A survey of data recovery on flash memory

Van Dai Tran, Dong-Joo Park
Department of Computer Science and Engineering, Soongsil University, Korea

Article Info

Article history:

Received Nov 28, 2018
Revised Aug 5, 2019
Accepted Aug 29, 2019

Keywords:

Data recovery
Flash memory
FTL
PLR

ABSTRACT

In recent years, flash memory has become more widely used due to its advantages, such as fast data access, low power consumption, and high mobility. However, flash memory also has drawbacks that need to be overcome, such as erase-before-write, and the limitations of block deletion. In order to address this issue, the FTL (Flash Translation Layer) has been proposed with useful functionalities like address mapping, garbage collection, and wear-leveling. During the process of using, the data may be lost on power failure in the storage systems. In some systems, the data is very important. Thus recovery of data in the event of the system crash or a sudden power outage is of prime importance. This problem has attracted attention from researchers and many studies have been done. In this paper, we investigate previous studies on data recovery for flash memory from FTL processing solutions to PLR (Power Loss Recovery) solutions that have been proposed by authors in the conference proceeding, patents, or professional journals. This will provide a discussion of the proposed solutions to the data recovery in flash memory as well as an overview.

Corresponding Author:

Dong-Joo Park,
Department of Computer Science and Engineering,
Soongsil University,
369 Sangdo-ro, Dongjak-gu, Seoul, (06978)-Korea.
Email: djpark@ssu.ac.kr

1. INTRODUCTION

Flash memory is now available in most areas of the world, because of its advantages such as fast data access and low power consumption. It also has some weaknesses including erase-before-write or limited life cycle (e.g., the number of erases is limited around 10000 ~ 100000 times per each block). To address these shortcomings of flash memory, the FTL has been proposed with useful functionalities like address mapping, garbage collection, and wear-leveling. In addition, the recovery of data in the event of the system crash or a sudden power outage is of prime importance. Hence many methods have been proposed to address the data loss when a sudden power outage or power failure occurs like FTL processing solutions or PLR solutions.

FTL [1, 2] has three functionalities including logical to physical address mapping, garbage collection, and wear-leveling. Some FTL algorithms have been proposed to overcome the physical limitations and improve the overall performance of flash memory. Despite the fact that the performance of flash memory has been enhanced by using FTL algorithms, it may experience performance degradation when performing B-tree direct indexing because the activity of flash memory overwrites frequently occurs in the case of update nodes. To solve this problem, some B-tree variants have been proposed. Some of these B-trees use the main memory as buffers to delay updates on the B-tree. Any inserted record is temporarily stored in the buffer. When the buffer is full, all the records in the buffer are transferred to the flash memory. These B-tree variants provide good performance as the number of flash activities decreases. However, there is a risk of losing data if a power outage occurs because many records are in the main memory. Moreover, they consume some of the main memory resources that are limited in different embedded systems.

Many studies on the PLR scheme [3-6] have been proposed to help the storage system avoid losing data. These studies can be classified into 3 groups. The first group, In-Block Backup, reserves certain blocks in flash memory for metadata and stores modified metadata into blocks whenever there is any change. In the event of a sudden power occur, the storage controller locates the metadata area and uses information stored in the area during boot to restore the system state. The second is called In-Page Backup. In this approach, the page which contains the metadata associated with each page is also stored in its backup area. During the restore process, the controller will rebuild the system state by updating the backup area of each page. Finally, Hybrid Backup is a mixture of the above two techniques: it periodically stores a backup of the mapped table in the metadata area and it also copies the map information into the area for the corresponding data page. In fact, these algorithms help the systems reduce the risk of losing data. However, these techniques may cause high costs for storage systems because they require an additional flash program for every mapping update. In addition, they take a long time to reproduce the metadata from the backup area, especially backing up in the page.

This article provides an overview of the techniques and methods proposed to solve the problem of recovering data on flash memory in terms of policies. The rest of this paper is structured as follows: Section 2 reviews the fundamental knowledge of flash memory, FTL, B-tree, and T*-tree. Next, Section 3 is presented by the recovery techniques of flash memory and finally, Section 4 concludes the paper.

2. BACKGROUND AND RELATED WORK

2.1. Flash memory

Flash memory [1, 2] is a non-volatile storage format used today. Unlike traditional hard disk storage, the flash memory contains a range of memories, controls, and SRAMs that are used as input buffers and store link information. NAND flash memory consists of several blocks, each containing a number of pages (for example, 32, 64, or 128). The page is the smallest unit for read and write operations while the block is the smallest unit of erase operations.

Flash memory supports three basic operations: read, write, and erase. Read operation is the fastest one among these operations, which is about 10 times faster than a write operation. The erase operation is very time-consuming; a per-block-erase-time usually takes about 2ms, over 10 times slower than a write operation. Flash memory is a high-density storage device which requires an additional layer of software called the Flash Translation Layer (FTL) [1, 2] to control and manage data to improve the performance of flash memory. There are two types of flash memory: NOR and NAND. NOR flash memory is used in the industry. It provides an easy-to-use and user-friendly interface for code execution and makes a good device without data storage. Since NOR flash memory performance is good at reading, but not good at erasing and writing, it is used as a storage device. However, today's devices are becoming smaller by day, so they are expected to offer more features, and more information storage. Capacity requirements are more important factor than the ability to code and store data, and significantly faster erase times. NAND flash memory provides all the above features and affordable price in capacities. However, engineers do not like to use it because of complex management and non-standard interface. The structure of NAND Flash memory is shown in Figure 1.

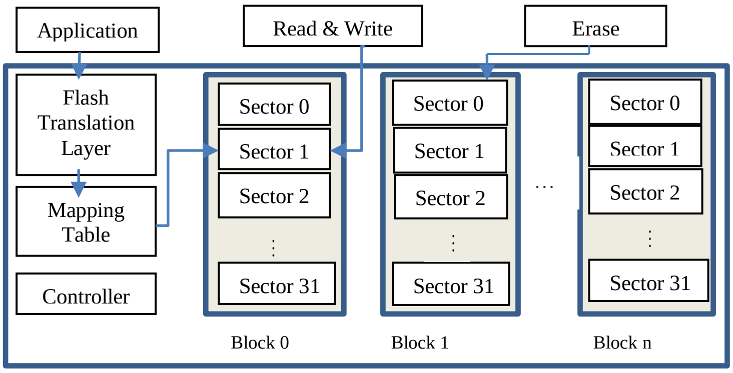

Figure 1. Structure of NAND flash memory

2.2. Flash translation layer (FTL)

The Flash Translation Layer (FTL) is a good environment software module for storing data. It records, and modifies information from an empty page and moves the request to the new address. FTL ensures that the most recent record is distributed across all pages that are likely to be processed. It keeps a list of invalid pages, markers, deletions, and reuse. The main purpose of FTL is to simulate the function of a traditional device block with its characteristic flash memory cleared before recording. It provides the process of converting from logical addresses to physical addresses, garbage collection, and wear leveling when requested [22, 23]. NAND flash memory has a log erase time when performing a block delete operation. With FTL log erase time is transferred when FTL writes data to an available physical page and then marks the data containing the previous page as invalid data rather than deleting the entire block.

One of the main functions of FTL is the address mapping. Basically, it maps logical addresses from the host to physical addresses of flash memory. The basic address mapping algorithms can be divided into three schemes based on the mapping unit: sector mapping, block mapping, and hybrid mapping. Sector mapping is unit-based mapping, mapping unit of block-mapping is a block, and hybrid mapping is to remedy shortcomings of sector-mapping and block mapping. Hybrid mapping has been widely used in large scale systems. The Hybrid policy has shown huge improvements on the performance of FTL. Table 1 shows the comparison of FTL algorithms.

