A Survey of Address Translation Technologies for Flash Memories

DONGZHE MA, JIANHUA FENG, and GUOLIANG LI, Tsinghua University

Flash is a type of Electronically Erasable Programmable Read-Only Memory (EEPROM). Different from traditional magnetic disks, flash memories have no moving parts and are purely electronic devices, giving them unique advantages, such as lower access latency, lower power consumption, higher density, shock resistance, and lack of noise. However, existing applications cannot run directly on flash memories due to their special characteristics. Flash Translation Layer (FTL) is a software layer built on raw flash memories that emulates a normal block device like magnetic disks. Primary functionalities of the FTL include address translation, garbage collection, and wear leveling. This survey focuses on address translation technologies and provides a broad overview of existing schemes described in patents, journals, and conference proceedings.

Categories and Subject Descriptors: B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Memory technologies*; D.4.2 [**Operating Systems**]: Storage Management—*Secondary storage*; D.4.3 [**Operating Systems**]: File Systems Management—*File organization*; D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; E.5 [**Files**]—*Organization / structure*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Flash memory, flash translation layer, wear leveling, garbage collection

ACM Reference Format:
Dongzhe Ma, Jianhua Feng, and Guoliang Li, 2014. A survey of address translation technologies for flash memories. ACM Comput. Surv. 46, 3, Article 36 (January 2014), 39 pages.
DOI: http://dx.doi.org/10.1145/2512961

1. INTRODUCTION

Flash memory was invented more than two decades ago [Wikipedia 2012c]. However, flash memories started to become an affordable alternative to traditional disks only recently, and flash technology has started to attract the attention of the industrial and the academic communities.

There are two major categories of flash memories dominating the flash memory market today, namely NOR and NAND, both invented by Dr. Fujio Masuoka at Toshiba in 1980s [Wikipedia 2012c]. The commercial NOR flash product was introduced by Intel in 1988, and Toshiba introduced NAND architecture in 1989 [Tal 2003]. NOR flash provides independent address and data buses and thus allows a single byte to be read or written independently. Many devices use NOR to store and run small programs, such as graphic cards, magnetic disk drives, and the BIOS, exploiting its eXecute In

This work was partly supported by the National Natural Science Foundation of China under Grant Nos. 61003004 and 61272090, National Grand Fundamental Research 973 Program of China under Grant No. 2011CB302206, a project of Tsinghua University under Grant No. 20111081073, Tsinghua-Tencent Joint Laboratory for Internet Innovation Technology, and the “NExT Research Center” funded by MDA, Singapore, under Grant No. WBS:R-252-300-001-490.
Authors' address: D. Ma (corresponding author), J. Feng, and G. Li, Department of Computer Science and Technology, Tsinghua University, Beijing 100084, P.R. China; email: mdzfirst@gmail.com, {fengjh, liguoliang}@tsinghua.edu.cn.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2014 ACM 0360-0300/2014/01-ART36 $15.00
DOI: http://dx.doi.org/10.1145/2512961

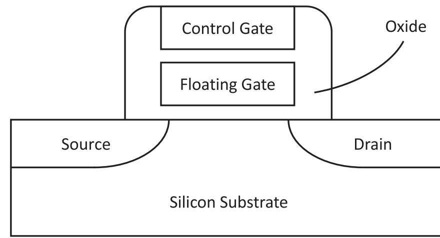

Fig. 1. Structure of a flash memory cell. Basically, it is a standard transistor with an additional floating gate below its control gate. The floating gate is surrounded by an oxide layer so that electrons can be trapped to represent information.

Place (XIP) capability. Although its characteristic makes it a perfect replacement of traditional Read-Only Memory (ROM), NOR suffers from extremely long write and erase latencies. On the other hand, NAND is more like magnetic disks, where address and data share the same I/O interface, and therefore NAND can only be read or written by pages. However, since each storage cell requires less area than NOR, allowing higher density and lower cost per bit, NAND has become a perfect candidate for secondary storage. Besides, the write and erase performance of NAND is improved,[1] and the endurance is also enhanced.

Although some NOR-based technologies are also introduced in this survey, we focus on NAND flash memories because NAND dominates the storage market today.

1.1. Principle of Operations

Flash memory stores information in an array of storage cells that are quite similar to the Metal-Oxide-Semiconductor Field-Effect Transistor (MOSFET). However, unlike the standard transistor, a flash cell has an extra floating gate below the control gate, which is insulated from the control gate by a thin oxide layer. Figure 1 shows the basic structure of a flash cell.

When the voltage between the control gate and the source exceeds a certain threshold voltage V_T, the transistor will conduct. Information stored in a cell is represented by its V_T, which can be modified by adding and removing electrons to and from the floating gate. Once an electron is injected into the floating gate, it is insulated by the oxide layer unless there is a leakage, in which case a single bit error occurs. Modern enterprise-level NAND products can retain static (read-only) data at normal temperature for at least 10 years if the blocks are cycled less than 10 percent of the specified maximum endurance [Samsung 1999; Liu et al. 2012].[2] To determine the amount of charge trapped in the floating gate, one or more voltage levels are applied to the cell and the transistor is sensed to see whether it is conducting.

1.2. Single- and Multilevel Cell Technology

As mentioned in Section 1.1, the state of the flash cell represents the data stored. The Single-Level Cell (SLC) technology defines two states for a single cell and thus can store a single bit of information, whereas the Multilevel Cell (MLC)[3] technology may

[1]NOR has a sightly faster read performance than NAND [Tal 2003].
[2]MLCs tend to have shorter retention time than SLCs because of the smaller difference between voltage levels. There are also some works on improving the performance or endurance of flash-based Solid-State Drives (SSDs) by trading the data retention time (to 1 year or even a few months) [Pan et al. 2012; Liu et al. 2012; Mohan et al. 2012]. They are, however, out of the scope of this survey.
[3]Some manufacturers use Triple-Level Cell (TLC) to describe devices that can store three bits in a single cell. In our classification, MLC includes all devices that are not SLC.

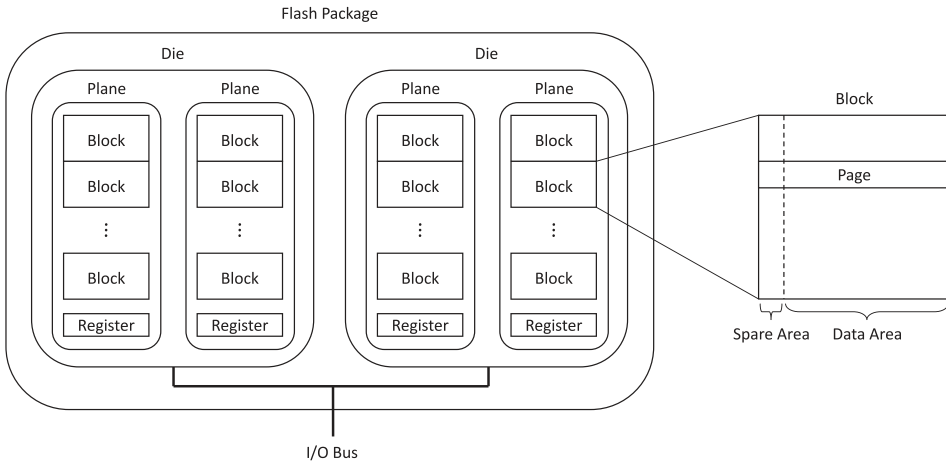

Fig. 2. Flash package organization.

Table I. Organizations of NAND Flash Blocks

	Block Size	Page Size	OOB Size	# Pages/Block
Small-block SLC	16KB	512 bytes	16 bytes	32
Large-block SLC	128KB	2KB	64 bytes	64
Large-block MLC	512KB	4KB	128 bytes	128

store more bits by defining more voltage levels for a single transistor [Dan and Singer 2003; Super Talent 2008]. Generally speaking, to store k bits in a single cell, 2^k voltage levels need to be supported by the transistor.

It is obvious that the MLC technology greatly increases the density of the flash chip and reduces the average cost per bit. These advantages are achieved at the sacrifice of performance and reliability, because it will take more time to place and sense the charge precisely, which implicitly extends the duration of high voltage applied to the transistor, and the closer two adjacent voltage levels are, the more likely errors will happen.

Generally, SLC flash is used in industry-grade devices to provide high performance and stable reliability, whereas MLC flash is often used in consumer products, such as MP3 players, USB flash drives, and flash cards.

1.3. Organization

Figure 2 shows the typical organization of a NAND flash package. A NAND flash package consists of several dies that share the same I/O bus and can operate independently. A die is made up of several planes, each of which owns an independent data register. A plane consists of a constant number of blocks that are the basic unit of erase operations, and a block is further divided into a constant number of pages, which are the granularity of read and write operations. Most flash manufacturers also provide in their products a spare area for each page to store some Out-Of-Band (OOB) data. OOB data includes the Logical Page Number (LPN) or the Logical Block Number (LBN), the Error Detection/Correction Code (EDC/ECC), and the state flags of the page. Typical organizations of NAND blocks are summarized in Table I [Micron 2007; Super Talent 2008; Kang et al. 2007]. (NOR flash can be read and written in bytes and typical block sizes range from 64KB to 256KB [Wikipedia 2012c].)

Basically, new generations of flash products tend to have larger block size to provide high density by reducing gaps between blocks. Larger page size can also help improve the write throughput by allowing more bits to be programmed in parallel.

1.4. Characteristics

In this section, we summarize four special characteristics that make flash memory different from traditional magnetic disks [Norheim 2008; Grupp et al. 2009].

—*No In-Place Update.* Like other Electronically Erasable Programmable Read-Only Memory (EEPROM) devices, a used page in flash memory has to be erased before it can be programmed again, because write operation can only change the state of a bit from 1 to 0.[4] Worse still, flash memory can only be erased in blocks, which is much larger than the granularity of read and write operations. If a single page is to be updated in the in-place manner like in magnetic disk, content of the whole block needs to be copied to Random Access Memory (RAM), updated as required, and written back after the block is erased. This inefficient method will not only affect the endurance of the flash chip due to excessive erase operations but also will bring a potential reliability problem, as if the system crashes after the erase operation, all data that have not been written back will be lost.[5] Therefore, out-of-place updates are usually adopted in flash memory. In other words, when a page is going to be updated, the overwriting data is always put in a newly allocated page.

—*Limit Number of Program/Erase Cycles.* After a certain number of program/erase cycles, some blocks may wear out due to the breakdown of the oxide layer of transistors. The value is usually around 1,000 to 100,000, depending on the type of the memory and the manufacturing technique. For the sake of yield cost considerations, NAND devices are produced with some backup blocks to replace scattered bad ones. Before shipped to consumers, the imperfect device is scanned and invalid blocks are identified and mapped to backup ones. Remaining backup blocks can be used to replace worn-out ones afterward but will be exhausted eventually. Therefore, erase operations need to be distributed across the entire device evenly to prolong the life span of the flash chip. This is also known as the wear leveling technology.

—*No Mechanical Latency.* Unlike magnetic disks, flash memory has no moving parts, such as the motor, the moving arms and heads, and the rotating platters. Therefore, flash memory does not have to suffer the seek or rotation latency. As a result, random accesses of flash memory can be as fast as sequential scans, which makes a distinguished feature compared with magnetic disks.

—*Asymmetric Speed of Read/Write.* Injecting electrons into the floating gate of a transistor always takes longer than sensing its status, resulting in an asymmetric performance between read and write. Traditional applications usually assume that the latencies of read and write operations are almost the same, as is true for magnetic disks, and therefore need further tuning to work with flash memories. Most performance optimizations for flash memory focus on reducing the amount of write and erase operations (including data migration during garbage collections).

1.5. Programming Restrictions

1.5.1. Partial Page Programming. Some flash products support partial page programming. In other words, each page can be divided into several segments, and each segment can be programmed individually [Samsung 1999]. This Number Of Partial (NOP)

[4] Many devices allow a page to be programmed many times. Refer to Section 1.5.1 for more details.

[5] Although some high-end products are equipped with capacitor arrays to provide power-loss data protection [OBrien 2012; Intel 2012] (e.g., write operations [Birrell et al. 2007]), most flash devices only guarantee the atomicity of a single write operation.

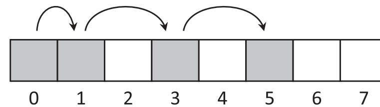

Fig. 3. An example of sequential page programming. Suppose a block consists of 8 pages, and within a program/erase cycle, pages 0, 1, 3, and 5 are used. The only acceptable programming order is the incremental one. If another page of data is going to be written in this block, only pages 6 and 7 can be allocated. Pages 2 and 4 will remain unusable until the whole block is erased.

page programs is usually less than 10 and varies from device to device. Although supporting partial page programming is quite helpful when implementing a journaling file system or a database management system [Lee and Moon 2007], MLC devices no longer support this feature [Dan and Singer 2003]. As a result, some flash-based file systems choose not to use the spare area other than keeping the EDC/ECC field and the bad block flag.

1.5.2. Sequential Page Programming. In SLC flash, pages in the same block can be programmed in any order. This is of vital importance to some Flash Translation Layer (FTL) technologies introduced in this article, since they require that data should reside in offsets corresponding with their addresses. Again, MLC flash disables this feature so that no page has to suffer from program disturbs caused by its adjacent pages twice [Dan and Singer 2003; Samsung 2009; Grupp et al. 2009]. It should be noted that the sequential page programming requires that pages in a block can only be programmed sequentially, and there is no need to use every page. An example following the rule of sequential page programming is given in Figure 3.

2. FLASH TRANSLATION LAYER

FTL is a software layer built on raw flash memories that emulates a normal block device like magnetic disks. Employing some type of FTL is the most straightforward way to deploy a flash-based application. One drawback of this choice lies in that most FTLs are designed for general purposes, and an application can hardly tune its performance. At the same time, some applications choose to work on flash memories directly, realizing the functionalities of FTL themselves. In both cases, address translation is an essential mechanism to all flash-based applications.

2.1. Functionalities

FTL is in essence a combination of several important mechanisms that are unavoidable when deploying a flash device. In this section, major functionalities of the FTL are introduced.

—*Address Translation.* As explained in Section 1.4, whenever a page is to be updated, new data are always put in a newly allocated page. Since most applications are not designed to track the continuous changes of the physical locations of data, a software layer, namely FTL, is usually employed to track the most up-to-date data. Upper applications use logical addresses, which do not change during operations.
—*Garbage Collection.* The update manner of flash memory will definitely generate many invalid data, which need to be erased. FTL is responsible to implement this mechanism, because traditional disk-based applications do not care about garbage collection. Garbage collection consists of three steps. First, a victim block is selected according to some policy. Second, valid pages are copied to a different block. Third, the victim is erased and put in the free list. Modern flash-based Solid-State Drives (SSDs) provide a TRIM command that allows the operating system to inform the SSD that some pages of data are no longer valid due to data deletion [Wikipedia 2012d].

