two erase operations and the maximum number of sectors per block of write operations. This merge operation can be optimized if the victim satisfies the following condition: all writes in the log-block are sequential and the total number of written sectors is equal to the number of sectors per block. In this case, instead of having a merge operation, BAST completes the merge operation by simply indicating the victim log block as a new data block, and by having just one erase operation on the old data block. This situation is shown in PBN 101 in Fig. 10. If overwrites of LSN 6 and LSN 7 are performed, then all data of PBN 101 would be filled sequentially; therefore, PBN 101 becomes a new data block for LBN 2 and the old data block, PBN 1, is erased. This optimized merge operation is called the "switch operation".

FAST The log-block scheme presented by Kim et al. was referred to as Block Associative Sector Translation (BAST) by Lee et al. [21]. They suggested the name BAST since the address associativity between logical sectors and log-blocks is of "block-level". Here, the block-level refers to the fact that when an overwrite occurs in a data block, the write can be redirected only by one dedicated log-block; it cannot be redirected to other log-blocks. Lee et al. observed that low space utilization can be caused by block-leveling since the log-block can have a large number of unused sectors while it is forced to have a merge operation. As a solution, Lee et al. propose to have a full associativity between the logical sector and log-blocks. Since it is fully associative, Lee et al. referred to this algorithm as Fully Associative Sector Translation (FAST) scheme.

The log-blocks in FAST are divided into two groups: SW log-block for sequential overwrites and RW log-blocks for random overwrites. Since most workloads are mainly com-

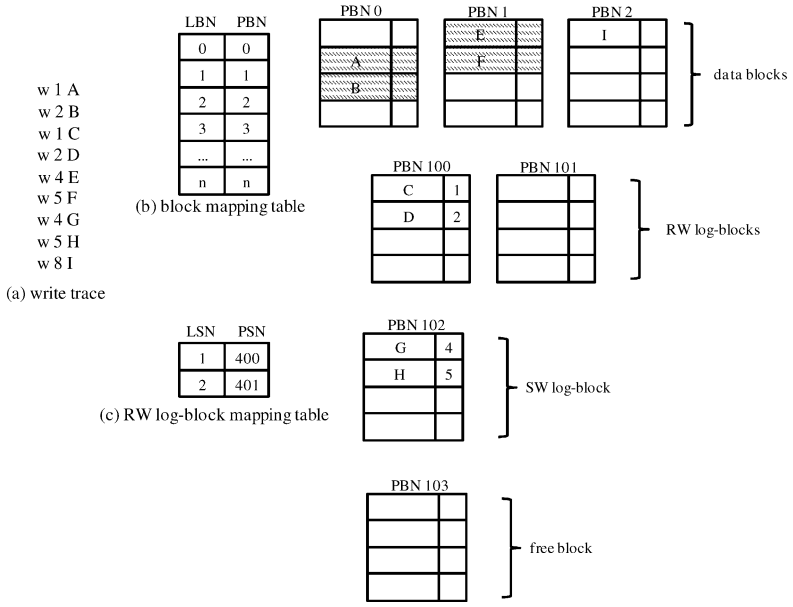

Fig. 11 FAST

posed of a large number of sequential writes, it is likely that many switch operations (initially proposed by BAST) arise from them. Therefore, Lee et al. allocate one SW log-block separately from RW log-block, and they utilize this log-block only for the switch operations. Lee et al. use the rest of log-blocks as their shared buffer. Instead of dedicating a log-block for one data block as in BAST, the RW log-block is fully shared by filling data sequentially from top to bottom. RW log-blocks have their own sector mapping table in RAM. Further description of RW and SW log-block will be explained in detail in Fig. 11.

In Fig. 11, we assume there are two RW log-blocks and one SW log-block. We are explaining with the same pattern of write operations that was used in Fig. 10, BAST. When the write command is requested, FAST finds the corresponding physical sector by calculating LBN (LSN/sectors per block) and the offset (LSN%sectors per block), and it retrieves PBN from the block mapping table. For example, the first two writes: "w 1 A" and "w 2 B" are written in the second and third sectors of PBN 0, respectively, since the quotient of division is 0 (1/4, 2/4) and the offset is 1 (1%4) and 2 (2%4), respectively.

If the targeted sector in a data block is already filled with data, then FAST writes data in the log-block. The condition of writing in the SW log-block occurs either when the offset is zero or when the sector is sequentially filling the SW log-block. If neither of the conditions is satisfied, then data is written in the first empty sector of the RW log-block. For example, "w 1 C" and "w 2 D" are written in the RW log-block, PBN 100, because the targeted sectors are filled with data ("A" and "B") and they do not satisfy being in the SW log-block. Since data "A" and "B" are updated, they are denoted as invalid. On the other hand, "w 4 G" and "w 5 H" are written in the SW log-block because the offset of "w 4 G" is 0 and "w 5 H" fills the SW log-block after data "G" has been written.

When all sectors in PBN 100 are filled with data, FAST chooses the next available RW log-block, PBN 101. If both RW log-blocks, PBN 100 and PBN 101, are full, then the merge operation is performed. A merge operation in FAST is reclaiming all associated data

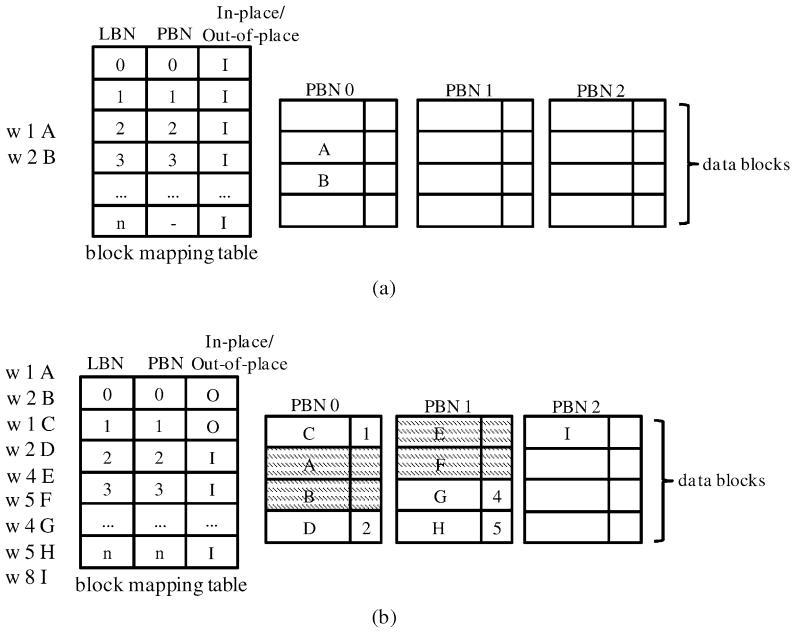

Fig. 12 EAST

blocks with the up-to-date data from RW log-blocks. Therefore, data from valid sectors are copied to free blocks and the erase operations would be performed on RW log-blocks and associated data blocks. In the worst case scenario, the data stored in the RW log-blocks can be associated with all different data blocks. This may require a chain of erase operations in all the associated data blocks by RW log-blocks. In the case of SW log-block, if the data is filled in the SW log-block, PBN 102, then the switch operation is performed similar to that of BAST.