Table 1. Comparison of FTL algorithms [1]

Algorithm	Mapping table size	Address computation overhead	Read cost
Sector mapping	Large	No overhead	Low
Block mapping	Less than Sector mapping	Yes	Low
Hybrid mapping	Same as Block mapping	Yes	High

Metadata is one of the important components of the stored data. It is used to describe what, where, and how data is stored on the medium. The address mapping data in the metadata is usually composed of the logical address number and the physical address number. FTL uses this information to create an address mapping table. So, when the host asks for an operation associated with the data (this data is stored in flash memory) FTL uses an address mapping table entry. You may have noticed that the metadata of address mapping is very important in the FTL algorithm. When a host requests a write operation, FTL chooses one of two scenarios. First, if there is no data in the requested address, FTL writes the data directly to that address. Otherwise, if there is the data in the required address, FTL finds the appropriate address to complete the write operation, and information change is stored elsewhere in the flash memory. If FTL does not save changed information it will be very difficult to find the data when the host wants to find it later. So FTL will store information in flash memory. Figure 2 shows the structure of Flash Translation Layer.

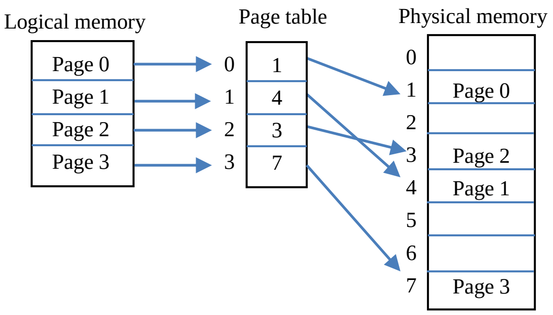

Figure 2. The flash translation layer structure

2.3. B-tree

B-tree [7] is a tree-type data structure that allows searching, sequential access, insertion, and deletion in logarithmic time. B-tree is a generalization of the binary search tree in which a node can have more than two children. Unlike self-balancing binary trees, B-trees are optimized for large read and write systems. They are often used in databases and file systems. In B-tree, non-leaf nodes may have a different number of nodes, limited to a certain range. When data is inserted to or deleted from a node, the number of its child nodes is changed. When using a B-tree, we prescribe an integer t for the minimum degree. Each node in the B-tree (except the root node) has t-1 to 2t-1 keys which ensures that nodes can be split or combined. With an internal node having 2t-1 locks, we can get the middle key, insert it into the parent node, and split the remaining 2t-2 keys into two new nodes, each with t-1 keys. Similarly, when each two nodes has t-1 locks, you can combine these two nodes with a key from the parent node into a new one with 2t-1 keys. The number of child nodes of a node is exactly equal to the number of keys of that node plus one. Therefore, the number of nodes of an inner node is from t to 2t.

B-tree maintains balance by ensuring that leaf nodes have the same depth. Tree height increases gradually as new nodes are inserted into the tree, but this number changes very slowly. As the height of the tree increases, the depth of all leaf nodes increases at the same time. B-tree has the advantage over other search data structures when access time is much larger than consecutive reads. This usually occurs when the nodes are stored on external memory. By increasing the number of children of each node, the height of the tree decreases and the number of hits decreases. In addition, the number of re-balancing operations is reduced. Typically, the parameter t (determines the number of keys per node) is selected according to the amount of information in each key and the size of each disk block. The B-tree term can be used to refer to a specific design, which can also be used to design a class of designs. In a narrow sense, the B-tree stores some locks on the nodes and does not store copies of those locks on the leaf node. In broad terms, it also includes other variants such as B+-tree [8] or B*-tree.

a. In the B+-tree all keys along with accompanying data, are stored at leaf nodes. In addition, leaf nodes also have a pointer to the next leaf nodes to accelerate sequential access.
b. B*-tree performs more rebalancing operations to store denser data. Each node in the root node must be filled to two-thirds instead of only half as in B-tree.

Figure 3 shows the structure of application using B-Tree over Flash memory.

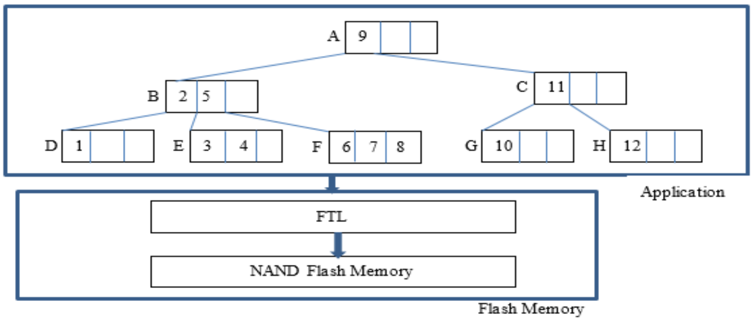

Figure 3. B-Tree on flash memory

2.4. T*-tree index technique

This technique uses T*-tree, where all data is stored in the main memory device unlike the existing disk-based indexing techniques. This index structure modifies defects and allows faster access to data and efficient use of memory, which are advantages of the T-tree index structure used in base systems of main memory data [9]. The structure of T*-tree is shown in Figure 4. In addition, unlike in the T-tree, inserted data items in the nodes of T*-tree are stored in order from the left; so the empty space is always to the left. Each element in the node is configured with a single key value and the address of the data item has a key value, and they are re-ordered according to the size of the item [9].

As shown in Figure 4, the rear cursor is added to the T*-tree pointing at the rear nodes that are configured for successive values. As a result, the T-tree's movements are improved and direct traversing to the next is possible, reducing the length of the movement path. In addition, when the nodes are created, the first item, of the maximum value, is stored from the right in the order. This allows the maximum overflowed value to be stored to the rightmost of the empty space of the LUB (least upper bound) node in case of overflow due to data insertion. In the case of underflow by deletions, the minimum value from the LUB node or the leftmost node is borrowed. By preventing the creation of additional data alignment or movement in nodes and allowing direct access to the LUB node using the rear cursor without going through the middle nodes, the T*-tree structure shows improved processing algorithms. Moreover, for efficient use of storage space, an additional rear cursor has been added to the T-node structure for T*-tree, when compared to the existing T-tree, while also having the same structure inside. Because the ratio of items per node is not much different, it retains the advantage of T-trees for its efficient storage capacity [9].

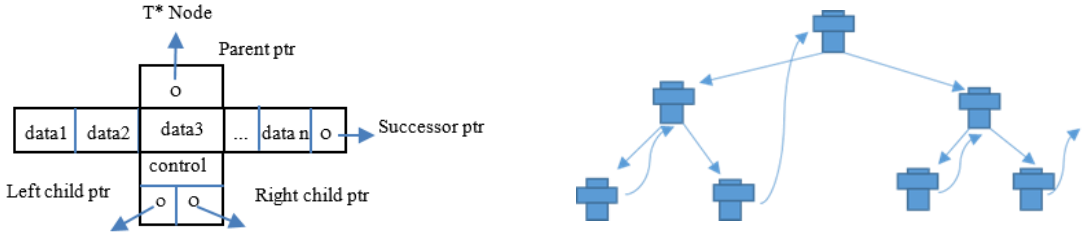

Figure 4. T*-tree

2.5. Power-off failure

In the hardware system concept, power is an important resource. If you lose power supply to hardware then all stop. Therefore the power source must be managed carefully. There are two types of power-off: normal power-off and sudden power-off. We only care about the case of sudden power outages and they are known as a power-off failure [10]. This error has a huge impact and data. In the storage view, flash memory is a hardware component, so it cannot avoid power-off failure. When power failure happens in flash memory, it must suffer unwanted errors. Therefore, managing the loss of power in the flash memory is important. When the mapping table changes in the page mapping FTL, existing data is updated or erased from flash memory. First, if existing data is updated, flash memory must write data to another address because of the feature that it cannot be overwritten. Accordingly, the mapping table must also be modified to reflect the changed address, and if not modified, the updated data cannot be accessed. Next, when the mapping table is modified, the data is deleted from flash memory. If the data is erased by the host or the garbage collection of the FTL, the mapping table must also remove address information for the erased data. When the mapping table is updated, it is necessary to store the changed information in flash memory to recover a power-off failure occurring later. However, storing the mapping information in flash memory every time data is updated causes the performance of flash memory to be degraded, and the storage of excessive mapping information may lead to a capacity shortage. Therefore, in order to solve the above problem, the FTL performs a data backup operation called “flush”. FTL generates flush processes periodically and usually stores mapping information in the flash memory’s data area or spare area.