Garbage collection can be carried out in the background or evoked on demand. Some SSD products implement background garbage collection in order to overlap the consequent overhead with other operations [Wikipedia 2012e].

—*Wear Leveling.* Wear leveling techniques try to prolong the life span of flash memories by distributing erase operations across the entire memory. Wear leveling involves many issues, such as how to identify worn-out blocks, which blocks to reclaim, and where to put the valid data.

—*Parallelization and Load Balancing.* Different dies in a NAND flash package have separate chip enable and ready/busy signals and can operate independently without competing for the flash channel, while several (e.g., two) planes in the same die can execute the same type of operations concurrently [Agrawal et al. 2008; Shin et al. 2009]. Therefore, parallel executions of FTL operations, such as address translation and garbage collection, can be exploited. For skewed access patterns, FTL is also responsible to balance the load for the sake of stable performance. RAID techniques, such as striping and dynamic allocation, have been intensively studied, which may help design high-performance FTLs.

—*Worn-Out Block Management.* During read and write operations, some fatal errors may be detected. FTL is responsible to manage this information and prohibit allocating or reclaiming those worn-out blocks.

These functionalities are relatively independent yet are also connected. Address translation may trigger a garbage collection procedure when the number of free blocks drops below a certain threshold, and garbage collection needs the assistance of address translation mechanism to relocate valid data in erase victims. Meanwhile, the address translation scheme that a device uses may also affect the garbage collection algorithm because of specific data layouts. Wear leveling policy, as well as worn-out block management, affects the allocation of free blocks and the victim selection of garbage collection, and, as a result, is related to the two previous components. At last, if parallel processing is supported, all other functionalities need to be carefully designed. Striping-based allocation improves parallelism but will harm the locality ofreference. Shin et al. [2009] provides a profound discussion about this issue.

Among these functionalities, address translation plays the most important role, from which FTL gets its name. It should be noted that the file system community adopts another way to cope with the out-of-place update requirement. Most, if not all, flash file systems (e.g., JFFS [Woodhouse 2001], YAFFS [Manning 2012], LogFS [Engel and Mertens 2005; LogFS Specification 2012], UBIFS [UBIFS 2013]) are log structured. The first log-structured file system [Rosenblum and Ousterhout 1992] was designed to improve the random write throughput of magnetic disks. All updates will be appended at the end of a sequential log, and the metadata structure tracks the segments for each file. The purpose of this mechanism is to perform sequential writes all the time and avoid in-place update, which accords the requirement of flash memory perfectly. Therefore, flash file systems do not employ any independent layer to perform address translation and, as a result, have to face the wandering tree problem. Some of them keep all metadata in main memory to support fast updates and rely on checkpoints to support fast boot-up. In this survey, we focus on existing address translation techniques.

2.2. Architectures

There are two approaches to implement FTL in the system.

In embedded systems, FTL is usually implemented in the file system shown in Figure 4(a), sharing the same CPU where user applications run. As introduced in the previous section, many flash file systems do not employ any independent translation

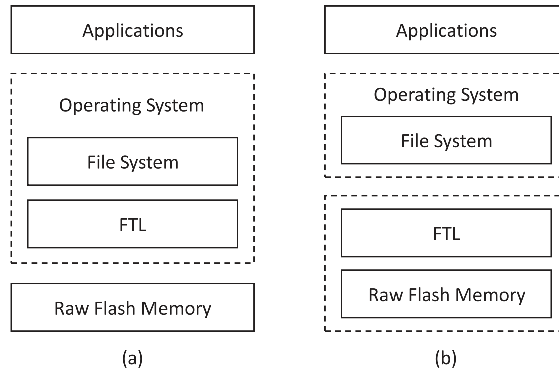

Fig. 4. Architectures of FTL. In the first approach, there is no independent FTL, and the operating system has to implement some important FTL functionalities (e.g., tracking the migration of data due to out-of-place updates). In the second approach, FTL is implemented in the firmware of the device, and existing systems can run atop them through standard interfaces (e.g., Serial ATA).

layer but track data migration themselves.[6] They also have to carry out other FTL functionalities, such as garbage collection and wear leveling.

Alternatively, in removable memory cards or SSDs, FTL is usually implemented in the firmware (Figure 4(b)), which consists of a ROM (to store and run the code of FTL), an SRAM[7] (to store runtime data, mainly the referenced parts of, if not the whole, mapping table), and a controller (to execute the FTL code).

2.3. Data Structures

There are two major data structures to implement address translation: a direct map and an inverse map [Gal and Toledo 2005].

The direct map is the fundamental data structure of an FTL, which maps a logical address to a physical address. The translation process can be as simple as an array lookup, although it may also involve searching a tree. At the same time, a logical address may need to go through several passes of translation to get its corresponding physical address. The direct map usually resides in SRAM or the flash memory itself, but for the sake of efficiency, at least the referenced parts of the direct map should be kept in SRAM.

The inverse map is essentially made up of identifiers of flash pages or blocks, which is always kept in flash memory. When scanning a physical page or block of a flash memory, in order to identify valid data during garbage collection or recover the FTL after a system failure, we can easily get its logical address from the inverse map.

In brief, the inverse map guarantees that we can always identify the logical address of a page or block, and the direct map provides fast address translation. It should be noted that an FTL may not necessarily employ a direct map and that not all FTLs store a complete mirror of the direct map in permanent storage.

2.4. Metrics

Before further introduction, we provide some metrics that are useful to understand the pros and cons of different FTL designs. Some of them depend on the mapping granularity, whereas others rest with the choice of data structures and algorithms.

[6]When using file systems, one accesses data through directory structures and offsets within files rather than device interfaces and addresses.
[7]SRAM is usually used due to its high performance and energy efficiency.

—*Translation Performance.* Performance of address translation depends on the choice of data structures in the first place. Most FTL designs implement their mapping table as an immediate lookup table, which supports fast lookups, whereas a few designs employ some type of search trees, such as a B-tree. Besides, the storage of mapping tables also plays an important role. Obviously, if the mapping table is small enough to reside entirely in the SRAM, operations will be quite efficient. In other cases, the mapping information is stored in the flash memory itself, either in the spare area or in the data area, and I/O operations will be needed when the required parts are not cached in the SRAM.

—*SRAM Overhead.* SRAM is a valuable resource to store at least part of the mapping table. Some FTLs are designed for small devices and can afford to keep the entire mapping table in the SRAM, whereas others keep the entire mirror in flash memory and cache the frequently referenced parts in the SRAM. The performance of the latter is only acceptable when the operations present high locality. Meanwhile, the amount of mapping information of variable-length mapping schemes depends on the particular workload, leaving the SRAM overhead more unpredictable.

—*Block Utilization.* Block utilization is defined as the average number of pages programmed before a block is erased. Block utilization has a direct influence on the number of erase operations, which affect the performance of the FTL as well as the life span of the device.

—*Garbage Collection Performance.* During garbage collection procedures, valid pages in the victim have to be moved to a different block. FTLs that are capable of separating hot data from cold ones may take an advantage by reducing the movement of cold data. Again, this affects the performance and the life span. Moreover, FTLs have to find a way to update their mapping information efficiently to reflect the movement of data.

—*Fault Tolerance.* For some embedded and mobile devices, it is highly possible that the power is cut off suddenly. The ability of recovering a correct state is very important to FTLs, because everything in the SRAM will be lost in case of failures. Some FTLs rebuild the whole mapping table during initialization, whereas others rely on an unreliable mirror in the flash memory. It is necessary for the latter type to identify lost updates and apply them to the materialized but out-of-date mapping table.

2.5. Classifications

Most early FTL designs are for NOR-type flash only, which is widely used to replace older on-board chips and also makes the basis of early flash-based removable media, such as CompactFlash [Wikipedia 2012a]. Therefore, some of the early FTLs may not work with NAND because NAND-type flash is not byte addressable.

In the early 2000s, NAND started to dominate the market [Samsung 2003], and recent FTLs are mostly designed for NAND. NAND is usually treated as a permanent storage device for user data, whose capacity can be hundreds of gigabytes; at the same time, the capacity of NOR merely ranges from 1 to 32MB [Tal 2003]. As a result, recent designers pay more attention to the scalability and update performance of their designs.

According to the granularity of the mapping unit, existing FTL designs can be divided into several categories: page-level, block-level, hybrid, log-based hybrid, and variable-length mappings [Gal and Toledo 2005; Chung et al. 2006]. Both page- and block-level schemes exist for NOR and NAND, and other granularities are mainly for NAND only.

As the name indicates, a page-level FTL maintains an entry for each page and is thus able to translate an LPN directly to a Physical Page Number (PPN). This design is quite flexible and efficient because it allows a page to be allocated to almost any position in the memory, and at the same time, cold and hot (a.k.a. static and dynamic [Wikipedia

2012e]) data can be easily separated. This separation is very useful to control write amplification [Wikipedia 2012e] and improve the performance. Hot data tends to be updated more frequently than cold data. To reclaim the space occupied by out-of-date data (usually hot ones), valid data (usually cold ones) located in the same block need to be copied to another place. If all pages in the block have been invalidated, which is common when hot data are collected, only a single erase operation is needed. One drawback of the page-level FTLs is that the size of the mapping table is relatively large compared with other granularities. Some page-level FTLs keep the entire mapping table in the flash memory and load the referenced parts into the SRAM dynamically.

Different from page-level FTLs, a block-level FTL first divides an LPN into an LBN and an in-block offset.[8] The LBN is translated to a Physical Block Number (PBN), and finally the physical block and its replacement blocks are searched to locate the required page. Obviously, block-level FTLs bring quite small SRAM overhead,[9] but the hot and cold data are hardly separated.

Hybrid mapping FTLs try to take the advantages of both page- and block-level mapping schemes by applying page-level mapping to some data and block-level mapping to others. This scheme can be further divided into two categories. Some hybrid FTLs treat the two granularities equally and pages are directed according to their access pattern or hotness, whereas other hybrid designs treat the fine mapping area as a buffer of modifications to the coarse mapping area and these updates are periodically merged with the original data. Update buffers in log-based hybrid mapping FTLs that employ page-level mappings are usually called the Log Block Area (LBA), and the block-level mapping area is referred as the Data Block Area (DBA).

It is also possible to adopt a variable-length granularity, which can be adjusted according to the access pattern. Since the mapping units do not have an identical size and may not be aligned to the boundaries of blocks, variable-length mapping FTLs can only organize their mapping tables in form of a search tree, such as a B-tree.

In the rest of this survey, we will provide a brief introduction of existing address translation schemes with different granularities.

3. EARLY FTL DESIGNS FOR NOR

3.1. SRAM-Based FTL

The simplest page-level FTL design is to keep in the SRAM the entire mapping table, in which each element addressed by LPNs contains the corresponding PPN [Wu and Zwaenepoel 1994; Estakhri and Assar 1998].

Each time a logical page is updated, a free page is allocated to accommodate the new data and the SRAM-based mapping table is updated, as shown in Figure 5. It should be noted that update of LPN 1 (see Figure 5(b)) is written to a different block from the old data as well as the adjacent logical page, showing high flexibility and efficiency of the page-level design.

When a physical block, say the first block in Figure 5(b), is selected as the victim of garbage collection, all valid pages that it contains are copied to a different block as if they are updated (Figure 5(c)), and then the victim can be erased.

There are two things to be considered about this FTL: scalability and reliability. First, early NOR chips usually have a low density, and it is not a big problem to store the mapping table in SRAM. Second, to protect the mapping table, eNVy [Wu and

[8]The LBN is not always calculated from an LPN. See Section 3.8 for more details.

[9]Consider a 1GB large-block SLC flash with 2KB pages and 128KB blocks (as shown in Table I). Each address is 4 bytes wide. A block-level mapping table takes only 1GB/128KB * 4 bytes = 32KB, but a page-level FTL requires 1GB/2KB * 4 bytes = 2MB to store the mapping table, just 64 (number of pages in a block) times the size of a block-level mapping table.

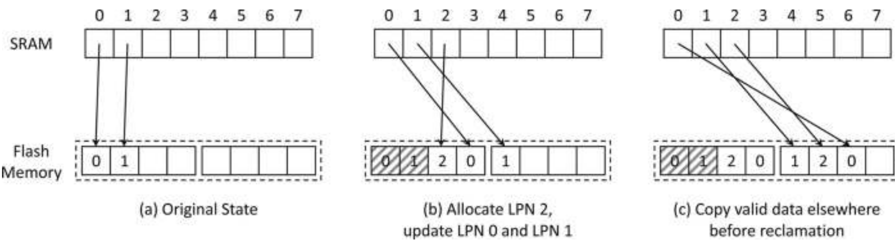

Fig. 5. SRAM-based FTL. The page-level mapping table is stored entirely in the SRAM. It should be noted that the logical address space does not have to be divided into logical blocks, as a logical page can be placed anywhere in the device.

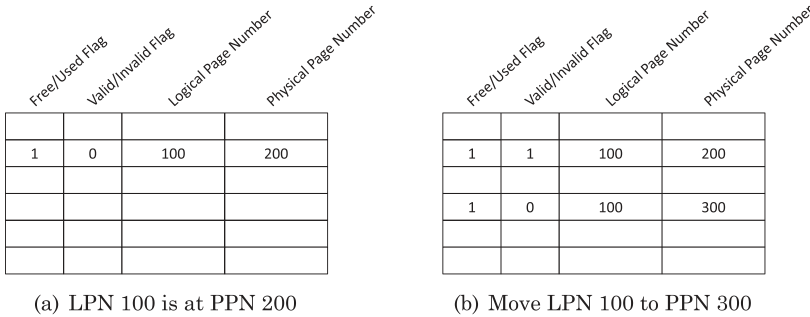

Fig. 6. CAM-based translation table. The two flag fields are used to indicate the state of an entry, and the LPN field represents the logical page to which an entry belongs. All of these three fields are concatenated together to serve a lookup request.

Zwaenepoel 1994] equips a battery to the SRAM, allowing the mapping table to live even after system shutdown. Different from eNVy,Estakhri and Assar [1998] choose to recover the mapping table after system boot-up by scanning all pages in the flash memory. Considering the small capacity of NOR flash, the latency that it causes should be acceptable.

Although quite simple, this design illustrates the basic mechanism about how FTL works.

3.2. Content Addressable Memory-Based FTL

Early designers also suggest to use nonvolatile Content Addressable Memory (CAM) to store the translation table, addressed by LPNs and their corresponding flags [Assar et al. 1995].

Unlike RAM, CAM is designed to determine if a data word supplied by the user exists anywhere in the memory. If the word is found, a list of addresses where the word can be found is returned [Wikipedia 2012b].