EAST As we have seen, FAST is fully associative between logical sector and log-blocks; however, there are two main drawbacks. The first one is low space utilization. Though FAST has been upgraded from BAST, it still contains some data blocks having only 33% of used sectors before being forced to merge [22]. The second drawback is the cost of the merge operation. Since each of the entries in RW log-blocks can be associated with different data blocks, flash memory can have a large number of data blocks for the merge operation.

Kwon and Chung [22] propose two solutions for the above mentioned drawbacks of FAST. For low space utilization, An Efficient and Advanced Space-management Technique (EAST) uses both the in-place and out-of-place techniques in each of the data blocks for writing data. This will fully utilize the empty sectors before any merge operations. For the costly merge operation, EAST proposes a similar approach as that of BAST where each log-block is associated with one data block, but each data block can have multiple log-blocks given by an equation. The detailed explanation of EAST is given below.

Figure 12 shows the write operation of EAST with a small block flash memory. In order to intuitively compare with BAST and FAST, the same write operations used in Figs. 10 and 11 are executed in EAST. Figure 12 assumes that each block consists of four sectors. Since EAST is a block mapping based algorithm, it first divides the logical sector by the

number of sectors held by one block. The quotient from the resultant refers to LBN and the residual becomes the offset. LBN and mapped PBN are determined through the block mapping table. In case of the first write operation, "w 1 A", LSN 1 is divided by 4 that is the number of sectors held by one block. At this time, the quotient is 0, and the residual value becomes 1. Accordingly, LSN 1 is mapped to PBN 0 from the block mapping table, and the offset is 1.

The write operation is executed according to the state of the corresponding block. Both the in-place (indicated as "I") and out-of-place techniques (indicated as "O") can be used in EAST. The default state of all physical blocks is in-place, and they stay in the in-place state unless an overwrite occurs. The first and second write operations, "w 1 A" and "w 2 B", are examples of the in-place technique since the data is written on the position where the offset of the logical sector is identical to the offset of the physical sector, as is shown in Fig. 12(a).

In case of the third write operation, "w 1 C", data already exists in the second sector of PBN 0. EAST changes the state of LBN 0 to "O" and executes the write operation on an empty sector searching from the first sector of that physical block. For the preceding command "w 2 D", the data is written in the fourth sector of PBN 0 since the state of PBN 0 is "O". The rest of the write operations are performed by the same algorithm as shown in Fig. 12(b).

Due to the frequent update of random data, the "O" stated blocks are likely to be updated constantly. When the "O" stated block is fully occupied with the data, EAST allocates additional log-blocks. The maximum number of log-blocks per "O" stated block is determined by the following equation:

$$num\_of\_blocks\_per\_LBN = \left\lfloor \frac{time\_of\_erase}{time\_of\_read \times sectors\_per\_block} \right\rfloor \tag{5}$$

The numerator of (5) refers to the time for erasing one log-block and the denominator refers to the time for reading all the sectors within one log-block. Equation (5) guarantees the time for reading all the sectors of log-blocks does not exceed an erase operation. For example, in the case of [4], the time required for one read operation is 15 μs; the time required for one erase is 2 ms; and the number of sectors existing in one block is 32 sectors. Hence, *num_of_blocks_per_LBN* is four blocks (2000/(15×32)). Even though all four log-blocks are read, the total cost is about 1.9 ms ($32 \times 15 \times 4 = 1,920$ μs) which is smaller than the cost of an erase operation, 2 ms.

When data corresponding to LBN 0 is written repetitively, there will be a situation when there is no space limited by *num_of_blocks_per_LBN*. In this case, the merge operation is performed.

LAST Locality-Aware Sector Translation (LAST) [23] tries to enhance the shortcomings of FAST by providing locality detecting policy and regional division for random log-blocks. The locality detecting policy defines each write command's data as hot or cold depending on the request size. The experiments showed that the request size of the hot data is smaller than that of cold data. When the request size of a write command is smaller than the predetermined request size, the write command's data is defined as cold and it is written to the sequential log-block or to the data block.

The regional division for random log-blocks refers to the fact that LAST divides the random log-blocks into hot and cold region in order to solve the high cost of merge operation in FAST. Even though the hot data is defined by the locality detecting policy, some hot data are updated more frequently than other. The random log-blocks with highly updated hot data

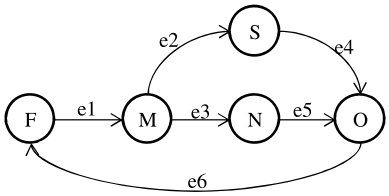

Fig. 13 Block state machine

would mainly consist of invalid data thus reducing the total number of merge operations. However, Lee et al. admit that the proposed locality detector cannot efficiently identify sequential writes when the write workload is mainly composed of the small-sized sequential writes. As LAST cannot adapt the dynamic workload patterns, its utility can be extremely restricted.

3.3.3 The state transition scheme [24, 25]

The basic idea of the state transition scheme is to enhance performance by referring to the state information when performing read/write commands. Among the various algorithms of state transition schemes, State Transition Applied Fast FTL (STAFF) [24] is well known for classifying the physical blocks into various types of states.

STAFF is based on the block mapping algorithm, but it can allocate two physical blocks to one logical block under a certain circumstance. STAFF minimizes the total number of erase operations by assigning a state to each block and by converting the block into the appropriate state according to the input pattern. STAFF performs read and write operations differently according to the following states.

- F (Free) state: If a block has been erased and has not yet been written, then the block is said to be in the "F" state.
- M (Modified In-place) state: If some (but not all) sectors of a block are written in the in-place technique, then the block is said to be in the "M" state.
- S (Complete In-place) state: If all sectors of a block are written in the in-place technique, then the block is called the "S" state block.
- N (Modified-out-of-place) state: If there is more than one sector written in the out-of-place technique, then the block is said to be in the "N" state.
- O (Obsolete) state: If there is no valid data within a block, then the block is stated as "O". The "O" stated block can be immediately erased, and the state is changed to "F".

The state information is stored in SRAM and in the spare area of each block. When a state is converted to another, as shown in Fig. 13, the state is re-recorded into the spare area by using partial programming. Since there are five states, the state can be represented by three bits in the spare area.