3. DATA RECOVERY TECHNIQUES ON FLASH MEMORY

Figure 5 shows an overview of the solutions to overcome the data recovery problem on Flash memory. In each group, we had investigated three techniques in which group determine the process, algorithm or technology to solve the problem of data recovery on the power-off failure of Flash memory using B-tree index buffer, index segment log directory, or the approach to surviving the unexpected event by safely backing up the metadata. The first group including BFTL, IBSF, and MR-tree, the second group including BISLD (B+-Tree Index Segment Log Technical Directory), T*-ISLD, and crash recovery technical, and the last group including A-PLR (Accumulation based Power Loss Recovery), HYFLUR (HYbrid FLUsh Recovery), and C-HYFLUR (Compression Scheme for HYbrid FLUsh Recovery). In each group, we compared the techniques in each other in order to make a decision about which technique is the winner.

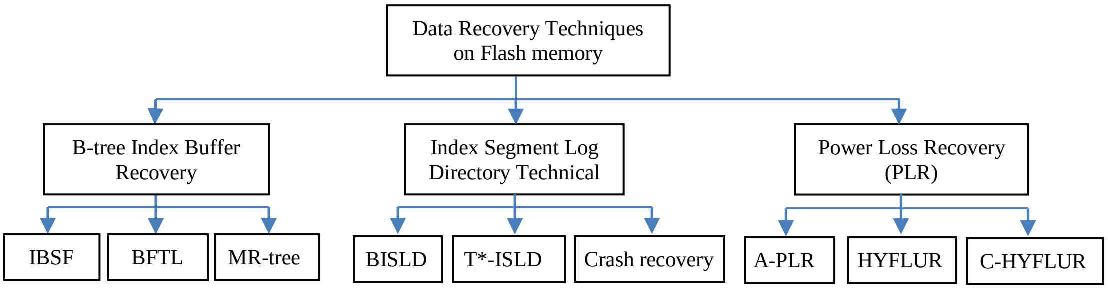
Data Recovery Techniques on Flash memory
B-tree Index Buffer Recovery
Index Segment Log Directory Technical
Power Loss Recovery (PLR)
IBSF
BFTL
MR-tree
BISLD
T*-ISLD
Crash recovery
A-PLR
HYFLUR
C-HYFLUR

Figure 5. Data recovery techniques on flash memory overview

3.1. B-tree index buffer recovery method group

This method group builds a secure B-Tree with a low cost that occurs when setting up a B-Tree in a fast SSD based on low-power flash memory. Moreover, it can reduce the recovery time after an incident. The Flash-based B-tree index uses a failed recovery management method to commit the data that passes through the save points to the SSD. This helps the system restores the data easily when something went wrong. Today B-Tree is mainly used in mass storage devices because of its efficient searching. Therefore, issues that may occur during the B-Tree setup in SSDs can be found in storage systems using the Bulk SSD.

To address these issues, some methods such as BFTL, IBSF, and MR-Tree technique have been proposed. Instead of writing the index units that occur during the creation of the tree to the flash directly, these methods temporarily store the index units in the buffer. When the buffer is full, the index units are collected and flushed to the flash. Through these processes, they reduce the number of write operations, leading to the performance enhancement of B-Tree index on flash memory.

3.1.1. IBSF technique

IBSF uses a software module that can insert a flash layer or convert the file system. Moreover, the order of the flash memory is limited to the more efficient B-Tree construction. ISBF is implemented through units of node information change that occurs due to the accumulation of B-tree and keeps the cache temporarily. Once the buffer is full, the node is built by collecting all the units and committed to one page. In addition, ISBF reduces the number of units by eliminating the redundant units (units that are duplicated) or keeping it as bitstream mode. Through this process, IBSF delays the time that the buffer is full reducing the number of write operations. Therefore, the ISBF can reduce the overhead of committing because it requires only one activity to complete [8]. However, IBSF suffers from data loss in the case of a power failure because some records have not flushed to flash memory yet, and, IBSF also consumes the amount of main memory.

Figure 6 shows the architecture of IBSF that includes index buffer, insertion policy, deletion policy, and commit policy. The index buffer keeps index units which reflect modified B-tree nodes when inserting, deleting, or modifying the records. According to the insertion and deletion policies, IBSF handles index units in the index buffer. If the index buffer is filled, it may generate the commit operation by the commit policy. Since IBSF stores the index for one node of the B-tree in one page, it needs not the translation table which is an additional overhead. This may improve the read performance of B-tree as compared to BFTL because IBSF visits one page to rebuild one node.

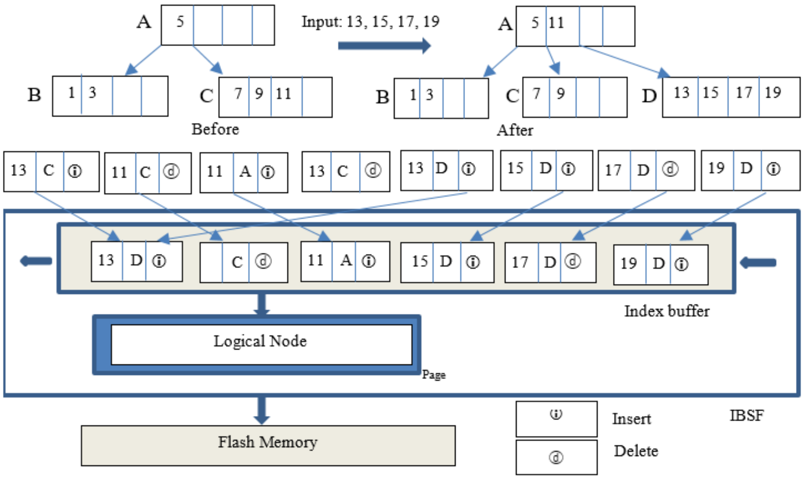

Figure 6. The architecture of IBSF

3.1.2. BFTL technique

BFTL is composed of a reservation buffer and a node translation table. Every inserted or modified record is temporarily stored in the reservation buffer. When the reservation buffer is full, all records in the buffer are flushed to the flash memory in FIFO order by an internal operation of BFTL called commit. BFTL builds an index unit to reflect the primary key insert/delete to the B-tree index. Many index units which belong to different B-tree nodes can be packed into a few sectors to reduce the number of pages physically written. Due to this packing mechanism, index units of one B-tree node can now exist in different sectors. To help BFTL collect index units of the same B-tree node and maintain the information of the pages having the index units of the corresponding B-tree node, a node translation table is used. A node translation table is introduced as an auxiliary data structure to make the collecting of the index units efficient. The node translation table maps a B-tree node to a collection of logical page addresses where the related index units reside. By using the reservation buffer, BFTL reduces the number of flash operations.

Although the number of write operations is reduced, BFTL may generate many read operations to access a B-tree node because the data of one node may be scattered in several pages. The buffer holds all generated index units even though these are redundant index units, so the buffer fills up quickly. Moreover, the node translation table and its list must be maintained in RAM and their sizes may grow rapidly. Therefore, BFTL consumes the amount of main memory and its data may be lost in case of sudden power failure [11-13].