As shown in Figure 6, each entry in the CAM-based translation table consists of four fields: a free/used flag, a valid/invalid flag, an LPN, and a PPN. The free/used flag indicates whether the entry has been used, and the valid/invalid flag indicates whether the entry has been invalidated. Each entry is addressed by the former three fields. To serve a lookup request, the LPN is concatenated with the set free/used flag (indicating a used entry) and the unset valid/invalid flag (indicating a valid entry), forming the lookup key of the translation table. When a page is updated, the address information is written in a newly allocated entry, and the old one is invalidated by setting the

valid/invalid flag. Here, the out-of-date entry is not reused to avoid in-place update, considering that the CAM itself may be formed of flash memory.

It should be noted that the PPN fields are not involved in the searching stage; therefore, they can be ordinary flash memory or another type of EEPROM. The PPN fields are stored explicitly rather than matching the physical address of an entry (output of the CAM lookup) with the PPN of the data page, because if so, any defect in the CAM will prevent the corresponding data page from being used.

Although CAM supports fast lookup, as is well known to all, each CAM cell requires its own comparison circuit to detect a match with the input bit, and the comparison results from all cells of a data word must be combined to form a word match result. Besides, to make things worse, each flash page matches at least one entry in the CAM, which means that the size of the CAM is linearly proportional to that of the flash chip. These requirements do not only affect the integration of circuits but also increase the cost of ownership.

3.3. Lookup by Searching

It is not always necessary to maintain a direct map between LPNs and PPNs. To get the most up-to-date version of a page, we may simply scan the whole flash chip and compare the LPN of each page [Assar et al. 1996]. When a match is found, corresponding flags are examined, and if they indicate a valid page, the data is returned.

It is preferred to have a complete mirror of LPNs and flags in RAM, since RAM is several orders of magnitude faster than flash memory. Modern devices also support parallel operations in different planes; therefore, the translation procedure can be easily parallelized.

Obviously, the latency of linear searching is only acceptable with small capacity. All other schemes employ some kind of direct map to provide fast address translation. Some schemes also rely on searching to find a requested page. However, this search is usually limited to a few blocks or offsets.

3.4. Standard NOR-Based FTL

Ban [1995] patented a page-level FTL for NOR in 1995. This design was later adopted as a Personal Computer Memory Card International Association (PCMCIA) standard [Intel 1998]. The translation process is illustrated in Figure 7.

As other page-level mapping schemes, this FTL maintains an entry for each page, and to provide high scalability, an entire mirror of the global mapping table (GMT) is reserved in flash memory. This mapping table is further divided into several mapping pages, and a secondary map is maintained in the SRAM, tracking the physical locations of each mapping page of the GMT. To speed up mapping table lookups, frequently referenced mapping pages are cached in the SRAM.

To serve a translation request, the input address is divided by the number of entries that a mapping page can hold. The quotient is used to search the secondary map. After the corresponding mapping page is located, the right entry that contains another address is obtained using the residual. At last, this intermediate address has to go through a block-level mapping table before the output physical address is obtained. This translation is similar to the former. The intermediate address is divided by the number of pages in a block. The quotient is used to search the block-level mapping table, and the resulting PBN is concatenated with the residual: the in-block offset.

There are two important issues to implement a page-level FTL that stores its mapping table in flash memory. First, when a page is overwritten, the corresponding mapping page in flash memory needs to be updated. If dirty pages are allowed to reside in the SRAM and written back only when swapped out, we will risk losing critical information when the system crashes. If the mapping page is updated immediately, which

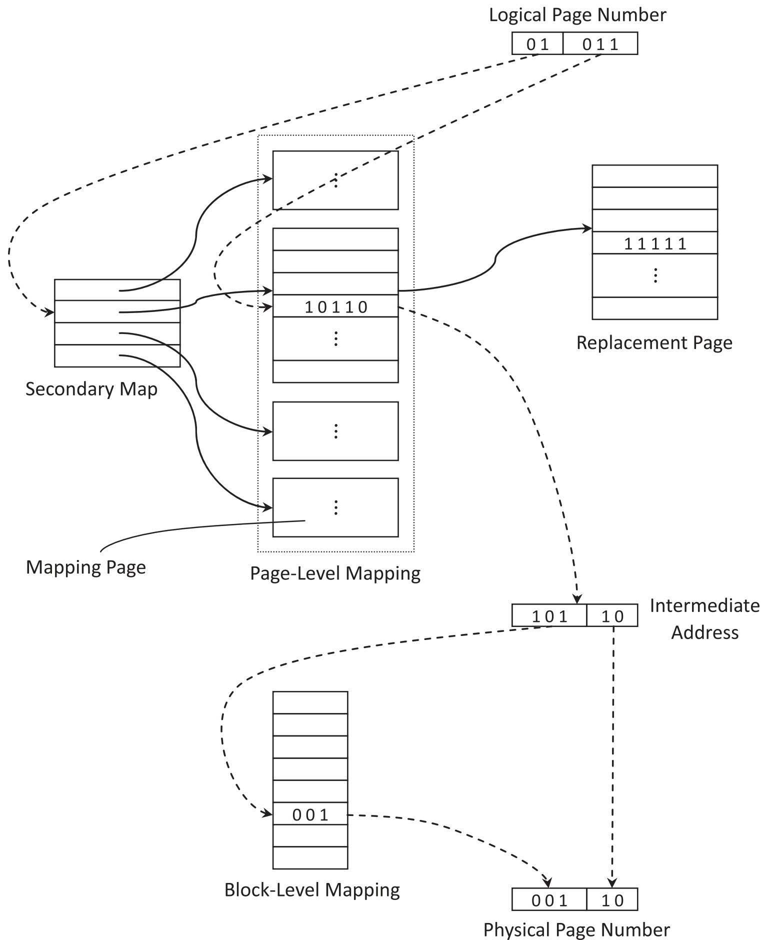

Fig. 7. PCMCIA FTL standard. There are two layers of address translation. A page-level mapping transforms an LPN to an intermediate address, which is then transformed to the final PPN by a block-level mapping. The page-level mapping table is so large to reside in the SRAM that it has to be divided into pages and stored in flash memory itself. Each mapping page is represented by a list of physical pages to support multiple updates. A secondary map is maintained in the SRAM, tracking the migration of page-level mapping pages. The block-level mapping helps improve the performance of garbage collection by hiding the movement of valid data from the page-level mapping layer.

means that the new mapping page is written to a different place and the secondary map is also updated, then performance will be hurt since at least two flash write operations will be needed to serve a single write request or, in other words, write amplification [Wikipedia 2012e] is at least 2. Second, when a block is selected as the victim of garbage collection, all valid pages need to be copied elsewhere, which implies modification of many entries in the mapping table. This operation may bring a tremendous overhead even if the first issue is well settled.

To handle the first problem, a replacement page list is assigned for each mapping page. To update an entry in a mapping page, the corresponding entry of the first replacement page is used, taking advantage of the byte-addressable property of NOR flash. If this entry is modified again, the second replacement page is used, then the third one, and so on. Therefore, each entry in a single mapping page can be updated

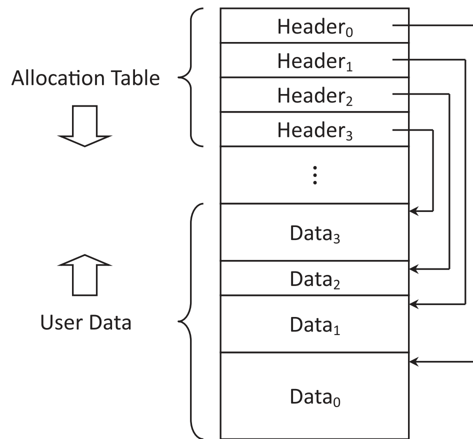

Fig. 8. Variable-sized page. An allocation table is utilized to track the in-block offset of pages, as their sizes are not identical after compression.

many times until the replacement page list is exhausted, at which time a new mapping page is allocated and the old one is invalidated along with the replacement page list.

The second problem is solved by the block-level mapping. Before reclaiming a block, all valid pages are copied to a newly allocated block, keeping the in-block offset unchanged. In this way, only one entry in the block-level mapping table needs to be modified to reflect this operation, and the page-level mapping table, including all replacement page lists, never needs to be touched.

Although this page-level FTL is quite efficient, it is not easy to deploy on NAND flash, as NAND can only be programmed in pages.

3.5. Variable-Sized Page for NOR

Wells et al. [1998] proposed a variable-sized page scheme. This design is essentially a page-level mapping, and the whole mapping table is kept in the RAM like the one introduced in Section 3.1.

The only difference is that pages in the memory are variably sized due to compression. Therefore, an allocation table is necessary for each block. As shown in Figure 8, the allocation table grows from the head of the block, and the user data grows from the back, which is quite similar to the page layout of modern database systems. The allocation table consists of many fixed-length headers, each of which stores the metadata of a page in the block, such as the LPN and the offset.

The mapping table translates an LPN to the PBN where the page locates and the index of the header. If compression is enabled, a block may contain hundreds or even thousands of pages (2,047 at most). Obviously, a single-byte index will not suffice, but a two-byte one will be a waste. In this design, the three least significant bits are ignored, and the rest is stored in a byte. Therefore, at most, eight headers need to be checked when trying to find a logical page.

3.6. Block-Level FTL for NOR: In-Block Logging

One drawback of page-level mapping is that it requires a relatively large SRAM to keep at least the active part of the mapping table. Shinohara [1999] proposed a block-level translation method for NOR flash (Figure 9).

In each block, some pages are reserved to serve the updates of others. When these log pages are exhausted, the whole block is reorganized and copied to a new free block, putting each page at its own offset.

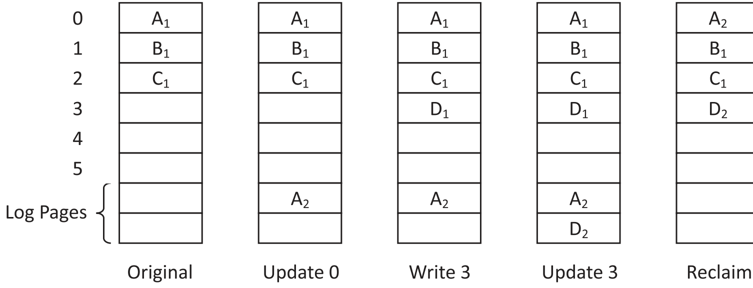

Fig. 9. In-block logging method. In the example, each block consists of eight pages, and the last two are used as logs.

To find a certain page, its matching offset is first examined. If the page has been invalidated, the reserved pages are searched. When an LPN is written for the first time, we put it at its own offset. Future updates are all directed to the reserved log pages.

Obviously, the reserved pages reduce the address space of the memory, since they cannot be shared between blocks and the number of reserved pages in each block should be fixed and identical; otherwise, it will be impossible to calculate the LBN directly from an LPN.

3.7. Block-Level FTL for NOR: Moving Blocks

Estakhri et al. [1999] proposed another block-level translation method for NOR, which was soon updated in Estakhri and Iman [1999]. We only introduce the updated version here.

This scheme is designed to capture any sequential update pattern. A replacement block[10] is allocated for a data block if any page in it is updated, and subsequent sequential updates are directed to the replacement block (Figure 10). To help locate the most up-to-date pages, a bitmap is maintained in RAM for each data block, indicating which pages have been moved to the replacement block.

In case a random update is detected or the update procedure reaches the end of the block, the data block and the replacement block are merged by moving all pages that have not been updated to the replacement block, and the replacement block becomes the new data block, whereas the old one is marked as a candidate for garbage collection.

3.8. A Two-Stage Translation Scheme

Kim and Lee [2002] mentioned a *conventional* mapping method. This method is basically a block-level mapping. Unlike other block-level schemes, the LBN is not obtained directly from the LPN, but by looking up an individual page-level mapping (Figure 11). This design brings an advantage that pages in the same logical block are not always stuck together. When necessary, pages can be reallocated to different logical blocks, even though this operation is not very efficient.

[10]The original paper used the term *moved block*. In this survey, we use the term *replacement block* to indicate a backup block that will likely, if not always, replace the corresponding data block. A similar term is *log block*. A log block is used to hold updated data of the corresponding data blocks temporarily. Data in log block will eventually be merged to the data blocks, although sometimes a log block can also replace a data block. It must be noted that a replacement block can only be owned by a single data block, but a log block may be shared by as many data blocks as the number of pages of a block. On the contrary, a data block may have more than one replacement block or log block.

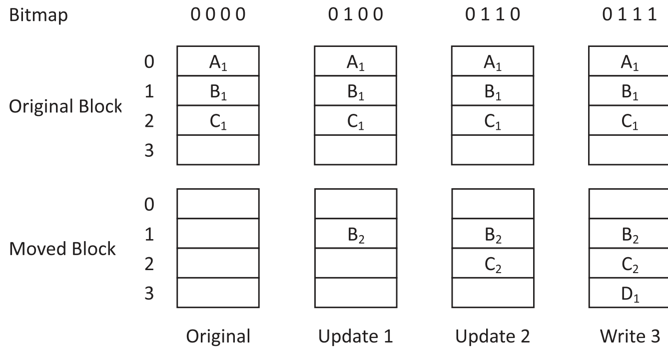

Fig. 10. Moving block method. This method chooses in advance a replacement block to which valid data in the old data block will be moved during garbage collection. Overwriting data are put in this block at the right offset directly so that the amount of data migration can be reduced especially for sequential updates.

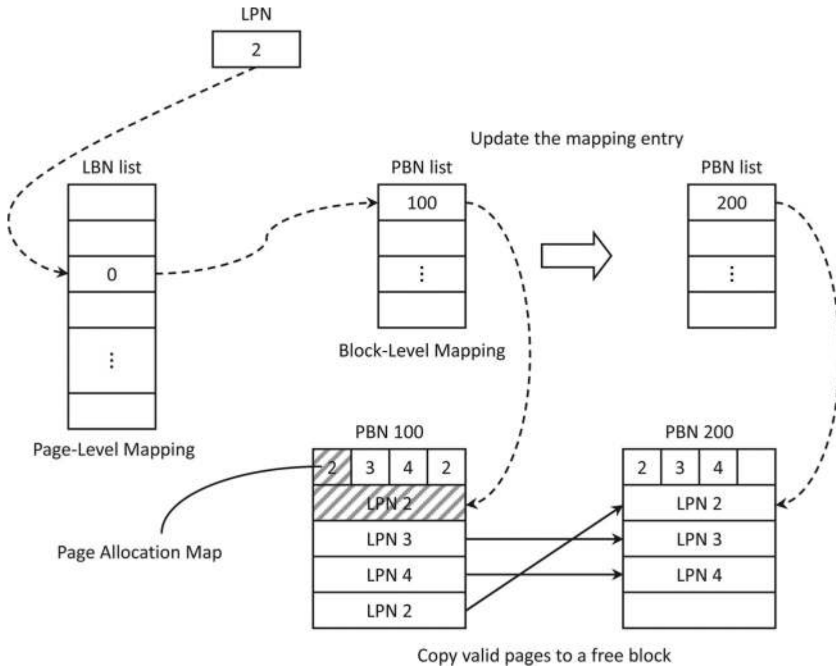

Fig. 11. A conventional mapping scheme. In this method, an LPN is transformed to the corresponding LBN not by simple calculation but by looking up a table. Therefore, it is much more flexible than other block-level mapping schemes. Note that an allocation map is necessary in each physical block since the in-block offsets obtained from the LPNs may conflict with each other. In the example, the block with the PBN 100 is reclaimed, and all valid data are moved to PBN 200.