Figure 13 shows the block state machine, and Fig. 14 explains the block state machine with an example. The initial block state is the "F" state. When an "F" state block receives the first write request, the "F" state block is converted to the "M" state (event e1). The first write operation, "w 0 A", is written in the first sector of PBN 0, since LBN 0 (0/4) retrieves PBN 0 from the block mapping table, and the logical offset is 0 (0%4). As the first write operation occurs in PBN 0, the state is converted to "M" in the block state table. The event e2 is shown in Fig. 14(b). The physical block, PBN 0, is changed to the "S" state, since all the empty sectors of PBN 0 are filled with valid data.

When an update, "w 0 E", occurs at the point of event e2, STAFF allocates a physical block, PBN 100, as a buffer to the original data block, PBN 0, and it writes the updated data

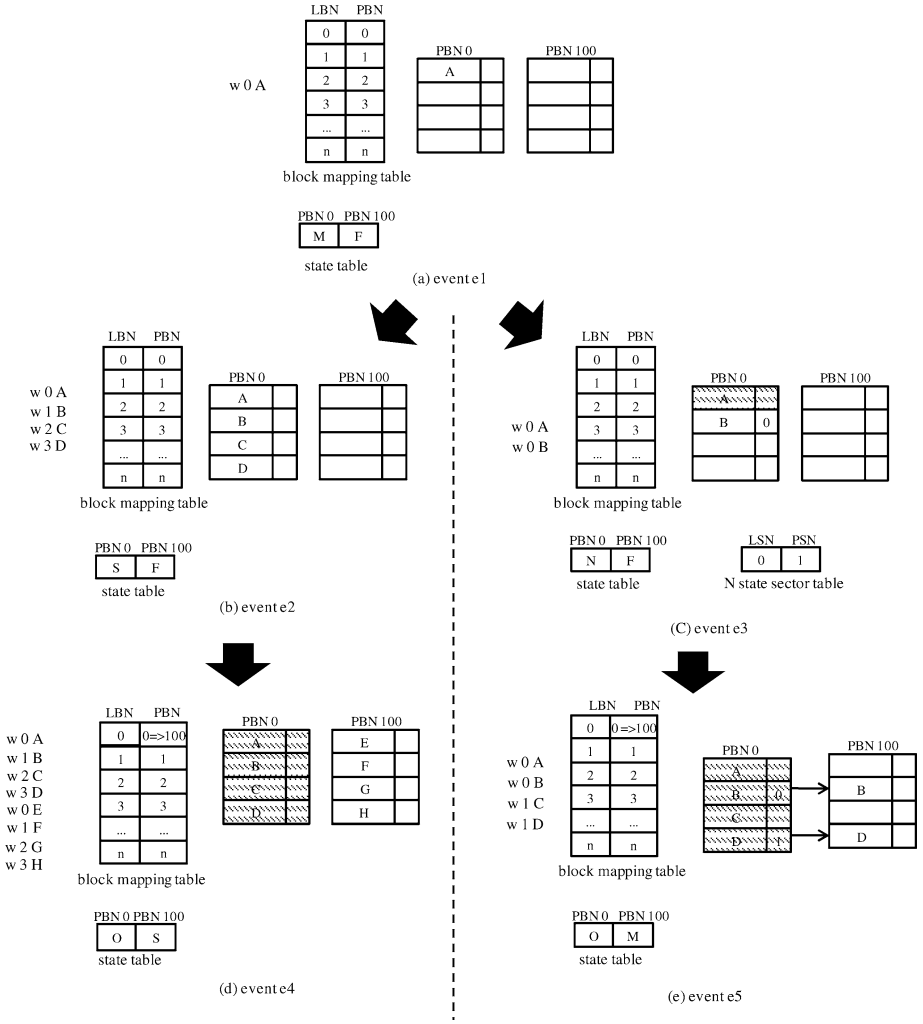

Fig. 14 STAFF

according to the in-place technique. The state of PBN 100 is changed from the "F" state to the "M" state. When all the sectors of the "S" state block are updated to the newly allocated buffer, the event e4 occurs as shown in Fig. 14(d). PBN 0 is changed to the "O" state, and PBN 100 is changed to the "S" state since all the updated data are stored in PBN 100.

Another scenario can occur at the point of event e1. When a write operation with an identical LSN occurs, the event e3 occurs. As shown in Fig. 14(c), "w 0 B" creates a situation, where the data needs to be written to another empty sector, since the first sector of PBN 0 is already occupied by data "A". In this case, the "M" state block, PBN 100, is changed to the "N" state, and the new data is written in the first empty sector searching from the top. Next, its logical sector number is indicated in the spare area, since the data is written according to the out-of-place technique. Therefore, "B" is written to the second sector of the PBN 0 and "0" is indicated in the second sector of spare area. STAFF manages a sector mapping

table for "N" state blocks since each "N" state block is written according to the out-of-place technique.

Event e5 occurs when the "N" state block, PBN 0, is full. Valid data in the "N" state block is copied to the newly allocated block, and the "N" state block is converted to the "O" state as shown in Fig. 14(e). Therefore, the states of PBN 0 and PBN 100 are changed to "O" and "M", respectively. The "O" state block can be immediately erased when a free block is needed. Erasing the "O" state block is event e6.

As seen from the explanation of each event, Chung and Park manage each physical block by marking and monitoring the states. However, one crucial drawback of STAFF is predicting the SRAM requirement. In other schemes such as log-sector and log-block scheme, the requirement of SRAM can be precisely predicted as the number of log-blocks is fixed. However, in STAFF, the number of the N state blocks varies according to the trace which gives an unpredictable size of SRAM.

LSTAFF [25] "System Software for Large Block Flash Memory (LSTAFF)" is modified from STAFF in order to efficiently adopt the characteristics of the large block flash memory. As we have mentioned in Sect. 2.2.1, the large block flash memory's page size is composed of multiple sectors, although the file system's write request size is only one sector. As solution, LSTAFF gathers the file system's data sectors into the unit of page by aggressively using the random data out/in technique (explained in Sect. 2.2.3). Instead of immediately writing data onto the flash memory array, the random data out/in technique allows gathering the file system data sectors in flash memory's internal buffer. LSTAFF gathers differently according to the state machine. Let us assume the file system has issued write commands corresponding to LSN 0, LSN 1, and LSN 10. If the corresponding block's state is set to "F" (Free) or "M" (Modified In-place), the data corresponding to LSN 0 and LSN 1 are gathered separately from LSN 10, since LSN 10 is not relevant to LSN 0 or LSN 1. On the other hand, in case the block's state is set to "N" (Modified-out-of-place), the block is written in out-of-place; therefore, the data corresponding to LSN 0, LSN 1 and LSN 10 are gathered as one page.