Figure 7 shows an example, the index units are packed on page 20 and 21 when inserting the data which have 1, 7, 9, 17, 21, and 24 as the key value. Since the index units can be stored each other pages for a node, the node translate table is needed. In order to form a correct logical view of a B-tree node, BFTL visit all pages where related index units reside and then construct an up-to-date logical view of the B-tree node as in Figure 6 node E has to reference page 13 and 20.

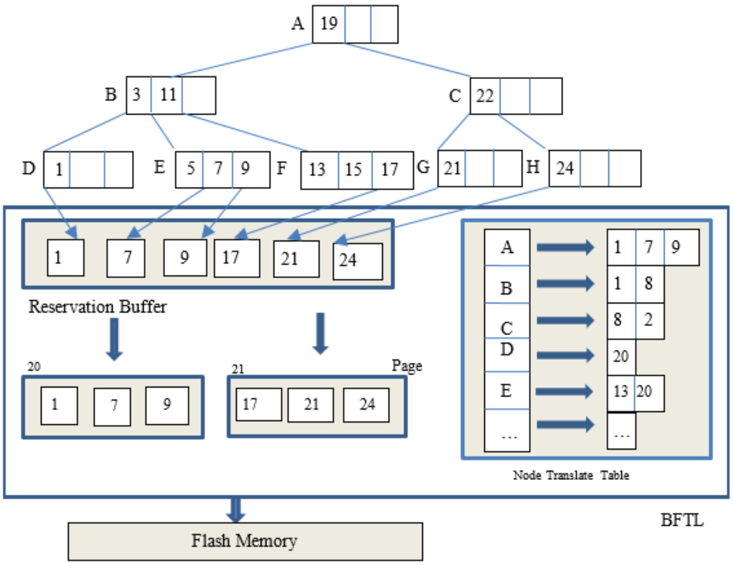

Figure 7. The architecture of BFTL

3.1.3. MR-tree technique

MR-Tree [14] is a disk-based index in the spatial index form that does not require the use of primary memory but can still access spatial data. In addition, the MR-Tree has a spatially structured index of transformed spatial data that can access the R-Tree [15]. In the main memory system, as the difference between the speed of the CPU and the speed of the memory is increasing, it is important to reduce the amount of cash that is missing, and taking into account the characteristics, the MR-Tree has a small advantage over the R-Tree by speeding up the use of the middle button while lowering the height of the tree. Also, through the use of indexing techniques in flash memory, effective access and management of large amounts of data is possible. However, Tree Index has frequent changes in the insert, delete and break actions. In addition, even though node entry changes, write and delete operations occur frequently, causing performance degradation problems.

Therefore, to address this problem, the realization of an effective R-Tree and reducing the frequency of the write operation of the BFTL buffer layer and flash memory system is used to troubleshoot low performance. But there are problems with search activities, and while the effective approach and management of large data are possible, there is the pressure to decode and record similar fields in the same place, when the activities of dynamic insertion/deletion and rebalancing operations such as split/merge are made. Figure 8 shows the structure of MR-Tree.

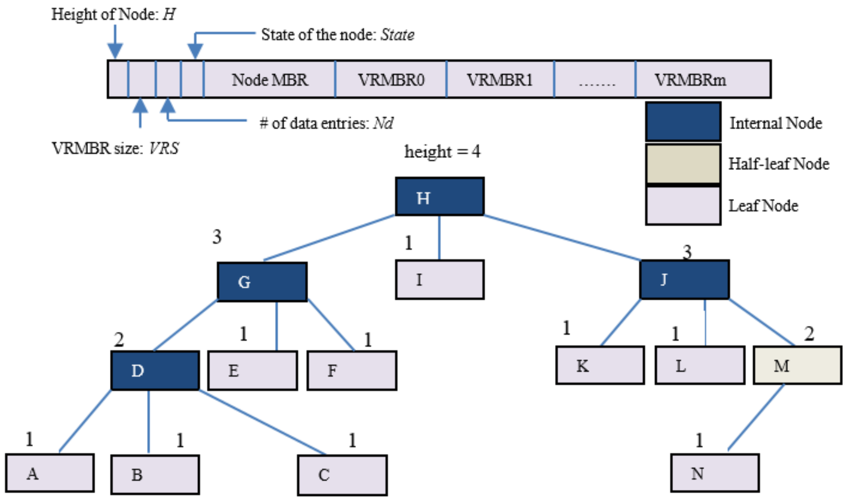

Figure 8. Structure of MR-tree

3.1.4. Comparison

IBSF can effectively reduce the number of write/delete operations. Furthermore, IBSF eliminates the table translation node as an additional cost. Therefore, it can reduce the number of reading activities, and IBSF yields better performance than BFTL. Table 2 shows the ratio of sequence comparison for the read, write, erase, and total cost (time µs) between BFTL and IBSL.

Table 2. Ratio of sequence (RS) comparison [11]

RS	BFTL	IBSF
0	Number of pages	Number of pages
Read	~20000	~5000
Write	~10000	~5000
Erase	~300	~100
Total cost (time µs)	~20000	~10000
0.3		
Read	~30000	~10000
Write	~25000	~10000
Erase	~600	~350
Total cost (time µs)	~40000	~20000
0.5		
Read	~40000	~10000
Write	~35000	~15000
Erase	~900	~400
Total cost (time µs)	~50000	~20000
0.7		
Read	~55000	~15000
Write	~45000	~20000
Erase	~1200	~500
Total cost (time µs)	~65000	~30000
1		
Read	~75000	~20000
Write	~65000	~25000
Erase	~1750	~750
Total cost (time µs)	~80000	~40000

3.2. Index segment log directory technical group

This is a group using the index segment log directory techniques for failure recovery. This group includes BISLD, T*-ISLD, and NAND Flash File Crash Recovery Technique.

3.2.1. BISLD technique

BISLD (B+-Tree Index Segment Log Technical Directory) technique is a failure recovery method using the B+-tree and the directory log buffer [16]. Among the different logging techniques, it applies delay techniques for recovery. Once completed, it performs recovery through reoperation and with updated activities in the log directory. The BISLD algorithm is as follows. Step 1: Get the inserted leaf nodes.

Step 2: Get a node with a value less than the key value of the index node. Step 3: If the inserted key is larger than the key value of the index node, follow the larger value key. Step 4: Take the next button. And the last step: Repeat until the leaf node to insert is found [16]. This technique uses B+-tree and is prone to updating the location frequently in the same area while working. For that reason, costs incur if the recovery process fails. Figure 9 shows the recovery structure of BISLD. The B+-tree algorithm is applied to the insertion and retrieval of a particular client so that it is possible to quickly retrieve the redo log in the event of specific client failure because it is superior to other indexes when searching.

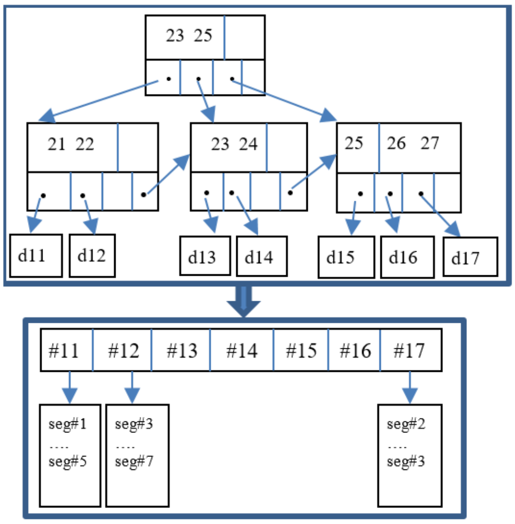

Figure 9. Recovery structure in BISLD

3.2.2. T*-ISLD method recovery system structure

The recovery structure of T*-ISLD includes the T*-tree index; the log directory as shown in Figure 10. The T*-tree algorithm is applied when inserting and retrieving data [9]. Applying algorithms, recovery is performed quickly by quick access to RedoLog, when errors occur with specific flash memory. This is because T*-tree excels in insertion and retrieval, compared to other indexes [9]. The current technique creates an index unit when T*-tree is changed, maintains it in the buffer, and executes the commit when the buffer is full. It also records and stores change the information in the log, commits all index units when the original node is changed and executes the checkpoint.