Acute readers may notice that this FTL is quite similar to the standard FTL described in Section 3.4, without the secondary map and the replacement page list parts. The reason we put this scheme in the block-level category is because the standard FTL sees each page as the mapping unit and reallocates any page that has been updated, whereas the conventional mapping method treats each block as the mapping unit and a updated page is put at a free offset in the original block. A page allocation map is maintained in each physical block, tracking the offsets of each logical page.

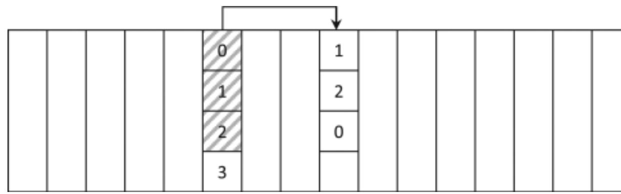

Fig. 12. NFTL with log blocks. Each data block is allocated a log block when necessary, and all updates are written in the log block sequentially. Several spare areas may need to be checked to find the most up-to-date version of a requested LPN.

3.9. Summary

In summary, each byte in NOR can be programmed independently, making it possible to design many types of NOR-based FTLs. Although SRAM is relatively expensive, scalability is not a major concern, and many designs store the entire data structure in the SRAM, since capacity of NOR flash is usually small. The PCMCIA standard FTL (Section 3.4) makes a great step forward toward NOR-based mass storage by keeping the entire mapping table in the flash memory itself. This design, however, cannot be transferred to NAND without modification.

4. BLOCK-LEVEL FTL SCHEMES

Ban [1995] patented two other FTL schemes in 1999. These schemes are designed for NAND-type flash memories and are known as the NFTLs [Choudhuri and Givargis 2007; Gal and Toledo 2005]. Detailed descriptions of these schemes are provided in Sections 4.1 and 4.2.

4.1. NFTL: Log Blocks

This FTL is designed for NAND flash that has a spare area for each page and has de facto become a standard [Micron 2011].

When a page is written for the first time, it is put in the data block where each page is in its own place—in other words, the offset of the physical location is identical to that of the logical address. When the page is updated, we first allocate a log block for the relevant logical block if there is none and write the overwriting pages one after another from the beginning of the log block, as shown in Figure 12.

To serve a read request, NFTL needs to scan all spare areas in the log block in reverse order to find the most up-to-date version of the requested page since pages are written in an out-of-place manner in the log block. If no matching logical address is found in the log block, the corresponding page in the data block is returned. Fortunately, spare areas in NAND-type flash memories are designed to support fast reference, and the overhead of this search process is relatively low.

In case all pages of a log block have been programmed or when this log block is selected for reclamation to create more free space or for wear-leveling reasons, the most up-to-date pages of the relevant logical block are copied from the data block or the log block to a newly allocated block, then the two invalidated blocks are erased and put back in the free block pool. Only in special situations when all pages in a log block are programmed sequentially from the first offset to the last one, could a log block be treated as the new data block directly.

4.2. NFTL: Replacement Block List

On the other hand, since some models of NAND flash memories have no spare areas to support fast search, this NFTL keeps a replacement block list for some of the logical blocks when necessary, and write requests for each logical page are first handled by

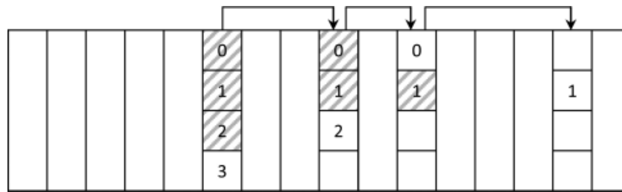

Fig. 13. NFTL with replacement block list. Each data block is allocated a replacement block list when necessary, and all logical pages are kept at their own offsets.

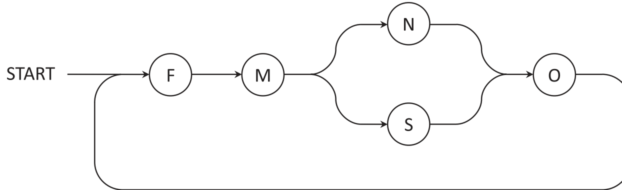

Fig. 14. State transition in STAFF. A free block is initially in F state. After allocation, we try to keep each logical page in its own offset, and the block is in M state. If the block is successfully filled up, its state becomes S; otherwise, the block is in N state (acting as a log block). After the corresponding logical block is reclaimed, all out-of-date blocks move to O state.

the first block in the chain and then the next one, keeping the in-block offset identical with that of the logical address, as shown in Figure 13.

Different from the former design, this NFTL has to scan the replacement block list to serve either read or write requests. For read operations, the last nonfree page with the same in-block offset as the requested LPN contains the most up-to-date data. For write operations, the first free page in the chain is the target place to locate the new page.

If all pages in the list with the requested offset have been programmed, a new block is allocated and appended to the back of the list. When free space declines to a certain threshold or when some replacement block list is too long to search, a logical block is selected as the victim of garbage collection and then all valid pages are copied to the last block in the list. Then, all blocks except the last one are freed.

Note that some manufacturers require that their products should be programmed sequentially within a block during a write/erase cycle, as described in Section 1.5.2. Obviously, this NFTL is not suitable for such products with the sequential page programming restriction, as it is impossible to determine which offset will be used first. However, the first NFTL scheme can be easily deployed on such devices by directing all written data to log blocks.

4.3. A State Transition FTL

Chung et al. [2004] proposed another block-level FTL—State Transition Applied Fast Flash Translation Layer (STAFF)—based on state transition. Five states are defined for physical blocks in this method (Figure 14):

—*F State:* A block in this state is free.
—*M State:* Data pages of a block in this state all reside at their own offset, and this block has not been filled up.
—*N State:* Pages of a block in this state are placed in an out-of-place manner, and this block acts as a log block.
—*S State:* Like M state, pages are written in the in-place manner, but the block has no free page.

—*O State:* A block in this state contains invalid data and needs to be erased before being allocated again.

The basic idea is to decide the page placement according to access patterns. The M and N states act as the replacement/log block in NFTL. The difference is that pages in M-state blocks are written at their own offsets, but pages in N-state blocks do not. Therefore, data in M-state blocks can be retrieved directly, but N-state blocks have to be searched to find the required data.

When a block is allocated, it is in M state, as shown in Figure 14, and all following updated pages are directed to their own offset if possible. Otherwise, the block is turned to N state, indicating that a total reorganization is needed during garbage collection procedure. It should be noted that this FTL always tries to write an updated page in the in-place manner, even in an N-state block.

When an M-state block is filled up, it is turned to an S-state block, whereas a full N-state block is marked as invalid after copying all pages to a new block in a merge operation. This newly allocated block is in M or S state depending on whether it contains free pages. If an S-state block is reached, further updates are served by a new M-state block. STAFF allows a single LBN to be mapped to at most two PBNs (M + S or N + S).

4.4. Summary

Block-level mapping incurs much smaller SRAM overhead than page-level mapping. However, a block-level FTL can hardly separate hot data from cold ones if they share the same LBN. Hot data tend to be updated frequently, which generates lots of invalid pages. In order to reclaim space, valid data, including cold ones, need to be copied to a different place before the old blocks can be erased. This will definitely degrade the garbage collection performance, since cold data have to be moved from time to time, although they are not changed. As far as we know, no block-level FTLs can solve this problem effectively.

5. HYBRID FTL SCHEMES

Hybrid FTLs try to take advantage of the flexibility and efficiency of page-level mapping while keeping the memory overhead as low as block-level mapping. Research of this type of FTLs focuses on how to identify randomly accessed data or hot data.

5.1. Adaptive FTL

Block-level mapping, such as NFTL (Section 4.1), exhibits inferior performance due to high reclamation overhead caused by mixture of cold and hot data, but SRAM-based page-level mapping requires large SRAM capacity to store the mapping table. A hybrid mapping scheme called Adaptive Flash Translation Layer (AFTL) was proposed to balance the advantages of the two granularities [Wu and Kuo 2006].

The flash is divided into two areas. The smaller one employs page-level mapping and the mapping table is kept entirely in the SRAM, providing high lookup performance. NFTL (Section 4.1) is applied to the rest of the memory. It must be noted that NFTL covers the whole address space, whereas the page-level mapping table only tracks selected addresses. Therefore, when accessing an LPN, the page-level mapping should be checked first.

The switching policy of AFTL is quite simple. Whenever a log block in the NFTL area overflows, instead of merging valid pages in the log block with the data block, AFTL assumes that they contain hot data and moves them to the page-level mapping area. If the page-level mapping area overflows, the least recently used page is swapped out and copied to the NFTL area. Recall that NFTL covers the whole address space. The swap-out is done by a normal write operation.

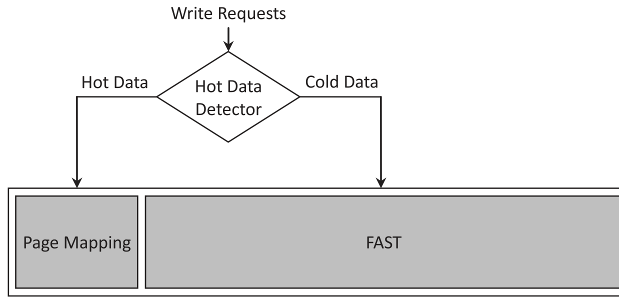

Fig. 15. Architecture of HFTL. Each time a write request arrives, the LPN has to go through a hot data detector. Pages that are identified as hot ones are written in a page-level mapping area, whereas others are served by the coarse-grained mapping area.

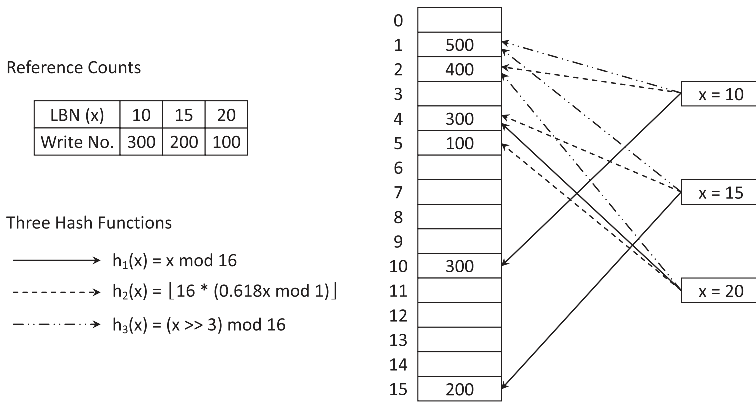

Fig. 16. Hash-based hot data identification. Three hash functions are used in the example. Each maps an LPN to a counter in the hash table. When an LPN is involved in a write operation, all corresponding counters are increased. A page is considered hot if all counters related to its LPN exceed a certain threshold.

5.2. HFTL

Lee et al. [2009] proposed a hybrid FTL scheme named HFTL (Hybrid FTL) that employs a hash-based hot data identification technique [Hsieh et al. 2006] and serves pages that contain hot data with a page-level mapping as long as they remain hot (Figure 15). Others are served by block-level or other (log-based) hybrid mapping schemes.

The hot data identification technique is based on the Bloom filter [Bloom 1970], as shown in Figure 16. The LPN of a page is first mapped to several reference counters by several independent hash functions. Suppose we use three hash functions. Then LPN 10 will be mapped to 10, 2, and 1, LPN 15 will be mapped to 15, 4, and 1, and LPN 20 will be mapped to 4, 5, and 2, respectively. Each time a page is updated, the corresponding counters are increased by one. To distinguish hot pages from cold ones, we only have to test whether all counters relevant to the LPN of a page exceed a predefined threshold. In Figure 16, LPN 10 contains hot data if the threshold is 250, and LPN 15 also contains hot data if the threshold is 150.

The purpose of using multiple hash functions is to reduce the probability of false identification caused by conflicts. In our example, LPN 20 will always be identified as cold page as long as the threshold is set to be larger than 100, although LPN 20 shares two counters with LPN 10 and LPN 15.

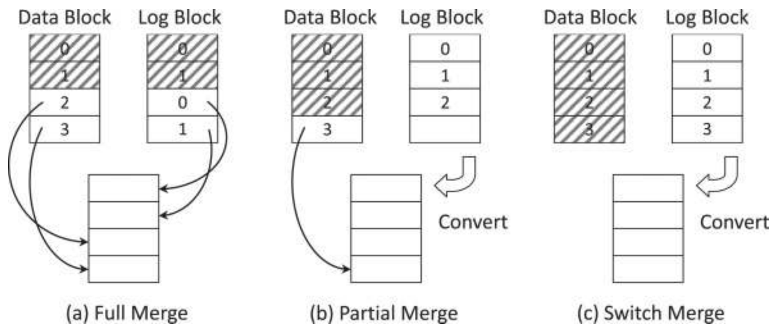

Fig. 17. Three types of merge operations [Ma et al. 2011].

In essence, HFTL can be seen as an improvement of other schemes and is efficient in the circumstance where the hot spot of references is centralized and stable. However, when the access pattern changes, some hot pages will need to be swapped out, which will introduce an extra overhead.

5.3. File System Aware FTL

Instead of detecting hot data passively, Wu et al. [2009] suggested that the FTL should be aware of upper applications, especially the file system. By applying page-level mapping to the file system metadata, which is usually quite small (tens of megabytes), systematic performance can be greatly improved, as the metadata may often need in-place updates (e.g., to update access time or file size).[11]

5.4. Summary

Hybrid FTLs try to make the best of page-level mapping while keeping the translation table small. The key problem is how to identify data that are updated frequently or in a random manner. If the access pattern changes, users may suffer from extra overhead when moving data from the fine mapping area to the coarse mapping area or vice versa.

6. LOG-BASED HYBRID FTL SCHEMES

Log-based Hybrid FTLs divide the flash memory into two major areas: the DBA and the LBA. Block-level mapping is applied to the DBA, which takes a large part of the memory, whereas the LBA is quite small and acts as a buffer of updates for the DBA. Log blocks in the LBA are merged on demand with the corresponding data blocks in the DBA to make room for further updates.