3.3.4 The block-set scheme [26–28]

Currently there have been few FTL algorithms based on the large block flash memories. Most of their algorithms are modified from the log-block scheme; therefore, the basic usage of log-blocks and read/write command policy is the same. However, as the capacity of flash memory has been significantly increased, traditional sector/block mapping algorithms have difficulties maintaining all the mapping information in RAM. As a solution, Hsieh et al. [26] propose Configurable Flash-Memory Management: Performance versus Overheads (CNFTL), which provides flash memory vendors flexibility to have flexibility on configuring the mapping table according to the vendors' RAM space constraints. The mapping table can be automatically tuned up from sector mapping to block mapping depending on the characteristics of flash memory by simply setting parameters, such as RAM, page and block size, in implementation. However, their main contribution can be viewed as the first proposal on "block-set" concept. In their paper, they have grouped fixed number of logical blocks as one region, and managed a logical region table instead of maintaining the block mapping table.

The main concept of the block-set scheme is to group N adjacent data blocks and M log-blocks into one block-set as summarized in Table 5. M log-blocks are used as a buffer for entire N data blocks. Normally N is fixed while M can be dynamically changed according to

Table 5 Characteristics of performance enhancing algorithms [28]

(a) Data blocks

	Schemes			
	Log-sector	Log-block	State transition	Block-set
Terminology	Primary or original block	Data block	F/M/S block	Data block
Max. degree of sharing	1	1	1	*N* data blocks in block-set
Management technique	In-place	In-place BAST FAST LAST / In-/out-of place EAST	In-place	Out-of-place

(b) Update blocks

	Schemes					
	Log-sector		Log-block		State transition	Block-set
	Space sectors	Replacement blocks	Log-block	R log-block S log-block		
Terminology	Mitsubishi	ANAND, FMAX, AFTL, RNFTL	BAST, EAST	FAST, LAST	*N* state block	Log-block
Max. degree of sharing	1		BAST, FAST	1 R log-block: pages per block S log-block: 1	1	*M* log-blocks in block-set
			EAST LAST	Equation (5) R log-block: pages per block S log-block: *n*		
Management technique	In-place ANAND	Out-of-place Mitsubishi, FMAX, AFTL, RNFTL	Out-of-place		Out-of-place	Out-of-place

the number of current log-blocks. When a new log-block is allocated to the block-set, M is increased. On the other hand, M is decreased when a merge operation is performed on data blocks and log-blocks. The block-set scheme has targeted and compared their algorithms to previous schemes, such as log-sector and log-block schemes. The variations of the block-set FTL algorithms are shown below.

SAST Park et al. propose an FTL algorithm called Set Associative Sector Translation (SAST) [27], which contains a block-set mapping table and maintains a page mapping within a block-set. The page mapping table is used to separate hot and cold data within the block-set. The mapping algorithm of SAST can be viewed as the extension of the hybrid mapping algorithm, because the block-set mapping can be seen as the extension of block mapping algorithm and the page mapping can be seen as the extension of the sector mapping algorithm.

Superblock FTL Superblock FTL [28] is very similar to SAST, since both FTL algorithms follow the concept of log-block and use the merge operation to reclaim the data block. The main difference between SAST and Superblock FTL is the method of managing the mapping table. SAST contains the page mapping using a small amount of RAM for rapid inquiry; on the other hand, Superblock FTL proposes the method of recording the page mapping information within the spare area of each sector. The method of maintaining the mapping information in the spare area is explained in Sect. 3.4. However, employing the mapping table within the spare area can cause frequent partial reads and writes.

3.4 Mapping information management

Most performance enhancing algorithms load the mapping tables to RAM. However, it is required to store the mapping information into the flash memory array from time to time as RAM is a volatile memory. There are basically two methods of storing the mapping information.

3.4.1 Block-level management

The block-level management refers to the method of allocating some dedicated blocks in flash memory for storing the mapping table. Kim et al. [20] term these blocks as "map blocks", and Gupta et al. [29] and Qin et al. [30] term these blocks as "translation blocks". The mapping information (pairs of logical and physical addresses) is recorded to the unused sectors of the map block. For example, if the requirement of RAM for indicating PBN is 3 or 4 bytes, each 512 bytes sector can store approximately 170 or 128 PBNs.

Kim et al. propose to store the page/block mapping table within the map blocks, and upload the entire mapping table to RAM when the system has been initiated. This method can provide fast booting since all the mapping information is gathered into few blocks; however, Gupta et al. pointed out the possibility of drastic increase of RAM requirement when the entire mapping table is uploaded to RAM. As a solution, they propose A Flash Translation Layer Employing Demand-based Selective Caching of Page-Level Address Mappings (DFTL) [29], which directly inquires the page mapping table from the map blocks and separates cached mapping table in RAM. The cached mapping table is designed to selectively cache the frequently accessed logical and physical addresses. This selective cached mapping table enables a direct access to the physical addresses of frequently accessed data without evoking extra read operations while searching within the map block. However, DFTL requires large number of map blocks for maintaining the page mapping table in flash memory [30]. Therefore, Qin et al. propose to store only a block mapping table within the map blocks, but to contain two-level caches. The first-level cache is used for storing frequently accessed block mapping table, and second-level cache contains the mapping information of the corresponding block's sequential access addresses and most frequently accessed addresses. According to their experiment, Qin et al. has reduced approximately 84 times of map blocks from DFTL.

3.4.2 Sector-level management

Instead of dedicating few blocks as map blocks, the sector-level management scatters its mapping information in some dedicated sectors or in some sectors' spare area. Wu et al. [31] have allocated every last sector/page of each block as "summary sector/page". When a block is about to fill up, the summary page stores the mapping information of a block. Jung et al. [28] propose another variation of the sector-level management. They propose the method of storing the mapping information within the spare area. They aggressively use the spare area of each sector (16 bytes) for storing the sector level mapping information. As the size of spare areas is not enough to store one sector/page mapping table, they have divided each physical address into three levels of bits: high, middle and low. As a result, they contain small-sized three level tables instead of one sector/page mapping table. The main disadvantage of the sector-level management is the possibility of slow booting due to the cost of scanning the scattered information.

3.5 Durability enhancing algorithms

Every block in flash memory has a program/erase cycle limit. When a block reaches the program/erase cycle limit, the block wears out and the data on that block cannot be reliable. Therefore, FTL helps to extend the life of flash memory by reducing the total number of erase operations and by evenly distributing the erase operations to all the blocks. Previous works have not clearly demarked the differences between wear-leveling algorithms and cleaning policies. Some have mixed the terms together while others separated each other with very unclear demarcation. We have separated the two terms depending upon their level of monitoring. The wear-leveling algorithms are characterized by the block-level monitoring whereas the cleaning policies with sector/page-level monitoring. If the level of monitoring gets finer, the extra read/write operations increases thus degrading the performance and durability of flash memory. We have explained the common characteristics of wear-leveling algorithms in Sect. 3.5.1, and categorized previously proposed wear-leveling algorithms and cleaning policies in Sects. 3.5.2 and 3.5.3, respectively. In Sect. 3.5.4, we have explained the erase count block, which is a widely known technique used in both wear-leveling and cleaning policies for convenient monitoring.