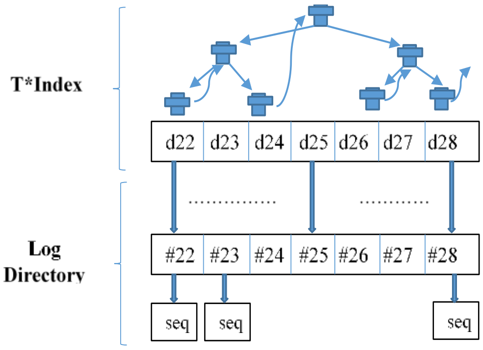

Figure 10. Recovery structure in T*-ISLD

Failure recovery algorithm is as shown in Figure 11. Insertion and retrieval are applied to the T*-tree algorithm. This algorithm retrieves RedoLog in case of a failure in specific flash memory and enables quick recovery. The log record of the failure recovery can be divided into Redo and Undo commands. The log record is managed in the log and has the form of <command, node number, key number> during the changes in the T*-tree. For example, <Insert, A, 20> is a command to insert key value of 20 to the node A. For failure recovery, the stored A, B, and C nodes in the flash memory are used to build the T*-tree, and then the log information is used to perform recovery before the failure.

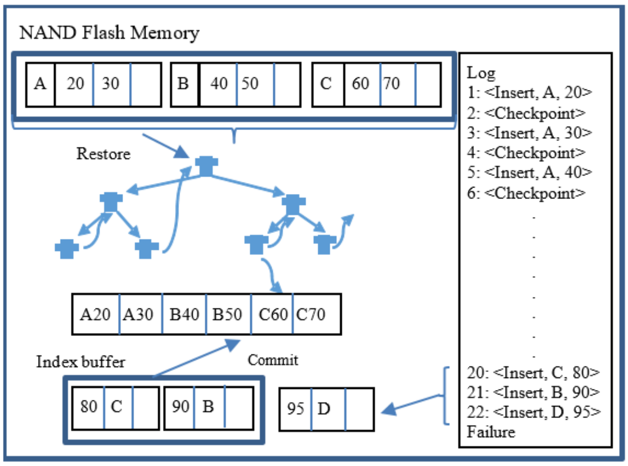

Figure 11. Failure Recovery Algorithm

3.2.3. NAND flash file crash recovery technique

This technique is used to effectively reset in abnormal file system crash situations by changing the settings of the work area so that the burn tool operation is only used in specific areas of the entire flash memory capacity. In addition, the work area data can be saved in the reserved space, so when the initialization is done, the user can capture the copy of the most recent set activity. Through this structure, only the most recent work area is inspected and initialized in accident situations. The amount of time required for a crash to be restored correspondingly increases with the size of the work area. This technique solves the problem of increasing scalability: no matter how large the amount of flash memory is, the recovery time corresponds to it. However, this technique has a weakness in the capacity of flash memory that is not commensurate with the recovery time [17]. Figure 12 shows the crash recovery technique.

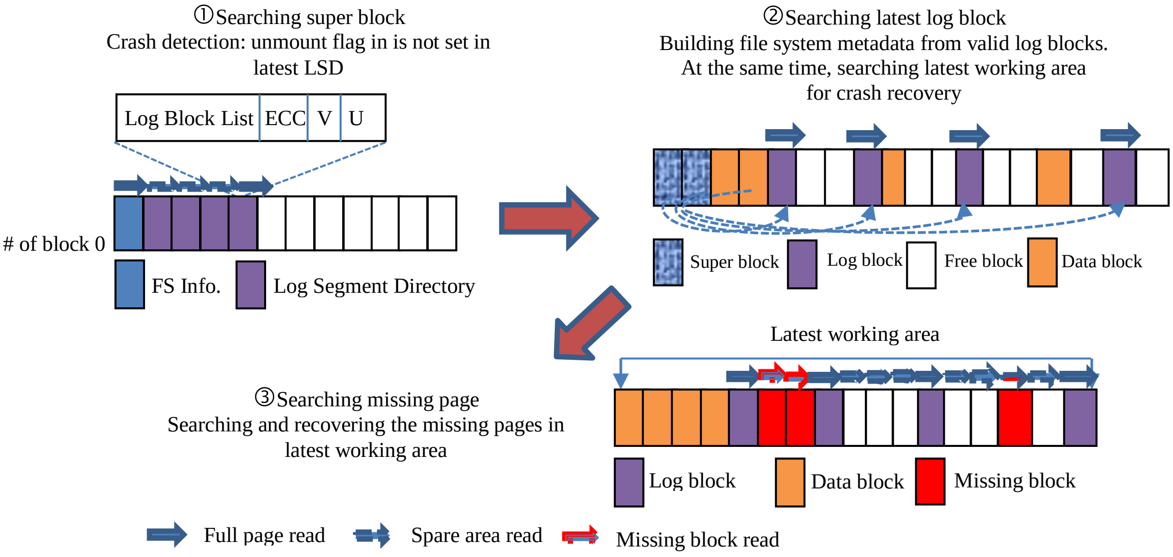
①Searching super block
Crash detection: unmount flag in is not set in latest LSD
Log Block List ECC V U
# of block 0
FS Info.
Log Segment Directory
②Searching latest log block
Building file system metadata from valid log blocks.
At the same time, searching latest working area for crash recovery
Super block
Log block
Free block
Data block
Latest working area
③Searching missing page
Searching and recovering the missing pages in latest working area
Log block
Data block
Missing block
Full page read
Spare area read
Missing block read

Figure 12. Crash recovery technique

3.2.4. Comparison

The system environment for the performance evaluation was Intel CPU Core i7-4790 (3.49GHz), 8GB RAM, OS Windows 7 [9]. The performance of T*-ISLD is better than the BISLD as the number of index tree increased. In the future, to reduce overhead for failure recovery in NAND Flash memory technique can be used in various fields. Table 3 shows the comparison of recovery performance (unit: ms) between BISLD and T*-ISLD.

Table 3. Recovery performance (unit: ms) comparison [9]

Index tree number	BISLD	T*-ISLD	Improvement
15000	13.87	9.40	32%
25000	14.61	6.05	59%
35000	15.10	4.52	70%
40000	15.29	4.02	74%
45000	15.46	3.62	77%
50000	15.61	3.30	79%
55000	15.75	3.04	81%
60000	15.87	2.82	82%

3.3. Power loss recovery (PLR) group

Power Loss Recovery (PLR) approach makes the storage to survive the unexpected event by safely backing up the metadata [24-25]. There are three approaches: the first, In-Block Backup, reserves a certain number of blocks in flash memory as a metadata area and stores the modified metadata into the blocks whenever any of them is altered. When a sudden power failure event occurs, the storage controller identifies the location of the metadata area and uses the information stored in the area during the boot-up process to restore the up-to-date system state. In the second approach, In-Page Backup, the metadata associated with each data page is also stored into its spare area. During a recovery process, the controller reconstructs the up-to-date system state by reading the spare area of each page. And the last, Hybrid Backup uses a mixture of the above-mentioned two techniques: it periodically stores a backup of the mapping table in the metadata area and it also copies page mapping information into the spare area of the corresponding data page. This group includes Accumulation based Power Loss Recovery (A-PLR), HYbrid FLUsh Recovery (HYFLUR), and Compression Scheme for HYbrid FLUsh Recovery (C-HYFLUR).