There are three types of merge operations (Figure 17). Generally, a merge operation needs to copy all up-to-date data within the same LBN to a newly allocated block and erase the old data blocks and log blocks. This is called a full merge and is usually quite expensive. Sometimes when a log block is prepared to replace the data block, or in other words, each page in the log block is at the right offset, a partial or switch merge can be carried out. Efforts of designing an efficient log-based Hybrid FTL are focused on the performance of merge operations by avoiding expensive full merge operations whenever possible.

[11]Flash file systems are usually log structured and do not use a fixed location (inode number) to access a file. As a result, no random writes will be involved.

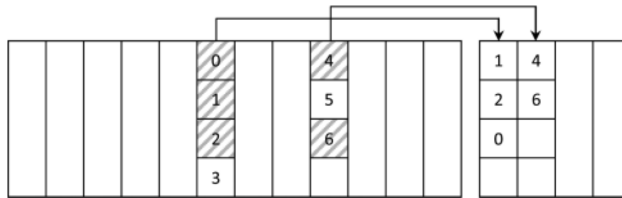

Fig. 18. BAST. The algorithm of data placement is similar to the NFTL introduced in Section 4.1 except that pages in log blocks are tracked by a page-level mapping to support efficient lookup. Each log block monopolizing by a single data block, BAST can easily run out of log blocks even if they are not completely utilized.

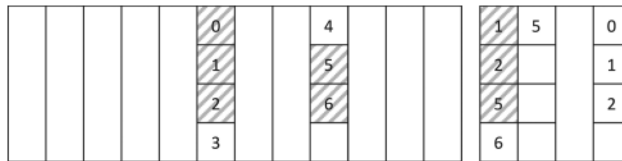

Fig. 19. FAST. This scheme allows a log block to accommodate overwriting data from any data block and thus it can be filled up before reclamation. The worst-case latency of reclaiming a single log block, however, can be much longer than BAST due to unlimited log block associativity. FAST also reserves a log block (the right-most one) to capture sequential updates. Unfortunately, sequential updates can be easily interrupted by other operations, especially in a multiprocess environment.

6.1. Block-Associative Sector Translation

Block-Associative Sector Translation (BAST) [Kim et al. 2002] is the first proposed log-based hybrid FTL scheme, which is similar to the design described in Section 4.1. The only difference is that BAST maintains a page-level mapping for the log blocks and the number of log blocks is limited to reduce the size of the mapping.

Obviously, reading performance of BAST is improved immensely, because looking up a SRAM-reside map is several orders of magnitude faster than searching in flash memory. However, random update patterns can easily exhaust valuable log blocks since BAST does not allow log blocks to be shared (Figure 18). Therefore, BAST is often forced to reclaim a log block that has not been fully utilized. This phenomenon is called the block thrashing problem. Although Wang et al. [2010] suggested to reuse the reclaimed data block in NFTL (Section 4.1) as a contaminated replacement block in order to postpone the erase operation, we argue that this can hardly help because storage systems (e.g., file systems) usually try to allocate a large continuous segment of the address space to a single application, leaving few holes to be reused, let alone whether the flash memory supports out-of-order programming of pages within a block.

To make things worse, it is almost impossible for BAST to perform a partial or switch merge unless the data in a data block is overwritten sequentially from the very beginning, as BAST always puts updated data to the first free page of a log block.

6.2. Fully Associative Sector Translation

Different from BAST, another hybrid FTL—Fully Associative Sector Translation (FAST)—goes to the other extreme [Lee et al. 2007]. FAST always fills a log block with random updates before allocating another one (Figure 19). As a result, FAST solves the block-thrashing problem incurred by BAST.

Although FAST successfully improves log block utilization and delays garbage collections, performance of reclaiming a single log block may turn out to be worse than BAST. Because FAST allows a log block to be shared by any data block, reclamation of a single log block may involve as many data blocks as the number of pages in a block,

also known as the associativity of a log block. This quantity is a good measure of the overhead to reclaim a log block. The larger, the higher.

In order to take the opportunity of partial or switch merge operations, FAST reserves a sequential log block (the right-most one in Figure 19) and performs in-place sequential updates in it. However, this optimization can be further explored because update streams of different processes are usually interleaved and a single sequential log block is far from enough.

Lim et al. [2010] suggested to give valid pages in garbage collection victims a second chance. FAST reclaims blocks in the LBA in the first in, first out (FIFO) order, and some data are forced to be merged before being overwritten. By moving these valid pages to the end of the LBA, it is possible that they can be updated in the second chance, and thus the merge operation is avoided. However, effectiveness of this modification relies on the relative size of the hot data compared with the size of the LBA. If random accesses dominate the workload, performance can be worse, because the second chance requires an extra copy of valid data and merge operations are incurred more frequently.

6.3. Superblock FTL

In order to avoid the drawbacks of the two extremes (BAST and FAST), a trade-off needs to be made between log block utilization and log block associativity. In other words, log blocks should be shared wisely.

Superblock FTL [Kang et al. 2006] made the first step to this goal by allowing N logical adjacent data blocks (or D-blocks) to share at most K log blocks (or U-blocks). Grouping adjacent data blocks and their log blocks into super blocks enables Superblock FTL to exploit the spatial locality of reference, especially in modern computing environment where the operating system tries to allocate continuous (logical) storage addresses to the same file for the purpose of avoiding fragmentation.

Unlike other log-based hybrid FTL schemes, Superblock FTL does not collect log blocks in an LBA but employs a three-level mapping structure, as shown in Figure 20. The first block-level mapping table is kept in the SRAM, and the lower two are stored in the spare areas of the super block. This design not only increases the difficulty to implement, test, and debug the design but also affects the performance due to OOB maintenance and extra lookups. To make things worse, the size of super blocks is quite limited since there is usually not much room in the spare area.

6.4. Set-Associative Sector Translation

Set-Associative Sector Translation (SAST) [Park et al. 2008] adopts the same idea as Superblock FTL. As shown in Figure 21, N adjacent logical data blocks are grouped into a Data Block Group (DBG), and at most K log blocks are grouped into a Log Block Group (LBG). One advantage of SAST is that it collects all LBGs in the LBA and maintains a page-level mapping in the SRAM for all pages in the LBA. As a result, read amplification of SAST is always minimal compared with Superblock FTL, which has to search the spare areas.

The parameter N aims to take advantage of spatial reference locality, whereas K takes the temporal reference locality into consideration. Both should be determined by the specific access pattern. Choice of N is a trade-off between log block utilization and the overhead of merge operations. On the other hand, although it appears to be advantageous to add more log blocks to an LBG, different DBGs may compete for log blocks if the LBA becomes too full. The situation is quite similar to BAST, where each log block is monopolized by a single data block.

Park et al. [2008] provided a method on how to choose proper N and K according to the trace of a given workload. The problem is that the trace may not accurately

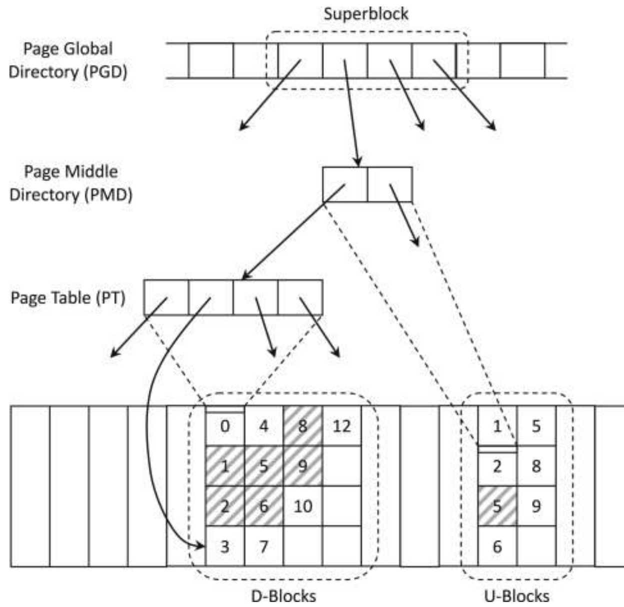

Fig. 20. Superblock FTL. This scheme makes a trade-off between log block utilization and associativity. The flash memory is divided into super blocks. Each super block consists of a few D-blocks to hold user data and a few U-blocks which act as log blocks. Within a super block, U-blocks are shared by all D-blocks. Unlike other log-based hybrid FTLs, Superblock FTL keeps some mapping information in the unused part of the spare area, therefore losing some flexibility and lookup performance.

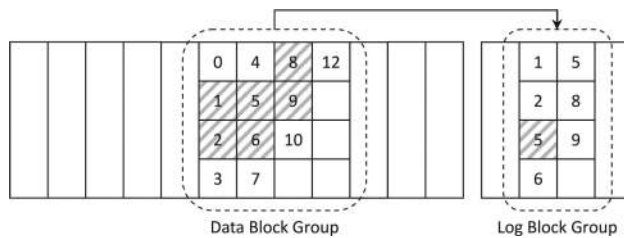

Fig. 21. SAST. Similar to Superblock FTL, SAST also allows a few log blocks to be shared among a group of logically adjacent data blocks. By keeping in the SRAM the page-level mapping information for the LBA, SAST provides better lookup performance.

represent future workload and a general system may run different applications that generate different access patterns. SAST is not flexible enough to these circumstances.

6.5. Adaptive Set-Associative Sector Translation

Koo and Shin [2009] proposed an improved version of SAST called Adaptive Set-Associative Sector Translation (A-SAST). A-SAST does not limit the size of an LBG and allows changing of the size of DBGs adaptively according to the update pattern.

Consider the circumstance in Figure 22, in which four data blocks in the DBG 0 share two log blocks in the LBG 0. Since the update pattern of the DBG 0 is quite random, associativity of blocks in the LBG 0 can hardly be lowered down. In this case, it is better to split the DBG 0 into smaller DBGs and the spatial locality of reference will be enhanced. Moreover, there are only a handful of updates to pages in the DBG 1 and the DBG 2, and the associativities of the corresponding LBGs are quite low (which

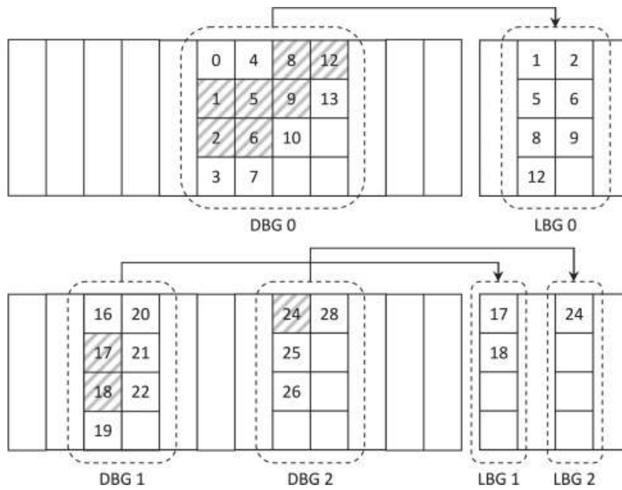

Fig. 22. A-SAST. This is an improved version of SAST. A-SAST adjusts the sizes of data block groups on the fly. In the example, DBG 0 is updated in a random manner. By dividing DBG 0 into smaller groups, the associativity of log blocks can be reduced. In contrast, DBG 1 and DBG 2 are rarely modified. The log block utilization can be improved if data blocks in these two groups are allowed to share the same log block.

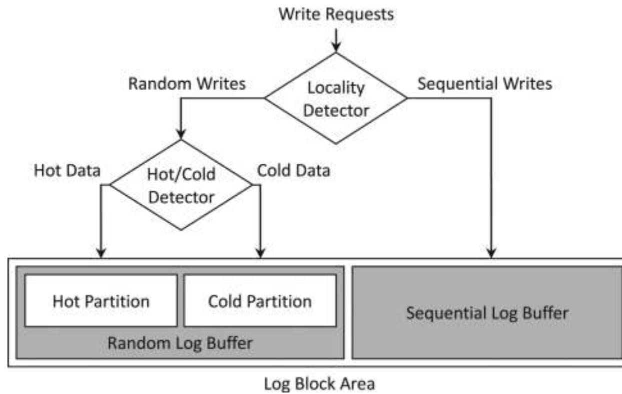

Fig. 23. LBA structure of LAST. LAST diminishes the overhead of merge operations by optimizing usage of the LBA. As shown in the figure, LAST further divides the LBA into smaller regions, each serving a particular kind of workload. Large and sequential updates are carefully placed in the sequential log buffer so that expensive full merge operations can be avoided. Frequently updated data are gathered in the hot partition because they are prone to be overwritten soon, and reclaiming a dead block that contains no valid data merely involves a single erase operation.

indicates high spatial locality). By combining these two DBGs, their log blocks can be shared and the space utilization will be improved.

6.6. Locality-Aware Sector Translation

Unlike Superblock FTL and SAST (including A-SAST), Locality-Aware Sector Translation (LAST) [Lee et al. 2008] does not share log blocks among adjacent logical blocks but directs updates to different regions in the LBA according to their access pattern. As shown in Figure 23, LAST divides the LBA into two parts: a Random Log Buffer (RLB) and a Sequential Log Buffer (SLB). The RLB is further divided into a Hot Partition (HP) and a Cold Partition (CP).

The idea of LAST is very simple. If sequential updates can be separated from random ones and handled properly, opportunities for switch and partial merge operations will be increased. In the SLB, each log block is associated with a single data block like BAST but is organized in the in-place manner; as a result, switch or partial merge can always be applied to the data block. But in the RLB, a log block can accommodate pages from any logical block like FAST, allowing the log blocks to be fully utilized. In order to identify sequential updates, LAST employs a very simple spatial locality detecting policy. If the size of the update request exceeds a certain predefined threshold (e.g., 4 or 8KB), it is considered a sequential update and will be served by the SLB; otherwise, the data will be written into the RLB.

LAST also tries to exploit the temporal locality of reference by collecting hot data in the HP of the RLB. When a full merge is performed against a log block in the RLB, all data blocks that correlate with the log block will need to be reorganized. To reduce this overhead, LAST delays the reclamation of log blocks in the HP, as pages in these blocks are hot and expected to be overwritten in the near future and thus the associativity of these log blocks tends to be lowered down. To identify hot data, LAST manages the HP and the CP as two sequential arrays of pages. Each time an LPN is updated, the duration since its last update is measured. If the duration is within a certain parameter k, this page is considered as hot data and is written in the HP; otherwise, the CP is used.

When the LBA is filled up, LAST employs different reclamation policies according to the functionalities of different segments:

—If the SLB overflows, a least recently used log block is selected as the victim and a switch or partial merge operation is performed.
—If the RLB overflows and a dead block exists (a block carrying no valid data) in the HP, it is reclaimed since only one erase operation is needed.
—If the RLB overflows and there is no dead block in the HP, a log block in the CP that has the lowest reclamation overhead or associativity is chosen as the victim.
—If some cold pages are misled to the HP and prevent the blocks from dying, blocks in the HP that have not been updated or invalidated within a certain period are considered too old and can be reclaimed compulsively.