3.5.1 The characteristics of wear-leveling algorithms

Most of the existing wear-leveling algorithms are based on the hot-block concept. The hot-block refers to the block with many erase operations as the write operations are repeated on the same sector for many times. After a fixed amount of time has passed or after a fixed number of operations has been executed, the wear-leveling algorithm monitors the physical blocks or groups, and hot and cold swapping is considered. The existing hot and cold-block swapping algorithms hold the following two common characteristics:

Practical implementation The flash memory cannot guarantee performance and durability if only one area is considered. We need to consider the practical difficulties of combining the performance and durability algorithms. While the performance and wear-leveling algorithms are executed, the states of sector/block are stored in the spare area and are updated by the partial programming. The number of partial programming allowed in the spare area before an erase operation is limited to $1 \leq \chi \leq 4$ as mentioned in Sect. 2. Due to this constraint, one might have difficulties in selecting a suitable wear-leveling algorithm. For example, if flash

memory contains two partial programmings and the performance algorithm requires one partial programming, the remaining partial programming for flash memory is just one. It is impossible to implement the wear-leveling algorithms, which require two or more partial programmings, even though these algorithms can result in better solution.

Periodic monitoring In addition to the practical implementation, the wear-leveling algorithms have a common characteristic of periodically monitoring the physical blocks or groups in a certain period of time or in a fixed number of operations. Efficient monitoring is an important issue, because the unnecessary monitoring of blocks/sectors can result in large performance deviation. For example, the read operations for information inquiry occur periodically even though the deviation of the erase operations is constantly narrow.

3.5.2 Wear-leveling algorithms

ECN based wear-leveling algorithms [32–35] The blocks in flash memory are constantly monitored to keep track of their conditions. Each block's ECN is recorded in the spare area. If the usage of a certain block is heavier than other blocks, the system triggers the wear-leveling process. We classify the wear-leveling algorithms which consider the wear-leveling process according to the each block's ECN as the ECN based wear-leveling algorithms.

One of the respective ECN based wear-leveling algorithms is called dual pool [32]. The dual pool refers to the simple but effective mechanism which classifies the blocks into three tables: cold-block, hot-block and block mapping. The hot/cold block table contains empty blocks with overused/underused blocks, and block mapping table refers to the block mapping algorithm as mentioned in Sect. 3.1.2. Acquiring the information of the hot/cold blocks does not require much computational power, since all the tables are stored in SRAM. Each table contains all the required information such as ECN, average of ECNs, and validation flag. Thus, the system can retrieve the information of each block from the tables without inquiring the spare area of flash memory.

The dual pool constructs each table as following. First, the categorization is initiated by receiving an initialization request. Next, the system sorts the blocks according to ECN. M most frequently used blocks are stored in the hot-block table, and N least frequently used blocks are stored in the cold-block table. The remaining blocks are stored into the block mapping table. Finally, the system stores the average of ECN in SRAM.

The blocks in the hot-block table are reserved, whereas the blocks in the cold-block table are frequently utilized in order to balance the overall durability degradation of flash memory. Therefore, after a certain amount of time, the previous hot/cold-blocks can no longer be hot/cold. The system redefines the blocks as hot or cold and reorganizes the tables by the following procedures.

First, the system calculates the difference between each block's ECN and the average. Next, the result is compared with a predetermined value "A". If the result is larger than "A", the block is determined as a hot-block, otherwise a cold-block. "A" may vary considerably depending upon the desired flash memory system environment. It is imposed on how often the wear-leveling process is performed. If "A" is too large (e.g., 1,000,000 erases), it will shorten the life of one or more blocks. On the other hand, if the "A" is too small, the overall performance of flash memory can be degraded due to the overheads of frequent wear-leveling. Finally redefined hot/cold blocks are newly inserted to the hot/cold table.

Write based wear-leveling algorithms [36, 37] Achiwa [36] describes another wear-leveling algorithm which is based on recording the number of writes in SRAM. The system

records a write count for each block in SRAM, and updates the count whenever the corresponding block or group has been accessed. The write counts are utilized to store highly accessed data to the least frequently erased blocks and to store less accessed data to the most frequently erased blocks. It is necessary to store highly accessed data into the least erased block, since data with the highest access count is likely to be updated in the future.

Chang [37] proposes the write based dual pool. The write based dual pool considers wear-leveling by the frequency of accesses instead of considering the ECN as the main factor for wear-leveling. The write based dual pool contains a highly accessed block table and a lowly accessed block table in SRAM. A block is considered to be highly or lowly accessed by counting the number of write operations performed on that block. The data swapping is similar to the ECN based wear-leveling algorithms. However, the special feature of this algorithm is resizing the table. Instead of having a fixed number of entries for highly and lowly accessed tables, Chang resizes the number of entries for each table. The number of entries does not change if the pattern of the write operation is the same. On the other hand, if the pattern changes, a highly accessed block can be changed to a lowly accessed block and vice-versa. In this case, Chang adjusts the number of entries within hot/cold tables and rearranges the blocks accordingly.

Group based wear-leveling algorithms [38–40] In the group based wear-leveling algorithms, the physical blocks are grouped into larger physical concepts to reduce the SRAM requirement and the amount of information required for wear-leveling. Let us assume one such group based algorithm groups ten blocks as one physical unit for 256 Mbytes flash memory. The number of entries for the group mapping table would be reduced into 1/10 compared to the block mapping table. The group mapping table would only require 1.6 Kbytes (16 Kbytes/10), since the block mapping table requires 16 Kbytes (2 bytes $\times$ 8192) according to Sect. 3.2.2. The amount of information required by additional tables, such as the ECN table and hot and cold ECN tables, can also be reduced by storing the average of each group.

Lofgren [38] has grouped three to four blocks as one physical unit and termed it as a bank. Their algorithm manages a group mapping table, and calculates an average for every bank's corresponding blocks. Hot and cold group swapping is performed when the deviation between the highest and lowest bank's average is larger than threshold "A".

Conley [40] manages flash memory in zones. A zone is a much larger physical unit compared to a block or a bank. The entire blocks of flash memory are divided into four to twenty zones, and FTL attempts to keep the equivalent number of free and data blocks in each zone. If one zone contains a smaller number of free data blocks compared to other zones due to factory defects or frequently erase operations, then the insufficient blocks are supplied by other overly contained zones.