3.3.1. A-PLR

In-Block Backup scheme stores backup data in a set of dedicated blocks in flash memory called map blocks. First, map blocks to store mapping information in flash memory is prepared to be used when mapping the generated information. Since this scheme also creates additional write operations, it will incur an additional cost. When a power failure occurs in flash memory, the recovery algorithm scans the map blocks. And based on the scanned information, it creates mapping tables. Compared to the In-Page Backup scheme, the recovery latency of the In-Block Backup scheme is small [18].

In-Page Backup scheme, the backup area in each data page is usually used to store ECC (Error Correction Code) parity bits and other FTL management information. As a general rule, the size of the backup area is 1/32 of the page size and is not visible to the user. Redundant areas can be programmed (or read) in a written (or read) operation without operating costs. The backup plan in the site stores the mapping information in redundant areas in a uniformly distributed manner [18]. Since this scheme uses redundant areas, there is no cost for running backups. However, during recovery, the storage controller must read the entire page of the storage device, so the recovery lag may be too long.

Hybrid Backup uses a mixture of the two techniques mentioned above and effectively reduces both recovery time and run time backup costs. In this scheme, page-level map table updates (for example, logical page number: LPN) for each page record are maintained in the standby area of the corresponding pages and cumulative page mapping tables and updated block-level mapping table (for example, logical block number: LBN) is continuously stored in map blocks. During the recovery process, page-level and block-level mapping tables are restored from the map blocks and the backup area of the pages. This technique helps to improve the recovery range in the In-Block Backup and to reduce the recovery time compared with In-Page Backup. The run-time backup overhead of this scheme is as small as that of In-Block Backup because the read/write latency of the spare area is negligible [18]. As mentioned above, the main disadvantage of In-Page Backup is the recovery delay. To overcome this disadvantage, Jung *et al.* [18] proposed an advanced In-Page Backup scheme called A-PLR. This scheme stores mapping information in the spare area as In-Page Backup. However, unlike In-Page Backups, the mapping information is stored in a different way.

The simple process of this method is as follows. First, A-PLR uses a special buffer in its own RAM called MIB (mapping information buffer), whose size is fixed, and the accumulation of mapping information. When A-PLR stores mapping information in the spare area, it stores in the MIB. As A-PLR stores mapping information, the size of the MIB increases. When MIB becomes full, the information is also stored in the spare area and this page is called an intermediate page.

This page is used during the recovery process. The recovery process of A-PLR is very simple. When the recovery process runs due to a power failure, A-PLR uses an intermediate page. For example, A-PLR browses all pages. Therefore, its recovery delay is lower than that of In-Page Backup because A-PLR uses a re-mapping table by browsing intermediate pages. Figure 13(a) shows the backup process and Figure 13(b) shows the recovery process of A-PLR.

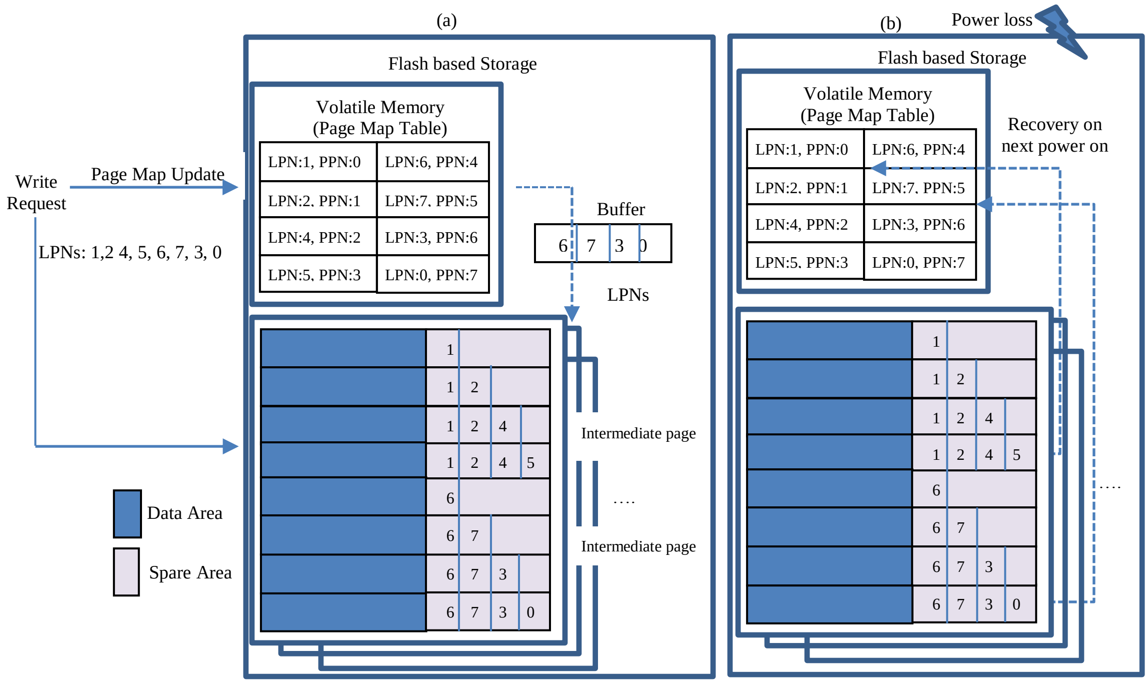

Figure 13. A-PLR (a) backup and (b) recovery process

3.3.2. HYFLUR

HYFLUR [19] focuses on the hybrid backup scheme and FTL page mapping scheme. This scheme is divided into three flush steps to transfer information from RAM to flash memory: SWF (spare write flush), URF (update RAM flush), and MTF (mapping table flush). This scheme also stores metadata in a certain block in the flash memory as In-Block Backups.

Flash memory is a hardware; it cannot avoid power-off failure and must suffer unwanted errors like other mediums. Therefore, recovery from the loss of power in the flash memory is important. If a power failure occurs suddenly during the execution of FTL operations, the probability of data loss is very high. Therefore, it is necessary to save the data for recovery. To reduce the risk of losing data, the system saves data periodically. In general, the data of the flash memory changes during the write or delete operation. So, to save the changed data, FTL only cares when the write and delete operations are executed.

A backup operation is called flush. As mentioned above, the flush is performed periodically in FTL. During the flush process, saving the location is as important as saving the location associated with the recovery delay. Generally, flushed data is stored in the spare area or data area on the page in flash memory. To reduce the recovery delay, HYFLUR uses three flush schemes, that is, MTF, URF, and SWF.

Figure 14 shows three flush operations implemented with this timeline. MTF, URF, and SWF can be implemented with reasonable policies. SWF operation is very simple. For example, if the page is updated, the data will be saved on another page. This case suggests changing mapping information such as ppn (physical page number). In this scheme, the old and new ppn changes are written in the spare area of the RAM and the process is called SWF.

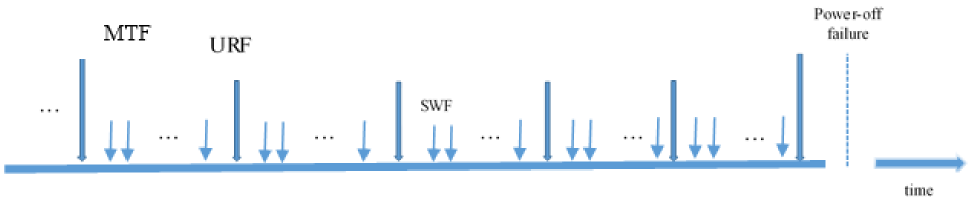

Figure 14. Description of a flush operation

URF operation is the process of writing old and new ppn to certain pages of blocks called MSB. As mentioned above, when a page is updated, ppn information is written to RAM. This scheme assigns 6 bytes for each ppn so it can write 1365 ppns in 8KB memory. That means it can store mapping information of 1365 pages and Figure 15 briefly describes the URF operation.