6.7. K-Associative Sector Translation

Cho et al. [2009] proposed another design direction called K-Associative Sector Translation (KAST), where the systematic performance becomes the secondary goal. KAST is designed for real-time systems and focuses on controlling the response time of a single operation. Basically, KAST is the same as FAST, but the associativity of all log blocks is restricted to be k at most. Therefore, the worst-case latency to reclaim a single log block is expectable.

It must be noted that the parameter k should be chosen carefully. If k is too small, KAST will face the block thrashing problem like BAST, and if too large, reclamation overhead of a log block will become out of control and KAST will degrade to the FAST scheme.

6.8. Janus FTL

Janus FTL [Kwon et al. 2010] is another variant of FAST that exhibits some different characteristics as follows:

—The primary drawback of FAST is the worst-case performance of merge operations due to unlimited log block associativity. Janus FTL handles this with garbage collection in the LBA rather than merge operations. In other words, if the merge cost is

too high, Janus FTL moves valid pages in the garbage collection victim to other log blocks rather than reorganizes the data blocks.
—The DBA in log-based hybrid FTLs usually employs block mapping schemes, which means that each LPN owns its particular page in the DBA. However, since some data are served by the LBA, their corresponding position can never be used by other LPNs, resulting in holes in the DBA. Janus FTL moves this kind of data block to the LBA to allow those holes to be utilized.

Janus FTL introduced two operations: *fusion* and *defusion*. When a write request arrives, Janus FTL detects the update pattern first. Sequential updates are served by sequential log blocks, as FAST does. And for random updates, Janus FTL puts the overwriting data in a normal log block and starts a *fusion* procedure. The *fusion* operation converts a data block to the LBA, which requires no data movement.[12] The rationale behind that is to create an opportunity to reuse the free pages in the data block, because pages in the LBA are tracked independently. When the LBA becomes full, Janus FTL chooses a victim and decides whether to merge it (*defusion*) or reclaim its space according to the associativity.

Unlike other schemes described in Section 6, Janus FTL allows data to be moved between the DBA and the LBA in both directions. The reason we classify Janus FTL as a log-based hybrid scheme lies in that all writes are served by the LBA and the *fusion* operation aims to reuse free pages in the DBA rather than because the data are recognized to be hot.

6.9. Summary

Log-based hybrid FTL schemes also employ page-level mapping and block-level mapping at the same time. However, page-level mapping is subordinate to block-level mapping and acts like an update buffer. Research efforts on this topic mainly focus on how to reduce the need for full merge operation, which is usually slow.

7. VARIABLE-LENGTH FTL SCHEMES

Variable-length mapping means that the granularity of the mapping table is neither blocks nor pages, still less the hybrid of them, but a continuous part of the address space that can expand or shrink along with the changes of access patterns.

7.1. Variable FTL

As far as we know, Chang and Kuo [2004] proposed the first variable-length FTL scheme, which will be cited as Variable FTL in this survey.

The basic mapping unit of Variable FTL is called Physical Clusters (PCs). A PC consists of a set of physically continuous pages in the flash memory and may be in the state of any combination of *free* (F) or *live* (L) and *clean* (C) or *dirty* (D):

—An FCPC (free and clean PC) means that all pages in the PC are clean and can be allocated for newly written data.
—An FDPC (free and dirty PC) means that all pages in the PC contain invalid data and can be simply erased.
—An LCPC (live and clean PC) means that all pages in the PC contain valid data.
—An LDPC (live and dirty PC) means that some pages in the PC contain valid data and some contain invalid ones. New space could be generated by reclaiming an LDPC.

[12]The page-level mapping table needs to be updated.

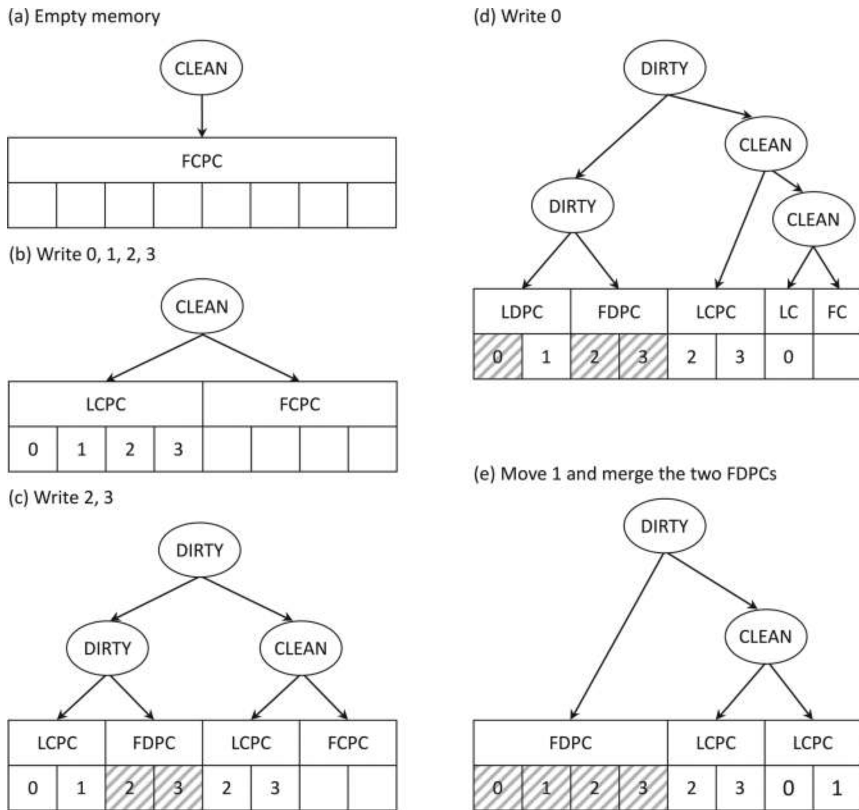

Fig. 24. Variable FTL. The flash memory is divided into variable-sized clusters. A tree structure is maintained in the SRAM to represent the state of each cluster. Clusters in a subtree rooted at a dirty node can be reclaimed to generate free space.

Information of all PCs is organized in a tree structure in the SRAM, as shown in Figure 24, and each internal node has a property of *clean* or *dirty*. A subtree rooted at a *dirty* node can be considered as a reclamation candidate.

To serve a write request, Variable FTL first gets an FCPC that is larger than or equal to the request size, writes the new data at the very beginning, and generates a new LCPC (Figure 24(b–d)).

When pages in an LCPC are updated, the LCPC is turned into an LDPC (Figure 24(d)) or an FDPC if all pages in it have been updated.

To reclaim space from an LDPC, it is split into several LCPCs and FDPCs, and the FDPCs can be erased. Sometimes, data in an LCPC is moved to a new location and adjacent FDPCs are combined into larger one in order to serve large write requests (Figure 24(e)).

The clean/dirty flag of an internal node indicates whether the subtree rooted at the current node can be involved in garbage collection procedures. After an FCPC is split, the newly generated internal node is marked as clean since its two descendants are either FCPC or LCPC. When an LCPC or LDPC is split, the newly generated internal node is marked as dirty since part of the PC is updated and might be reclaimed.

Mapping tables of other types of FTL schemes can be organized in simple address arrays since the mapping granularity is fixed. Variable FTL, instead, employs a hash-based mapping approach (Figure 25). First of all, the logical address space is equally

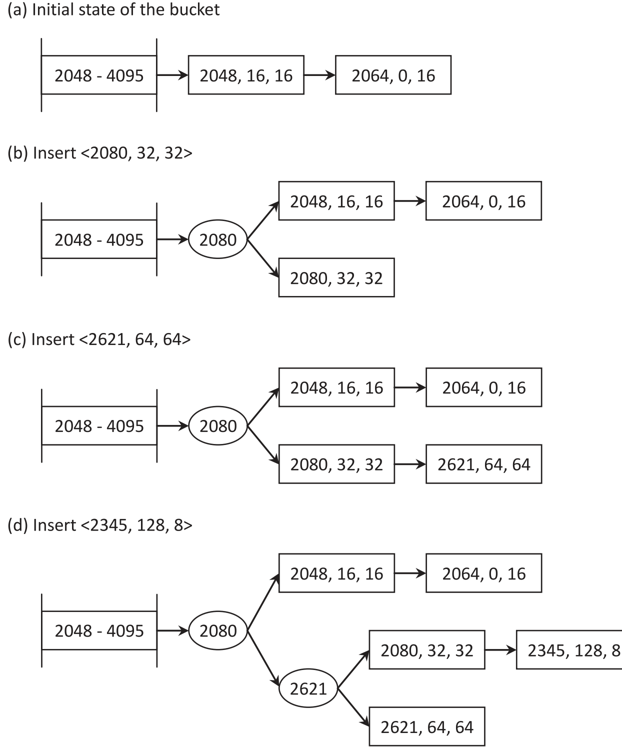

Fig. 25. Address translation of Variable FTL. Without a consistent size of the mapping units, Variable FTL cannot implement its mapping table as a simple address array. Instead, Variable FTL divides the whole address space into equal-sized regions and links the mapping information in each region together. If a linked list becomes too long to search, Variable FTL cuts it into segments and reorganizes them in a binary search tree.

divided into several regions, and mapping entries in each region are linked together, forming a bucket. A mapping entry is a triple containing the starting logical address of the PC, the starting physical address of the PC, and the size of the PC in pages. When the number of entries of a bucket exceeds a certain threshold, the bucket overflows and is split into two subbuckets according to the value of the starting logical address of the mapping entries. In other words, buckets in one region are organized in a binary search tree. Note that entries in the mapping table can be split and merged along with their relevant PCs.

7.2. μ-FTL

μ-FTL [Lee et al. 2008] is another FTL scheme that employs the variable-length mapping. The most important difference between μ-FTL and Variable FTL is that μ-FTL employs μ-Tree [Kang et al. 2007] as the structure of the mapping table.

μ-Tree is a variant of B-Tree that is able to store all pages on the path from the root to any leaf in a single page (Figure 26). Therefore, although μ-Tree is a tree structure linked by pointers, the update of a leaf node only incurs a single write operation.

As shown in Figure 26, leaf nodes of a μ-Tree occupy only half of a page, and along a path from a leaf to the root, the size of each node except the root is reduced by half compared to its direct descendant. Size of the root node is the same as its descendant. As a result, each path from the root to a leaf is fit for a single page. Although update of

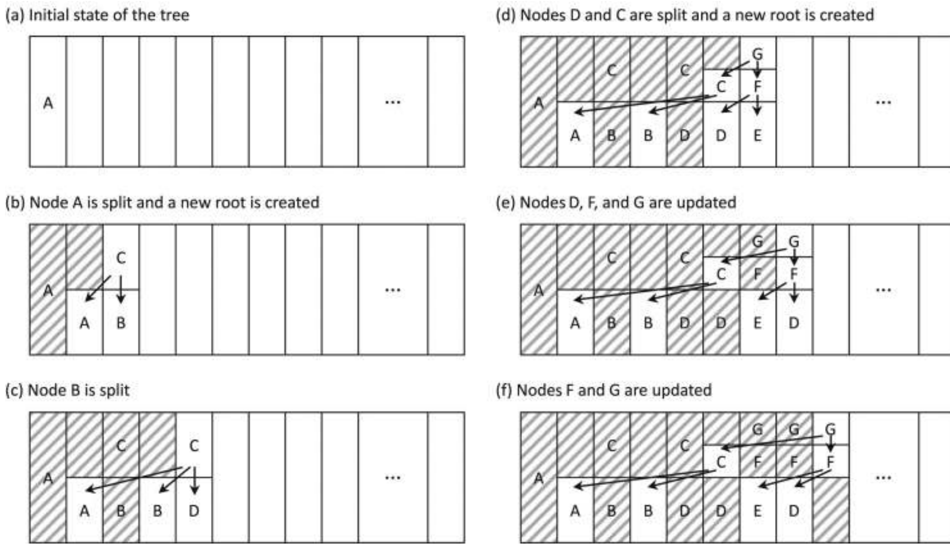

Fig. 26. μ-Tree: address translation structure of μ-FTL. μ-Tree is originally designed to solve the wandering tree problem caused by the out-of-place update requirement of flash memory. By packing all nodes on the path from a leaf to the root in a single page, write amplification is successfully controlled.

a leaf node leads to modifications of the whole path up to the root that contains direct or indirect pointers[13] to the leaf, it can be carried out efficiently by compacting the whole path in a single page.

7.3. Self-Adjusting FTL

Wu [2010] proposed another variable-length mapping FTL called Self-Adjusting Flash Translation Layer (SAFTL). This design is, at the same time, a combination of coarse- and fine-grained mappings.

Similar to other variable FTLs, the mapping unit of SAFTL is a segment of physically continuous blocks in the flash memory, but the content is a little complex (Figure 27). Each entry in the coarse map contains five fields: *LBN*, *PBN*, a *seq* flag, a *free* pointer, and a *swap* pointer. The *seq* flag indicates whether pages in the segment are written sequentially. The *free* pointer and the *swap* pointer point to the head of thefree area and the tail of the swap area, respectively. Note that the swap area grows from the end of the segment.

SAFTL employs a fine-grained mapping. Different from hybrid mapping and log-based hybrid mapping, this fine-grained mapping is not applied to an independent area but acts as a supplement to the coarse-grained mapping area and speeds up random accesses. Each entry in the fine-grained map contains three fields—*LBN, PBN,* and *size*—tracking a piece of out-of-place updates in the same segment.

In order to control SRAM overhead, the fine-grained map is organized as a cache. When it overflows, a victim segment is chosen and all of its mapping entries are collected and packed into pages. These pages are finally written to the swap area of the segment.

[13]Physical addresses are used here.

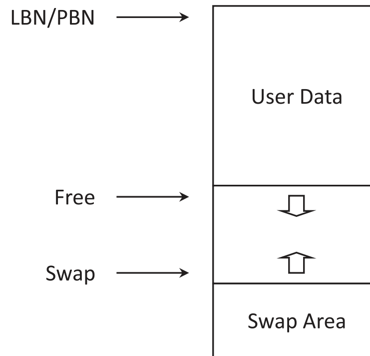

Fig. 27. Segment structure of SAFTL. Segments are the basic mapping unit of SAFTL, which can be very large compared with other FTL designs (e.g., >100MB). Within a segment, user data are appended as logs from the very beginning. A fine-grained mapping is maintained in main memory to support fast lookup. If too much fine-grained mapping information is generated, some of them are swapped out to the swap area of the corresponding segment.

7.4. Summary

Variable-length mapping FTLs try to adjust the granularity of mapping unit according to the access pattern. Unfortunately, the address translation algorithm, the space allocation policy, the garbage collection algorithm, and the implementation of wear leveling are all complicated because the mapping units may not be aligned to block boundaries and the sizes of them are not identical and keep changing.