3.5.3 Cleaning policies

Frequent data update leaves the status of previously written sectors with the same LSNs as invalid. These sectors with invalid data, known as dirty sectors, need to be cleaned to create free space for new data. The process is termed as cleaning policy or garbage collection. If FTL does not contain any cleaning policy, the blocks with large amount of dirty sectors will rapidly runout of free space thus evoking frequent erase operations.

The basic cleaning policies evaluate a block or a group of blocks as dirty according to the block/group utilization. Here, the block/group utilization refers to the ratio of dirty sectors over total number of sectors. The basic cleaning policies remove the dirty blocks/groups by copying the valid data to newly allocated blocks/groups and erasing the previous dirty blocks/groups.

The cleaning policy requires checking the cost of monitoring. The cost can be much higher than that of wear-leveling, since the cleaning policy requires monitoring the condition of each sector in order to determine the dirtiness of each block/group. Some cleaning policies allocate a small amount of RAM for gathering and indicating the condition of each sector condition. Others manage their algorithms by just utilizing the spare area.

The cleaning policy can be implemented with wear-leveling algorithms as in JFFS2 [41] and in YAFFS [42]. Each block/group follows the wear-leveling algorithm or cleaning policy depending on its condition. However, each block/group can be switched to follow from wear-leveling algorithm to cleaning policy or vice-versa. For example, when the number of executing wear-leveling algorithm within a block/group reaches some threshold value defined by the system, the block is switched to the cleaning policy.

Hot and cold data separation [43–50] Hot and cold data separation and victim selection are the techniques organized by Chiang [51]. These techniques organize data within flash memory to carefully select a dirty block/group.

The hot and cold data separation is a widely used technique in the cleaning policy. It reduces the overhead of copying the data during cleaning. The block that is chosen to be cleaned would contain data with varying numbers of updates. The data associated without any update is termed as cold data, and the data that has been repeatedly updated is called hot data. If a block is determined as dirty, all the hot and cold data within the block are copied to newly allocated block. The number of sectors containing the valid hot data would be relatively small, since most sectors related to the hot data would be invalid due to the frequent update. On the other hand, the number of sectors containing the cold data would large, since the cold data rarely updates.

When both hot and cold data coexist in a block, there is high chance that another cleaning process would be required in the near future due to the repeated updates of hot data. In order to overcome the disadvantages caused by coexistence, the hot and cold data are separated into different areas. There have been several proposals for hot-and-cold data separation: Dynamic dAta Clustering (DAC) [43], two level LRU [44], Hot and Cold Data Identification [45, 46], Log-Block Based Cleaning Policy [47], Block-Set Based Cleaning Policies [48, 49], and Improving Flash Wear-Leveling by proactively Moving Static Data [50].

- *Dynamic dAta Clustering (DAC)* partitions the flash memory into several different regions to classify the data according to the access frequencies. Each region consists of a set of blocks with similar write access frequencies. For example, let us assume that the FTL programmer created a design in which most updated data are stored in the top region and the least updated data at the bottom, or vice versa. The data within each page will be moved to the top of the region if the update frequency of the data is increased. On the other hand, the data with less updates will be moved to the bottom of the region.
- *Two-level LRU* separates the hot and cold data by having two lists: hot and candidate. The hot list contains the logical block addresses of hot data, where as the candidate list contains the logical block addresses of cold data. Each list is sorted by the access frequencies. When a write operation is performed, the two-level LRU checks whether the logical addresses exist within hot or candidate list. If the address does not exist in both lists, then the address is added to the candidate list. When the address is found in the hot list, the address is moved toward the head of hot list. On the other hand, if the address is found in the head of candidate list, then the address is transferred to the hot list.
- *Block-Set Based Cleaning Policies* are proposed for the block-set scheme. Their basic concept is very similar to two-level LRU in the aspect of separately managing the cold and hot data in different lists. However, the block-set based cleaning policies only identify

hot and cold within a block-set by obtaining the number of updates from the page mapping table. As the page mapping table is already required by the block-set scheme, there is no additional overhead of RAM.

– *Log-Block Based Cleaning Policy* is based on applying a cleaning policy for the log-block scheme. Cho et al. [47] proposes K-Associative Sector Translation (KAST) in which its mapping algorithm is based on FAST (log-block scheme). The main contribution of KAST is to modify the merge operation to reduce the log-block associativity. The log-block associativity refers to the number of data blocks which are relevant to the merge operation. When the log-block associativity is reduced, the cost of merge operation can be significantly reduced. As a result, the durability of flash memory can be increased, since the merge operation is a process of eliminating/cleaning the invalid data. KAST writes the updates with different LBN in different random log-block, instead of simply accumulating the updates as in FAST. This simple modification can reduce the number of associated data blocks within one merge operation, since it enhances the possibility of containing relevant LSNs within one random log-block.

  KAST modifies the switch operation (explained in Sect. 3.3.2) as well. According to their experiments in real-time systems, they claim the switch operation evokes inefficient erase operations, because it generates an erase operation whenever an update with page 0 or an update with non-sequential LSN occurs. As a solution, KAST gives an option of writing the update with page 0 to the random log-block. If the current sequential log-block does not contain a large amount of data, the space utilization can be low. Thus KAST writes the data in the random log-block. Furthermore, KAST provides a choice of changing the status of sequential log-block as random when an update with non-sequential LSN occurs. If the sequential log-block contains large amount of free space and there are no random log-block with low associativity, the sequential log-block is changed to random log-block and the update is written on it. These methods allow to utilize all the space of each log-block before having an erase operation.
– *Hot and Cold Data Identification* is proposed by Syu et al. [45] and Hsieh et al. [46]. Syu et al. utilize the register table to classify the data as hot or cold. The logical addresses are partitioned into several clusters, and each cluster has a corresponding one bit flag in the register table. The flag of the corresponding cluster is set to one when the write operation is performed, otherwise zero. The flag bits are checked by the system at each interval. If the flag bit is set to one frequently, then the data is considered as hot data. However, this method does not guarantee the overhead of RAM and does not consider its false identification. As a solution, Hsieh et al. propose an identification reviewing algorithm which examines and reduces the chance of identifying false hot data by adopting multiple independent hash functions. According to their experiment, the hash table only requires 2 Kbytes of RAM per 512 Mbytes flash memory.
– *Improving Flash Wear-Leveling by proactively Moving Static Data* terms the cold data as static data. It proactively moves the static data according to the access frequencies and it considers each block's ECN for allocating static and non-static data.

Various methods of dirty block/group selection [52, 53] One of the most important aspects of the cleaning policy is efficiently selecting a dirty block/group. The methods of selecting a dirty block/group differ depending on each cleaning policy, but the main aim of the selection process is to choose a block/group with much less usable space and a lot of invalid data. The greedy policy, known as the basic cleaning policy, simply selects and cleans the block/group with the most invalid sectors. It is very effective when the blocks are uniformly accessed; however, this differs depending upon the pattern of write operations [53]. Some

policies evaluate the cost benefit from the block/group information. Here, the block/group information includes the ratio of valid data within a block/group and the elapsed time since current modification.