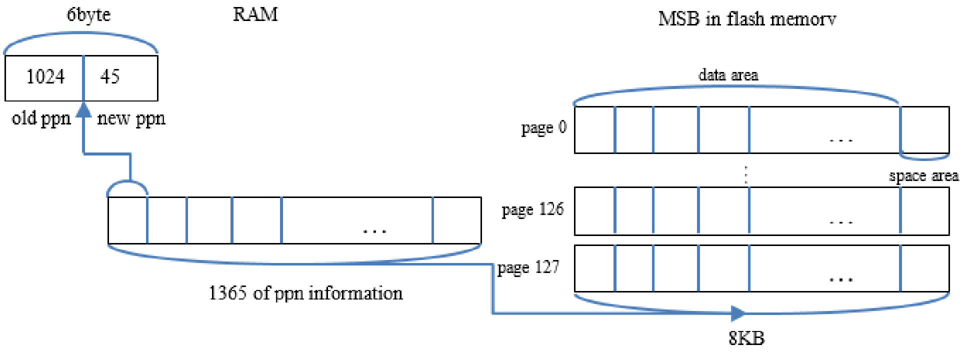

Figure 15. A brief description of URF operation

If 8KB memory is full, the URF operation implemented also means approximately 1.3GB of flash memory is updated. When URF operation is executed and all information is written to the pages of the MSB it can store mapping information about 30.8% of flash memory. MTF operation is the process of recording all mapping information in flash memory by saving the mapping table, which uses 3 bytes to write ppn. Thus, it can record 2730 ppns in 8KB pages. This operation occurs when the MSB is full because of the URF operation. Therefore, when the MTF executes, the new mapping is output using the mapping table and the MSB. Also, it saves the new table map information as shown in Figure 16.

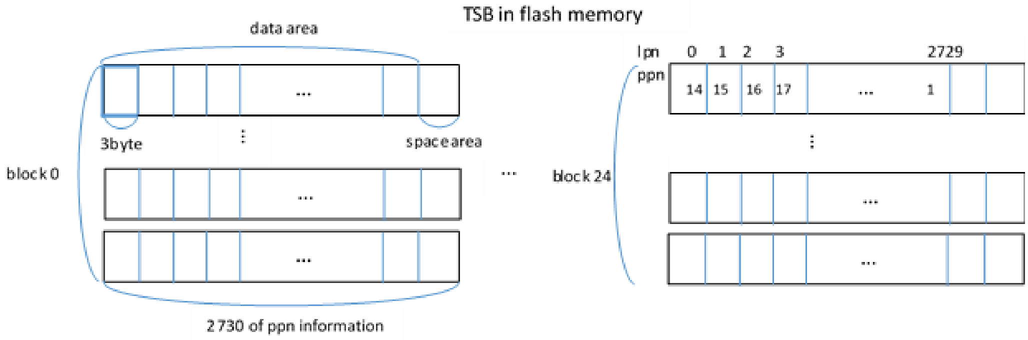

Figure 16. A brief description of MTF operation

When MTF operation is executed, ppn is written to pages and these index pages become lpn (logical page number) so this scheme can build a mapping table by reading TSB. Power-off failure recovery process is divided into three cases. To recover mapping information, it need read the MSB pages generated by the URF operation, the TSB generated by the MTF operation, or the MSB and TSB.

a. In the first case, when a power failure occurs, TSB is not created. So, it reads the mapping table and pages of the MSB to recovering the mapping table. In addition, it reads the spare area of the pages for mapping information because all the mapping information is generated by the SWF operation.
b. In the second case, when a power outage occurs, the TSB is created. In this case, this scheme reads the TSB to create a mapping table because all the mapping information is stored in the TSB. So it does not need to read MSB and this scheme works best.
c. In the last case, the power outage occurs before the new TSB is created. In this case, the recovery scheme reads the existing TSB and the pages of the MSB to find the mapping information generated by the SWF operation.

3.3.3. C-HYFLUR

C-HYFLUR [10] is the modified HYFLUR recovery with the compression algorithm applied to it. The compression algorithm Lempl-Ziv-Storer-Szymanski (LZSS), with improvements from LZ77 [20] is applied to the HYFLUR algorithm to address the spatial cost of a project page mapping vulnerability. Figure 17 shows the RAM configuration when the first RAM is full, the compression process is performed and compressed data is stored in 2nd RAM. This process repeated three times to make 2nd RAM full.

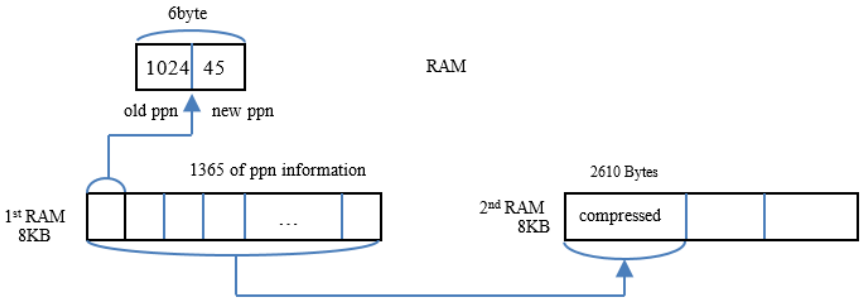

Figure 17. RAM configuration

Figure 18 shows a brief URF (update RAM flush) operation. If the 8KB of 2nd RAM is full, URF operation is performed. This means 32MB of flash memory has been updated. And URF operation is performed and all information is written to the pages of MSB. And MSB can store compressed mapping information of flash memory.

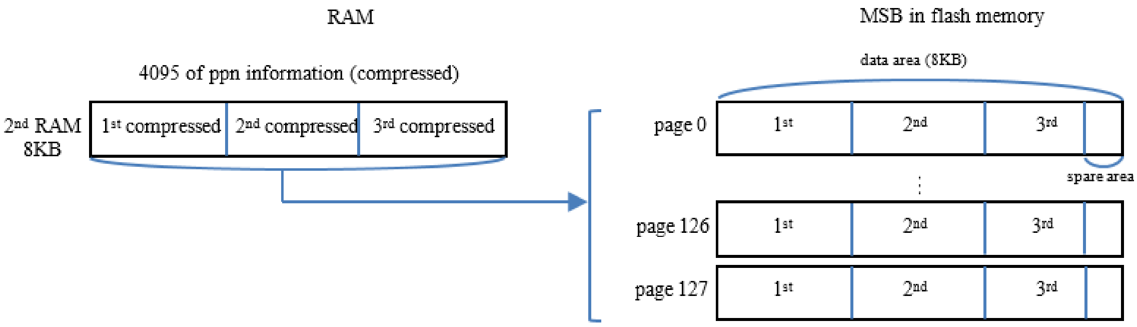

Figure 18. URF operation

MTF operation is the process of writing all mapping information in flash memory. This scheme using 3byte for writing ppn, compress the three 8KB pages with 2730 ppn and write to one page of TSB. This operation occurs when MSB is full because of the URF operation. Figure 19 shows new compressed mapping table information and write the ppn in pages of TSB. For example, when MTF operation is executed, new ppn is written to pages and these pages index becomes lpn. It compressed three 8KB pages and write them to one page of TSB so that total 8190 ppn can be stored in on one page.