8. PAGE-LEVEL FTL SCHEMES

Among all mapping granularities, page-level mapping shows the best performance, as it can separate hot and cold data easily, needs no merge operations, and can be implemented in simple data structures, although a page-level FTL requires a large mapping table. This section introduces a few typical page-level FTL designs.

8.1. Demand-Based FTL

Demand-Based FTL (DFTL) [Gupta et al. 2009], a page-level FTL scheme, made the first attempt to applying page-level mapping to NAND flash memories. As shown in Figure 28, DFTL is similar to the standard FTL described in Section 3.4 but without the replacement pages, as NAND can only be programmed in pages.

When a page is overwritten, DFTL modifies the corresponding mapping entry in the SRAM to indicate the physical location change of the page. To minimize the overhead of maintaining the page-level mapping table, DFTL writes a dirty mapping page only when this page is swapped out of the SRAM according to the replacement policy, such as the LRU algorithm. Therefore, DFTL faces a serious reliability problem, because all modified information in the SRAM will be lost if a system failure occurs. In this case, the whole memory needs to be scanned to recover a consistent mapping table.

8.2. LazyFTL

LazyFTL [Ma et al. 2011] is another page-level FTL scheme that aims to provide high performance without giving up consistency.

As presented in Figure 29, besides a DBA and a mapping block area (MBA), LazyFTL reserves two other areas, namely cold block area (CBA) and update block area (UBA), to handle the moved data. The CBA is used to accommodate pages that are moved

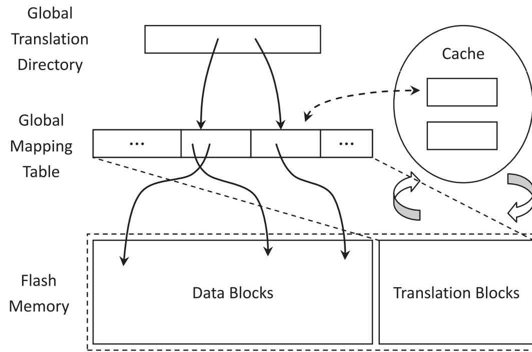

Fig. 28. DFTL. This scheme is very similar to the standard NOR-based FTL introduced in Section 3.4. The whole mapping table is stored in flash memory, and only the referenced parts are cached in the SRAM. Each time a page is modified, DFTL does not try to update its mapping entry in flash memory, but allows the dirty mapping page to reside in the cache until it is swapped out.

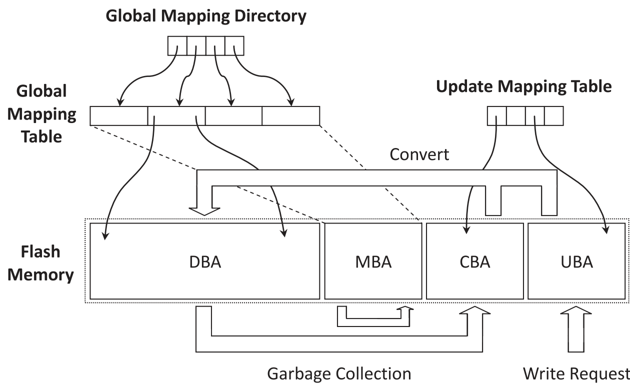

Fig. 29. Architecture of LazyFTL [Ma et al. 2011]. Based on former designs, LazyFTL tries to solve the reliability problem of DFTL. By collecting overwriting data and moved data during garbage collection in two special logical areas (the UBA and the CBA, respectively), LazyFTL can efficiently recover the system to a consistent state after power failures. An independent page-level mapping table is employed for data in the UBA and the CBA. Unlike hybrid mapping schemes, LazyFTL never performs merge operations.

during garbage collection. These pages are considered cold. Updated data are written in the UBA. This is a simple mechanism to separate hot data from cold ones.

The GMT, which is stored in the MBA, is a page-level mapping that tracks all valid data in the DBA. The most frequently accessed parts of the GMT are cached in a small LRU cache in the SRAM. A secondary map named global mapping directory (GMD) tracks the physical locations of different parts of the GMT.

Like hybrid mapping FTLs, LazyFTL maintains an update mapping table (UMT) for the CBA and the UBA. The difference is that in hybrid FTLs, the LBA is only a temporary location to accommodate updates of user data, but in LazyFTL, the CBA and the UBA merely indicate the extent to which the address information of user data might need to be updated to the GMT and there is no merge operation. When the CBA and the UBA overflow, LazyFTL chooses a victim that has the minimal overhead and converts it to a normal data block by updating the address mapping information of its content.

To help indicate the states of pages in the CBA and the UBA, LazyFTL maintains two flags for each page—namely, the update flag and the invalidate flag.

To minimize the overhead of convert operations, we update all translation information that shares the same mapping page as some page in the victim block whose address needs to be updated. Therefore, we need the update flag to indicate whether the translation information of the corresponding page needs to be updated to the GMT. The update flag should be:

—Set when the page is written in the CBA or UBA
—Cleared when the GMT entry is updated

When a page is overwritten, the new data is written in a newly allocated page. At this time, the corresponding mapping page may not reside in the SRAM. Therefore, we do not try to invalidate the old page right now but instead wait until the mapping page is loaded into the SRAM to be updated. The invalidate flag is used to indicate whether the old page that the corresponding GMT entry points to needs to be invalidated. An invalidate flag should be:

—Set when the page is written in the UBA for the first time
—Inherited from the old page during write operations if a validate one is found in the UBA or the CBA
—Cleared when the GMT entry is updated (since the old page is invalidated during the operation)
—Cleared when the old block is reclaimed (since the old page no longer exists)

8.3. Summary

As far as we know, page-level mapping FTLs perform the best against most real-world datasets, as they can separate hot data and cold data easily, do not have merge operations, and can be implemented easily [Ma et al. 2011]. However, if the working set is so large that the corresponding portion of the mapping table cannot reside in the cache, performance will degrade due to frequent swapping operations. Fortunately, most applications do exhibit some extent of access locality. For workloads that have many sequential updates, the mapping table can be efficiently compressed so that more requests can be served by the cached parts of the mapping table [Xu et al. 2012].

9. NEW CHALLENGES

9.1. No Partial Page Programming Support for MLC

One challenge met by researchers is that MLC no longer supports partial page programming. In other words, the spare area and the data area of a page can only be programmed once. Since most existing FTL designs rely on keeping state information of a page in its spare area (through flags), developers are forced to reconsider their basic assumptions when designing new FTL schemes.

One solution to avoid storing this flag explicitly is to keep the whole mapping table in the SRAM like the page-level mapping introduced in Section 3.1 (for efficiency reasons). To determine whether a page is still valid, we only have to look up the mapping table to see whether it is being pointed. Unfortunately, SRAM is a limited resource, and it is not always possible to keep everything in it.Qin et al. [2011] suggested that we should co-locate the changed part of the mapping structure with the updated page, or, more specifically, in the spare area. Superblock FTL (Section 6.3) also employs a similar design. Unfortunately, since the spare area is usually small, not to mention the extra SRAM space required to track the information stored in spare areas, the SRAM saving is limited.

YAFFS 2, a flash file system running on raw flash memories, chooses to keep the states of all (file system level) pages in the system memory [Manning 2012]. Because modern computer systems usually provide a large memory and each page requires

only a bit to store its state, memory requirement is not a major concern for YAFFS 2. When mounting a YAFFS 2 partition, YAFFS 2 may recover the system state from a checkpoint or rebuild it by scanning the whole memory. Since YAFFS 2 does not explicitly invalidate a page, it is necessary to identify the valid data from the out-of-date ones. As a log-structured file system, YAFFS 2 treats the whole memory as a sequential log by assigning monotonic sequential numbers to each of the allocated blocks. In order to rebuild the system state from scratch, YAFFS 2 scans all blocks to determine their sequential numbers, sorts them in the memory, and scans the blocks again in reversed order. Obviously, the most up-to-date version of a page should be encountered first.

9.2. Limited SRAM Resource

As the capacity of flash memory increases, the SRAM has become a precious resource, and researchers have begun to pay more attention to the scalability of their FTL designs.

To reduce the SRAM overhead, Qin et al. [2010] suggested to store the whole mapping of NFTL in the flash memory and cache those frequently accessed pages in the SRAM. DFTL and LazyFTL may also work with relatively small SRAM. However, since page-level mapping tables are much larger than block-level ones, their performance is more sensitive to small SRAM.[14]

A second solution is to reduce the total size of the mapping information by employing larger and variable mapping unit as the designs described in Section 7. Depending on the access pattern, the size of a mapping unit can be hundreds of megabytes, and thus the size of the mapping table is greatly reduced.

Zhang et al. [2012] proposed another method by waiving some design goals of FTLs. As described in Section 2, most FTLs are designed to track the migration of all pages, thus requiring a large mapping table. Zhang et al. [2012] suggested to let the application (e.g., file system) track the positions of data pages and provide a small logical address space (like ordinary FTLs) to maintain metadata so as to prevent recursive update phenomenon. Consider a tree structure that is commonly used in file systems and database management systems. The leaf layer is stored in raw flash, using the physical address as the identifier. When a leaf node is modified, a free page is selected to accommodate the overwriting data. Since its identifier is changed, the pointer in its parent node needs to be updated. Fortunately, all internal nodes are stored in a small logical address space and can be updated in place. Fan-out of tree structures in secondary storage is usually very large (several hundreds or even thousands). As a result, this method can tremendously reduce the amount of data that need to be tracked, and in the meantime, the size of the mapping table. The only drawback is that the device interface needs to be redesigned as well as existing applications.

10. SIDE EFFECTS OF THE FTL ARCHITECTURE

The out-of-place update characteristic of flash memory and the architecture of FTLs bring several opportunities to the system design.

10.1. Atomicity of Write Operations

The atomicity of write operations is very important to a reliable system. Database systems use logs to achieve ACID, and file systems employ journals to ensure consistency (at least for metadata).

[14]For applications that regard performance as a top importance, the flash device can be equipped with a large memory (maybe part of the system memory) to implement page-level address mapping [Birrell et al. 2007].

Update of a single page can be easily ensured to be atomic [Gal and Toledo 2005]. The operation is carried out in five steps: (1) allocate a free page and clear its *free* flag, indicating that this page is no longer free; (2) write the overwriting data to the newly allocated page; (3) clear the *ready* flag of the page, indicating that the data is ready to serve read requests; (4) update the address mapping information to the FTL; and (5) clear the *valid* flag of the page where the out-of-date data resides to invalidate it. If the system fails or the power is cut off suddenly, an update operation can be interrupted at steps (2) and (4), since the other three steps are easily ensured to be atomic by hardware. If an error is detected by the EDC/ECC or the page is not marked as ready, we only have to invalidate the page and ignore its content. If the operation is interrupted before step (5), we identify the old data by comparing the timestamps in the spare areas and invalidate the out-of-date data.

Park et al. [2005] declared that ensuring atomicity of a single write operation is not enough. The authors argued that a file system usually stores the metadata of a single file in multiple pages, such as a tree structure of inodes; as a result, it is better to protect the metadata as a whole. However, this requires bookkeeping a little more information in the metadata and slight modification to the write and garbage collection algorithms.

File systems usually use journal information to enhance consistency, especially for the metadata. Choi et al. [2009] suggested that we should combine the designs of the file system and the FTL. By exploiting the out-of-place update characteristic of FTL and passing file system information, such as deletion of a file, to the FTL, a file system with high consistency level could be implemented without introducing too much overhead.

10.2. Transaction Processing

Besides protecting write operations of a file system, researchers also tried to build transaction processing systems over FTLs, such as LGeDBMS [Kim et al. 2006].

Lee and Moon [2007] proposed In-Page Logging (IPL) and suggested to keep log information in the same block as the corresponding data in order to reduce reclamation overhead. In this design, the last few pages[15] are reserved and divided into several log sectors.[16] In the memory pool, each page is associated with a log sector, and each modification to the page generates a log in the log sector. When a page is swapped out or when its log sector becomes full, the log sector is flushed to one of the log sectors in the same block where the data page locates. If all log sectors are filled up, the whole block is reclaimed. To support transaction processing, IPL discards logs whose transaction has been roll backed, merges logs whose transaction has been committed, and leaves alone logs of unfinished transactions.

10.3. Multiversion Support

System failure is not the only reason we need the out-of-date data. Sometimes, the overwritten data is reserved to provide multiversion support [Sun et al. 2008]. The method is quite straightforward. In an in-place update system, supporting multiversion requires copying the old data to a different place. However, since flash memory always writes updated data to a new page, leaving the old one unchanged, we only have to prevent the history version of a page from being reclaimed. Therefore, besides the effort of tracking the locations of history data, some type of purge policy needs to be embedded in the garbage collection algorithm.

[15]When Lee and Moon [2007] was written, a block contained only 16 pages. Therefore, only 1 page is used to store log information in the original design.

[16]Obviously, IPL requires the support of partial page programming, and the number of log sectors into which each log page is divided depends on the NOP of the device.

10.4. Data Sanitization

Sometimes, the out-of-place update nature of flash memory brings challenges as opposed to opportunities. There are many cases in which data in part of (e.g., files) or the whole storage device need to be sanitized, or, in other words, erased reliably [Wei et al. 2011]. People wish to remove all data before getting rid of (e.g., throwing in the trash or selling to a third party) their storage devices. Before sending their digital devices back to the store for repair, people wish to remove their personal or financial information (e.g., communication records or saved passwords).

There are several methods to purge the whole or part of a traditional magnetic disk, such as overwriting the protected data, using the built-in ATA or SCSI commands, or destroying the device physically (including degaussing) [Wei et al. 2011].

However, due to the use of FTLs, overwriting is no longer a reliable method, especially when only a few files need to be sanitized. For one thing, it is usually useless to overwrite the target file, because the overwriting data will be directed to a different physical address. For another, even if the whole free portion of the logical address space is filled with random pattern, the old data may still survive because the physical address space may be larger than the advertised logical capacity due to performance considerations. To make things worse, the limited life span of flash-based devices also circumscribes the use of overwriting.

Besides, built-in commands are also not reliable for flash-based devices. According to Wei et al. [2011], many existing flash-based SSDs cannot perform the “securely erase” command correctly and need intensive verification.

Since no software method available at hand is completely reliable, the only way to ensure a secure data destruction is to grind one’s flash-based device into fine powder. If one wants to enhance the security of a single file deletion, he will need the assistance of FTL, which owns all of the address information when the file is updated [Wei et al. 2011].

There are also some products that allow users to enable data encryption (e.g., AES). Although encryption can help enhance data security to a certain extent, one still needs to make sure that the encryption key, which is used to access a file, is securely sanitized if he does not want the file to be recovered by an attacker.