In Cost Benefit Policy [52], the block/group is decided to be cleaned according to (6). Equation (6) calculates the probability of each block/group's dirtiness.

$$cleaning = age \times (1 - u)/2u \tag{6}$$

Here, u is the percentage of the valid data in a block; therefore, $1 - u$ is the percentage of free space reclaimed, and *age* is defined as the elapsed time since the most recent modification. $2u$ stands for the cleaning cost of the erase block (e.g., u to read valid data from the block and u to write them to another block). All the blocks are calculated via the equation, and the block with the maximum value is chosen to be cleaned.

Another variation of dirty block/group selection is the Cost Age Time Policy (CAT). It uses the time factor for the data reclamation as shown in (7).

$$cleaning = u/(1 - u) \times (1/age) \times number\_of\_cleaning \tag{7}$$

The feature of CAT is to give the hot-block more time to accumulate invalid data before reclamation. CAT selects the blocks with smaller ECN for cleaning.

Some of the policies such as Cleaning Index Cycle Leveling [53] use both wear-leveling and cleaning policy to avoid wearing out flash memory. Cleaning Index Cycle Leveling selects the victim block in similar fashion to other dirty block/group selection algorithms, but also periodically executes wear-leveling as in JFFS and YAFFS. Every policy has its own different advantages and disadvantages, and can be used efficiently according to the requirements or for better results.

3.5.4 Erase count block [34, 54]

The informations, which are required by the FTL algorithms, are recorded in SRAM and in the spare area. Generally, when the system initiates, each block/group's information is inquired from the spare area, and is uploaded to SRAM. Most of the wear-leveling algorithms treat ECNs as important information, since ECN is directly related to the wear-leveling. However, inquiring ECN from each block's spare area can be a costly process. As a solution, Chang [34] proposes to gather ECNs in one or few blocks. Chang terms these blocks as the Erase Count Blocks (ECB).

Each entry within the ECB has an associated block in the flash memory, and the information can be obtained by just reading the ECB. This enables all the frequently used blocks to be easily identified. While identifying the frequently used blocks, unusable blocks are also identified. The blocks can no longer be usable when they have reached the program/erase cycle limit or when they have factory defects. An unusable block is marked with "FFFFFF" in either case.

The ECB is initialized according to the following order. First, the system examines the blocks and identifies all the unusable blocks. All the unusable blocks are marked in the ECB. Next, the ECNs of usable blocks are recorded to the ECB, and the average of the ECNs is recorded as well if needed.

Lasser [54] proposes the "turning off" mechanism in order to achieve fast wakeup. The fast wakeup is a rapid initialization of the system that is achieved by reading the required information from the ECB instead of searching the spare area of each block. In order to utilize ECB, the system needs to assure the flash memory does not have any unsaved data

before shutting off. As a solution, a validation flag is used at the end of ECB. If the turning-off procedure is properly executed, then the system would achieve a validation flag at the end of ECB, and initializing flash memory would be fast since ECNs are directly read from the erase count block. If the application software made a non-orderly turning-off, then there would not be a validation flag in ECB; a regular wake-up procedure is initiated, thereby ensuring data integrity.

4 Conclusion

In recent years, there have been many studies on flash memories, especially on NAND type flash memory, due to its low electronic power, non-volatile storage, high performance, physical stability and portability. We have explained the hardware organization of small block and large block NAND-type flash memory, and we have pointed out two main hardware issues: erase-before-write and degradation of physical blocks. In this survey paper, "FTL Algorithms for Flash Memories", we define the role of the intermediate software layer, FTL, as the solution to the hardware issues of flash memory, and we have studied the available FTL algorithms. We classify the important and prominent FTL algorithms used in NAND-type flash memories according to different aspects.

First, we have classified FTL algorithms into basic and advanced mapping depending on the complexity of the mapping method. The basic mapping algorithms are the naive and intuitive algorithms that simply map logical addresses to physical addresses in the unit of a sector or block. The advanced mapping algorithms are derived from one or two basic mapping algorithms with advanced features such as the partial programming and static/dynamic allocation.

The advanced mapping algorithms can be further classified according to their roles: performance and durability. Performance enhancing algorithms are mostly concerned with the erase-before-write issue. It is related to a reduction in the number of erase operations utilizing small amount of RAM. We classify these performance enhancing algorithms into three major schemes: log-sector based, log-block based and state based depending upon the methods of utilizing spare blocks of flash memory.

Another area of classification on the advanced FTL algorithms is the durability enhancement. The durability enhancement is related to the degradation of the physical block. It is the FTL algorithm's duty to distribute the erase operations as evenly as possible. We have analyzed the common characteristics of the durability enhancing algorithms, and further classified the durability enhancing algorithms into two areas: wear-leveling and cleaning policy depending on the level of monitoring. We have viewed the monitoring level of wear-leveling as a block, since they are mainly concerned with ECN. On the other hand, the level of monitoring in the cleaning policy is a sector, since the system requires to check each sector's validation in order to analyze the dirtiness of blocks/groups.

In this study, we have aimed at properly surveying the important features of flash memories and FTL algorithms with a new level of classifications. Till now, clear demarcation of the work on the flash memories is not available. This work will be greatly beneficial to future studies in this area of overall developments in the field of flash memories.