10.5. I/O Characteristics

Although characteristics of raw flash memories have been well specified (Section 1.4), flash-based devices, such as SSDs, USB drives, and SD cards, which are integrated with an FTL, turn out to be a black box from the system’s view of point and present different characteristics [Bouganim et al. 2009]. Many of these characteristics can be explained by the design choices of the FTL, and we highlight some of them in this section:

—Read operations are always very efficient, which coheres with the read performance of flash chips. Sequential reads may be slightly faster than random ones if not all mapping information resides in the SRAM.
—Sequential writes can be several times faster than random ones. Random updates may result in holes in the blocks, and many erase operations are needed to reclaim the space. On the contrary, sequential writes can invalidate a continuous segment of pages that needs fewer erases.
—If random writes exhibits high spatial locality, performance may benefit. For one thing, some requests can be served by the write cache; for another, many FTLs, such as AFTL, HFTL, and LAST, treat hot data especially.

As demonstrated in Bouganim et al. [2009], different devices may employ different FTL algorithms, and their performance may vary dramatically. Readers should be careful when designing data structures and algorithms for their devices.

11. CONCLUSIONS

Flash is a promising memory technology that has emerged for tens of years. Since in-place updates are no longer supported, address translation technology becomes an indispensable approach to make flash memory a new layer in the modern storage hierarchy. FTL is a software layer built in the firmware or in the system software that implements address translation, garbage collection, wear leveling, and so forth, and wraps the flash memory into a normal block device like magnetic disks.

Unfortunately, many of these technologies are either undocumented or only described in patents. This survey provides a broad overview of some typical state-of-the-art address translation technologies described in patents, journals, and conference proceedings. We hope that this survey will facilitate future research and development of flash-based applications.

REFERENCES

Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, et al. 2008. Design tradeoffs for SSD performance. In *Proceedings of the USENIX 2008 Annual Technical Conference on Annual Technical Conference*. 57–70.

Mahmud Assar, Petro Estakhri, Siamack Nemazie, et al. 1996. Flash memory mass storage architecture incorporating wear leveling technique without using CAM cells. (January 1996). United States Patent No. 5,485,595.

Mahmud Assar, Siamack Nemazie, and Petro Estakhri. 1995. Flash memory mass storage architecture. (February 1995). United States Patent No. 5,388,083.

Amir Ban. 1995. Flash file system. (April 1995). United States Patent No. 5,404,485.

Amir Ban. 1999. Flash file system optimized for page-mode flash technologies. (August 1999). United States Patent No. 5,937,425.

Andrew Birrell, Michael Isard, Chuck Thacker, et al. 2007. A design for high-performance flash disks. *ACM Operating Systems Review* 41, 2 (April 2007).

Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (July 1970), 422–426.

Luc Bouganim, Björn þór, Jónsson, and Philippe Bonnet. 2009. uFLIP: Understanding flash IO patterns. In *Proceedings of the 4th Biennial Conference on Innovative Data Systems Research*.

Li-Pin Chang and Tei-Wei Kuo. 2004. An efficient management scheme for large-scale flash-memory storage systems. In *Proceedings of the 2004 ACM Symposium on Applied Computing*. 862–868.

Hyunjin Cho, Dongkun Shin, and Young Ik Eom. 2009. KAST: K-associative sector translation for NAND flash memory in real-time systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*. 507–512.

Hyun Jin Choi, Seung-Ho Lim, and Kyu Ho Park. 2009. JFTL: A flash translation layer based on a journal remapping for flash memory. *ACM Transactions on Storage* 4, 4 (January 2009).

Siddharth Choudhuri and Tony Givargis. 2007. Performance improvement of block based NAND flash translation layer. In *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*. 257–262.

Tae-Sun Chung, Dong-Joo Park, Sangwon Park, et al. 2006. System software for flash memory: A survey. In *Proceedings of the 2006 International Conference on Embedded and Ubiquitous Computing*. 394–404.

Tae-Sun Chung, Stein Park, Myung-Jin Jung, et al. 2004. STAFF: State transition applied fast flash translation layer. In *Proceedings of the International Conference on Architecture of Computing Systems*. 199–212.

Raz Dan and Rochelle Singer. 2003. Implementing MLC NAND flash for cost-effective, high-capacity memory. M-Systems White Paper 91-SR-014-02-8L, REV 1.0. (January 2003).

Jörn Engel and Robert Mertens. 2005. LogFS—Finally a scalable flash file system. Retrieved from http://www.informatik.uni-osnabrueck.de/papers_pdf/2005_07.pdf.

Petro Estakhri and Mahmud Assar. 1998. Direct logical block addressing flash memory mass storage architecture. (December 1998). United States Patent No. 5,845,313.

Petro Estakhri and Berhanu Iman. 1999. Moving sequential sectors within a block of information in a flash memory mass storage architecture. (July 1999). United States Patent No. 5,930,815.

Petro Estakhri, Berhau Iman, and Ali R. Ganjuei. 1999. Moving sectors within a block of information in a flash memory mass storage architecture. (May 1999). United States Patent No. 5,907,856.

Eran Gal and Sivan Toledo. 2005. Algorithms and data structures for flash memories. *Comput. Surveys* 37, 2 (June 2005), 138–163.

Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, et al. 2009. Characterizing flash memory: Anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. 24–33.

Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. 2009. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. 229–240.

Jen-Wei Hsieh, Tei-Wei Kuo, and Li-Pin Chang. 2006. Efficient identification of hot data for flash memory storage systems. *ACM Transactions on Storage* 2, 1 (February 2006), 22–40.

Intel Corporation. 1998. Understanding the Flash Translation Layer (FTL) Specification. Technical report AP-864 (December 1998).

Intel Corporation. 2012. Enhanced power-loss data protection in the Intel Solid-State Drive 320 Series. http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/ssd-320-series-power-loss-data-protection-brief.pdf

Intel. 2012. Enhanced power-loss data protection in the Intel solid-state drive 320 series. (2012).

Dongwon Kang, Dawoon Jung, Jeong-Uk Kang, et al. 2007. μ-Tree: An ordered index structure for NAND flash memory. In *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software*. 144–153.

Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, et al. 2006. A superblock-based flash translation layer for NAND flash memory. In *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*. 161–170.

Bum-Soo Kim and Gui-Young Lee. 2002. Method of driving remapping in flash memory and flash memory architecture suitable therefor. (April 2002). United States Patent No. 6,381,176 B1.

Gye-Jeong Kim, Seung-Cheon Baek, Hyun-Sook Lee, et al. 2006. LGeDBMS: A small DBMS for embedded system with flash memory. In *Proceedings of the 32nd International Conference on Very Large Data Bases*. 1255–1258.

Jesung Kim, Jong Min Kim, Sam H. Noh, et al. 2002. A space-efficient flash translation layer for compact flash systems. *IEEE Transactions on Consumer Electronics* 48, 2 (May 2002), 366–375.

Duckhoi Koo and Dongkun Shin. 2009. Adaptive log block mapping scheme for log buffer-based FTL (flash translation layer). In *Proceedings of the International Workshop on Software Support for Portable Storage*.

Hunki Kwon, Eunsam Kim, Jongmoo Choi, et al. 2010. Janus-FTL: Finding the optimal point on the spectrum between page and block mapping schemes. In *Proceedings of the 10th ACM International Conference on Embedded Software*. 169–178.

Hyun-Seob Lee, Hyun-Sik Yun, and Dong-Ho Lee. 2009. HFTL: Hybrid flash translation layer based on hot data identification for flash memory. *IEEE Transactions on Consumer Electronics* 55, 4 (November 2009), 2005–2011.

Sungjin Lee, Dongkun Shin, Young-Jin Kim, et al. 2008. LAST: Locality-aware sector translation for NAND flash memory-based storage systems. *ACM SIGOPS Operating Systems Review* 42, 6 (October 2008), 36–42.

Sang-Won Lee and Bongki Moon. 2007. Design of flash-based DBMS: An in-page logging approach. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. 55–66.

Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, et al. 2007. A log buffer based flash translation layer using fully associative sector translation. *ACM Transactions on Embedded Computing Systems* 6, 3 (July 2007).

Yong-Goo Lee, Dawoon Jung, Dongwon Kang, et al. 2008. μ-FTL: A memory-efficient flash translation layer supporting multiple mapping granularities. In *Proceedings of the 8th ACM International Conference on Embedded Software*. 21–30.

Sang-Phil Lim, Sang-Won Lee, and Bongki Moon. 2010. FASTer FTL for enterprise-class flash memory SSDs. In *Proceedings of the 2010 International Workshop on Storage Network Architecture and Parallel I/Os*. 3–12.

Ren-Shuo Liu, Chia-Lin Yang, and Wei Wu. 2012. Optimizing NAND flash-based SSDs via retention relaxation. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*.

LogFS Specification. 2012. The LogFS Flash File System Specification. Retrieved from http://www.kernel.org/doc/Documentation/filesystems/logfs.txt.

Dongzhe Ma, Jianhua Feng, and Guoliang Li. 2011. LazyFTL: A page-level flash translation layer optimized for NAND flash memory. In *Proceedings of the 2011 International Conference on Management of Data*. 1–12.

Charles Manning. 2012. How Yaffs works. Retrieved from http://www.yaffs.net/documents/how-yaffs-works.

Micron Technology, Inc. 2007. Small-Block vs. Large-Block NAND Flash Devices. Technical Note TN-29-07 (May 2007).

Micron Technology, Inc. 2011. NAND Flash Translation Layer (NFTL) 4.6.0. NFTL User Guide Rev. L. (February 2011).

Vidyabhushan Mohan, Sriram Sankar, and Sudhanva Gurumurthi. 2012. *reFresh SSDs: Enabling High Endurance, Low Cost Flash in Datacenters*. Technical Report CS-2012-05. University of Virginia and Microsoft Corporation.

Hans Olav Norheim. 2008. How Flash Memory Changes the DBMS World. Retrieved from http://www.hansolav.net/blog/content/binary/HowFlashMemory.pdf.

Kevin OBrien. 2012. Samsung SSD SM825 Enterprise SSD Review. Retrieved from http://www.storagereview.com/samsung_ssd_sm825_enterprise_ssd_review.

Yangyang Pan, Guiqiang Dong, Qi Wu, et al. 2012. Quasi-nonvolatile SSD: Trading flash memory nonvolatility to improve storage system performance for enterprise applications. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*. 1–10.

Chanik Park, Wonmoon Cheon, Jeonguk Kang, et al. 2008. A reconfigurable FTL (flash translation layer) architecture for NAND flash-based applications. *ACM Transactions on Embedded Computing Systems* 7, 4 (July 2008).

Sunhwa Park, Ji Hyun Yu, and Seong Yong Ohm. 2005. Atomic write FTL for robust flash file system. In *Proceedings of the 9th International Symposium on Consumer Electronics*. 155–160.

Qhiwei Qin, Yi Wang, Duo Liu, et al. 2010. Demand-based block-level address mapping in large-scale NAND flash storage systems. In *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. 173–182.

Zhiwei Qin, Yi Wang, Duo Liu, et al. 2011. MNFTL: An efficient flash translation layer for MLC NAND flash memory storage systems. In *Proceedings of the 48th Design Automation Conference*. 17–22.

Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-structured File System. *ACM Transactions on Computer Systems*, 10, 1 (February 1992), 26–52.

Samsung Electronics Co., Ltd. 1999. Application Note for NAND Flash Memory, Rev. 2.0 (December 1999).

Samsung Semiconductor, Inc. 2003. Selecting the Right Flash Partner to Turn Technology Advantages into Profits. Position Paper CG2020-A (January 2003).

Samsung Electronics Co., Ltd. 2009. Page Program Addressing for MLC NAND (Version 0.2). Application Note (November 2009).

Ji-Yong Shin, Zeng-Lin Xia, Ning-Yi Xu, and others. 2009. FTL design exploration in reconfigurable high-performance SSD for server applications. In *Proceedings of the 23rd International Conference on Supercomputing*. 338–349.

Takayuki Shinohara. 1999. Flash memory card with block memory address arrangement. (May 1999). United States Patent No. 5,905,993.

Kyoungmoon Sun, Seungjae Baek, Jongmoo Choi, et al. 2008. LTFTL: Lightweight time-shift flash translation layer for flash memory based embedded storage. In *Proceedings of the 8th ACM International Conference on Embedded Software*. 51–58.

Arie Tal. 2003. Two Technologies Compared: NOR vs. NAND. M-Systems White Paper 91-SR-012-04-8L, Rev 1.1 (July 2003).

Super Talent Technology, Inc. SLC vs. 2008. MLC: An analysis of flash memory. White Paper (March 2008).

UBIFS. 2013. The UBIFS Documentation. Retrieved from http://www.linux-mtd.infradead.org/doc/ubifs.html.

Yi Wang, Duo Liu, Meng Wang, et al. 2010. RNFTL: A Reuse-Aware NAND Flash Translation Layer for Flash Memory. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems*. 163–172.

Michael Wei, Laura M. Grupp, and Frederick E. Spada. 2011. Reliably erasing data from flash-based solid state drives. In *Proceedings of the 9th USENIX conference on File and Storage Technologies*.

Steven Wells, Robert N. Hasbun, and Kurt Robinson. 1998. Sector-based storage device emulator having variable-sized sector. (October 1998). United States Patent No. 5,822,781.

Wikipedia. 2012a. CompactFlash. Retrieved from http://en.wikipedia.org/wiki/CompactFlash.

Wikipedia. 2012b. Content-addressable Memory. Retrieved from http://en.wikipedia.org/wiki/Content-addressable_memory.

Wikipedia. 2012c. Flash Memory. Retrieved from http://en.wikipedia.org/wiki/Flash_memory.

Wikipedia. 2012d. TRIM. Retrieved from http://en.wikipedia.org/wiki/TRIM.

Wikipedia. 2012e. Write amplification. Retrieved from http://en.wikipedia.org/wiki/Write_amplification.

David Woodhouse. 2001. JFFS: The journalling flash file system. Retrieved from http://sources.redhat.com/jffs2/jffs2.pdf.

Chin-Hsien Wu. 2010. A self-adjusting flash translation layer for resource-limited embedded systems. *ACM Transaction on Embedded Computing Systems* 9, 4 (April 2010).

Chin-Hsien Wu and Tei-Wei Kuo. 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*. 601–606.

Michael Wu and Willy Zwaenepoel. 1994. eNVy: A non-volatile, main memory storage system. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*. 86–97.

Po-Liang Wu, Yuan-Hao Chang, and Tei-Wei Kuo. 2009. A file-system-aware FTL design for flash-memory storage systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*. 393–398.

Zhiyong Xu, Ruixuan Li, and Cheng-Zhong Xu. 2012. CAST: A page-level FTL with compact address mapping and parallel data blocks. In *Proceedings of the 2012 IEEE International Performance Computing and Communications Conference*. 142–151.

Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, et al. 2012. De-indirection for flash-based SSDs with nameless writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*. 1–16.

Received October 2012; revised August 2013; accepted August 2013