References

1. http://en.wikipedia.org/wiki/flash-memory

2. http://www.samsung.com/sec/business/semiconductor/products/Products.html
3. Intel Corporation (2010) Nor flash memory. 28F640J3A data book
4. Samsung Electronics (2010) Nand flash memory. K9F5608X0D data book
5. Gal E, Toledo S (2005) Algorithms and data structures for flash memories. ACM Comput Surv 37(2):138–163
6. Chung T-S, Park D-J, Park S, Lee D-H, Lee S-W, Song H-J (2009) A survey of flash translation layer. J Syst Archit Embed Syst Design 55(5–6):332–343
7. Samsung Electronics (2010) Nand flash memory. K9F1G16U0M data book
8. Samsung Electronics (2010) Nand flash memory. K9GAG08U0M data book
9. Dan R, Singler R (2003) Implementing MLC NAND flash for cost efficiency, high-capacity memory. M-Systems Inc
10. Samsung Electronics (2009) Page program addressing for MLC NAND application note
11. Kawaguchi A, Nishioka S, Motoda H (1995) A flash memory based file system. In: 1995 USENIX technical conference, pp 155–164
12. Resenblum M, Ousterhout J (1992) The design and implementation of a log-structured file system. ACM Trans Comput Syst 1(1):26–52
13. Wu M, Zwaenepoel W (1994) eNVy: a non-volatile, main memory storage system. In: International conference on architectural support for programming language and operating systems
14. Ban A (1995) Flash file system. United States Patent, No 5,404,485
15. Kim B-s, Lee GY (2002) Method of driving remapping in flash memory and flash architecture suitable therefor. United States Patent, No 6,381,176
16. Shinohara T (1999) Flash memory card with block memory address arrangement. United States Patent, No 5,905,993
17. Ban A (1999) Flash file system optimized for page-mode flash technologies. United States Patent, No 5,937,425
18. Wu CH, Kuo TW (2006) 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, pp 601–606
19. Wang Y, Liu D, Wang M, Qin Z, Shao Z, Guan Y (2010) RNFTL: a reuse-aware NAND flash translation layer for flash memory, LCTES, pp 163–172
20. Kim J, Kim JM, Noh SH, Min SL, Cho Y (2002) A space-efficient flash translation layer for compact flash systems. IEEE Trans Consum Electron 48(2):366–375
21. Lee S-W, Park D-j, Chung T-S, Lee D-H, Park S, Song H-J (2007) A log buffer based flash transition layer using fully associative sector translation. ACM Trans Embed Comput Syst 6(3):1–27
22. Kwon SJ, Chung T-S (2008) An efficient and advanced space-management technique for flash memory using reallocation blocks. IEEE Trans Consum Electron 54(2):631–638
23. Lee S, Shin D, Kim Y, Kim J (2008) LAST: locality-aware sector translation for NAND flash memory-based storage systems. In: Proceeding of IEEE international workshop on storage and I/O virtualization, performance, energy, evaluation and dependability (SPEED08), pp 36–42
24. Chung T-S, Park H-S (2007) STAFF: a flash driver algorithm minimizing block erasures. J Syst Archit 53(12):889–901
25. Chung T-S, Park D-J, Ryu Y, Hong S (2004) LSTAFF: system software for large block flash memory. In: AsiaSim 2004, pp 704–712
26. Hsieh J-W, Tsai Y-L, Kuo T-W, Lee T-L (2008) Configurable flash-memory management: performance versus overheads. IEEE Trans Comput 57(11):1571–1583
27. Park C, Cheon W, Kang J-U, Roh K, Cho W, Kim J-S (2008) A reconfigurable FTL (flash translation layer) architecture for NAND flash-based applications. ACM Trans Embed Comput Syst 7(4):1–23
28. Jung D, Kang J-U, Jo H, Kim J-S, Lee J (2010) Superblock FTL: a superblock-based flash translation layer with a hybrid address translation scheme. ACM Trans Embed Comput Syst 9(4):1–41
29. Gupta A, Urgaonkar B (2009) DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In: ASPLOS'09, pp 229–240
30. Qin Z, Wang Y, Liu D, Shaom Z (2010) Demand-based block-level address mapping in large-scale NAND flash storage systems. In: CODES+ISSS'10, pp 173–182
31. Wu P-L, Chang Y-H, Kuo T-W (2009) a file-system-aware FTL design for flash-memory storage systems. In: DATE 2009, pp 393–398
32. Assar M (1995) Flash memory mass storage architecture incorporation wear leveling technique. United States Patent, No 5,479,638
33. Han S-W (2000) Flash memory wear leveling system and method. United States Patent, No 6,016,275
34. Chang RC (2006) Method and apparatus for managing an erase count block. United States Patent, No 7,103,732

35. Wells SE, Heights C, Calif (1994) Method for wear leveling in a flash EEPROM memory. United States Patent, No 5,341,339
36. Achiwa K (1999) Memory system using a flash memory and method of controlling the memory system. United States Patent, No 5,930,193
37. Chang L-P (2007) On efficient wear leveling for large-scale flash-memory storage systems. In: Proceedings of the 2007 ACM symposium on applied computing, March 11–15, 2007, Seoul, Korea
38. Lofgren KMJ (2005) Wear leveling techniques for flash EEPROM systems. United States Patent, No 6,850,443
39. Jung D, Chae Y-H, Jo H, Kim J-S, Lee J (2007) A group-based wear-leveling algorithm for large-capacity flash memory storage systems. In: Proceedings of the international conference on compilers, architecture, and synthesis for embedded systems (CASES), Salzburg, Austria, September 2007
40. Conley KM (2002) Zone boundary adjustment for defects in non-volatile memories. United States Patent, No 6,901,498
41. Woodhouse D (2001) JFFS: the journaling flash file system. In: Proceeding of Ottawa Linux symposium
42. Manning C, Wookey (2001) YAFFS specification. Aleph One Limited
43. Chiang M-L, Cheng C-L, Wu C-H (2008) A new FTL-based flash memory management scheme with fast cleaning mechanism. In: 2008 international conference on embedded software and systems, pp 205–214
44. Chang LP, Kuo TW (2002) An adaptive stripping architecture for flash memory storage systems of embedded systems. In: IEEE eight real-time and embedded technology and applications symposium, San Jose, USA, September 2002
45. Syu SJ, Chen J (2005) An active space recycling mechanism for flash storage systems in real-time application environment. In: 11th IEEE international conference on embedded and real-time computing systems and applications, pp 53–59
46. Hsieh J-W, Chang L-P, Kuo T-W (2006) Efficient identification of hot data for flash memory storage systems. ACM Trans Storage 2(1):22–40
47. Cho H, Shin D, Eom YI (2009) KAST: K-associative sector translation for NAND flash memory in real-time systems. In: Design, Automation & Test in Europe (DATE) 2009, pp 507–512
48. Chu Y-S, Hsieh J-W, Chang Y-H, Kuo T-W (2009) A set-based mapping strategy for flash-memory reliability enhancement. In: Design, automation & test in Europe (DATE) 2009, pp 405–410
49. Liu Z, Yue L, Wei P, Jin P, Xiang X (2009) An adaptive block-set based management for large-scale flash memory. In: Proceedings of the 2009 ACM symposium on applied computing, pp 1621–1625
50. Chang Y-H, Hsieh J-W, Kuo T-W (2010) Improving flash wear-leveling by proactively moving static data. IEEE Trans Comput 59(1):53–65
51. Chiang ML, Lee Paul CH, Chang RC (1999) Using data clustering to improve cleaning performance for flash memory. Softw Pract Exp 29(3):267–290
52. Chiang ML, Chang RC (1999) Cleaning policies in mobile computers using flash memory. J Syst Softw 48(3):213–231
53. Kim HJ, Lee SG (1999) A new flash memory management for flash storage system. In: IEEE COMPSAC computer software and applications conference, pp 284–289
54. Lasser M (2003) Method of fast wake-up of a flash memory system. United States Patent, No 6,510,488