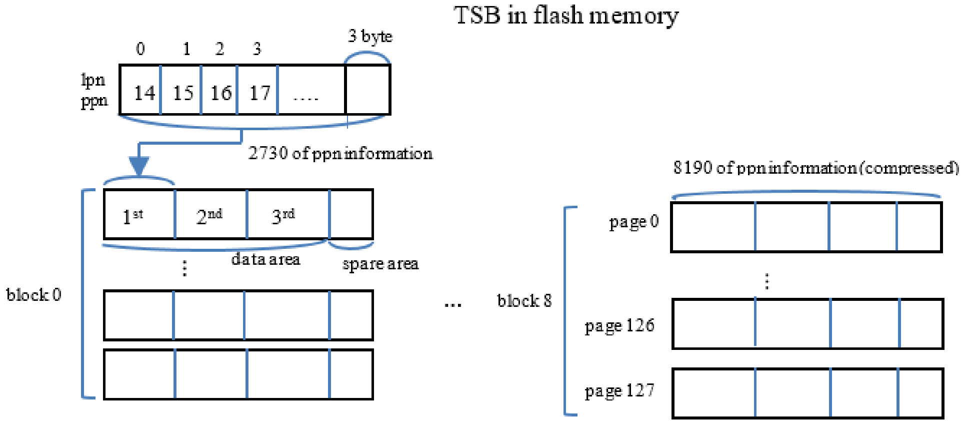

Figure 19. The structure of TSB

C-HYFLUR is implemented on open SSD project board [21] and special hardware as follows. The platform uses 16GB NAND flash and compressed memory of 8KB pages. Blocks of memory are composed of 128 pages. In short, NAND memory consists of a total of 66,432 blocks, also platform uses 64MB of mobile SDRAM. Also, this scheme reserves a specific block of flash memory as a certain memory of RAM to store the mapping information. And the block does not save the data. Because of saving mapping and table data, C-HYFLUR stores 15 blocks and 25 blocks, respectively. In addition, each MSB (mapping information store block) and TSB (table store block) shows HYFLUR storage rates, conventional recovery algorithms, and C-HYFLUR, using a compression algorithm. The compression ratio of LZSS is about 31% and 8KB data can save about 2.5 KB. The mapping information stored in the HYFLUR, MSB can update about 31% of the flash memory, but the C-HYFLUR update data is about three times as large as HYFLUR's. In addition, TSB requires 25 blocks to store all mapping tables, but C-HYFLUR can store all mapping tables with only nine blocks. This can address the disadvantageous cost of FTL page mapping [10]. Power-off failure recovery process is divided into three cases. To recover mapping information it need to read the MSB pages generated by the URF operation and decompression processes, the TSB generated by the MTF operation and decompression processes, or the MSB and TSB and decompression processes.

a. In the first case, when a power failure occurs, TSB is not created. So, it reads the mapping table and pages of the MSB to recover the mapping table. In addition, the pages storing the mapping information are compressed, the decompression process should be performed. And it reads the spare area of the pages for mapping information because all the mapping information is generated by the SWF operation.
b. In the second case, when a power outage occurs, the TSB is created. In this case, this scheme reads the TSB to create a mapping table because all the mapping information is stored in the TSB. So it does not need to read MSB. This process requires decompression the compressed pages to complete, and this scheme works best.
c. In the last case, the power outage occur before the new TSB is created. In this case, the recovery scheme reads the existing TSB and the pages of the MSB. It involves a decompression process. And read the spare area of pages to find the mapping information generated by the SWF operation.

3.3.4. Comparison

A-PLR reads all intermediate pages in the recovery process. Thus, the number of pages to read are greater than HYFLUR. Compression algorithm reduces the number of a read operation and resolves the space overhead of the page mapping FTL. Therefore, C-HYFLUR recovery time overhead is very lower than A-PLR and HYFLUR. However, since additional page write operations and compression process are required, the response time is a little longer than A-PLR and HYFLUR. Table 4 shows the comparison of average response time between A-PLR, HYFLUR, and C-HYFLUR.

Table 4. Average response time (ms) [10]

	A-PLR	HYFLUR	C-HYFLUR
Write, sequential (100%)[a]	0.6	0.7	0.8
Write, random (100%)[b]	1.1	1.1	1.2
Write, sequential (50%)[c]	0.9	0.9	1
Write (50%), sequential (50%)[d]	0.9	1	1

[a]Only write operations are issued and all requests are sequential (100%).
[b]Only write operations are issued and all requests are random (100%).
[c]Only write operations are issued and requests are sequential and random (50%).
[d]Only write (50%) and read (50%) operation and request is sequential (50%) and random (50%).

4. CONCLUSION

Flash memory has more and more applications in life and capacity enhancement solutions are widely applied in the support devices. It is extremely important to recover data when something goes wrong with Flash memory in cases a sudden power outage or power failure. Therefore, recovering data is a very active research field and attracts a lot of researchers. So far, there are many technical solutions have been proposed. This article provides an overview of some recovering methods that are presented in articles, patents, or special journals. In the future, data recovery algorithms will be embedded in flash-based applications and theories of improving recovery performance and processing time will be explored.

ACKNOWLEDGMENT

This research was supported by the MSIP (Ministry of Science, ICT & Future Planning), Korea, under the National Program for Excellence in SW) (2018-0-00209-001) supervised by the IITP (Institute for Information & communications Technology Promotion).

REFERENCES

[1] T. S. Chung, *et al*., "A Survey of Flash Translation Layer," *Journal of Systems Architecture*, vol. 55, pp. 332-343, 2009.

[2] Y. Li and K. N. Quader, "NAND Flash Memory: Challenges and Opportunities," *Computer*, vol. 46, pp. 23-29, 2013.

[3] T. S. Chung, *et al*., "PORCE: An efficient power off recovery scheme for flash memory," *Journal of Systems Architecture*, vol. 54, pp. 935-943, 2008.

[4] A. Ban, "Flash File System," The United States Patent No. 5,404,485, 1995.

[5] S. H. Lim and K. H. Park, "An efficient NAND flash file system for flash memory storage," *IEEE Trans. Comput*., vol. 55, pp. 906-912, 2006.

[6] H. W. Tseng, *et al*., "Understanding the impact of power loss on flash memory," *Proceedings of the 48th IEEE/ACM International Conference on Design Automation Conference (DAC'11)*, vol. 10, pp. 35-40, 2011.

[7] V. P. Ho and D. J. Park, "A Survey of the-State-of-the-Art B-tree Index on Flash Memory," *International Journal of Software Engineering and Its Applications*, vol. 10, pp. 173-188, 2016.

[8] D. S. Batory, "B+ Trees and Indexed Sequential Files: A Performance Comparison," *Proceeding ACM SIGMOD International Conference on Management of Data*, pp. 30-39, 1981.

[9] K. R. Choi, *et al*., "T*-tree: An Efficient Indexing Technique for Main Memory Database," *Journal of the Korean Institute of Communications and Information Sciences*, vol. 21, pp. 2597-2604, 1996.

[10] J. H. Chung, *et al*., "C-HYFLUR: Recovery for Power-off Failure in Flash Memory Storage Systems Using Compression Scheme for HYbrid FLUsh Recovery," *International Conference on Mobile and Wireless Technology (ICMWT 2017): Mobile and Wireless Technologies*, pp. 284-294, 2017.

[11] H. S. Lee, *et al*., "An Efficient Recovery Management Scheme for an Index Buffer of B+tree based on NAND Flash memory," *Journal of Database Research Society*, vol. 27, pp. 20-40, 2011.

[12] C. H. Wu, *et al*., "An Efficient B-Tree Layer for Flash-Memory Storage Systems," *Proc. RTCSA, Tainan, Taiwan*, pp. 409-430, 2003.

[13] H. S. Lee and D. H. Lee, "An Efficient Index Buffer management Scheme for Implementing a B-Tree on NAND Flash Memory," *Data & Knowledge Engineering*, vol. 69, pp. 901-916, 2010.

[14] K. C. Kim and S. W. Yun, "MR-Tree: A cache-conscious main memory spatial index structure for mobile GIS, Web, and wireless geographic information systems," *The 4th international workshop (W2GIS 2004)*, pp. 167-180, 2004.

[15] C. H. Wu, *et al*., "An Efficient R-Tree Implementation over Flash-Memory Storage Systems," *Proc. Lf ACM CIS'03. New Orleans, Louisiana, USA*, pp. 17-24, 2003.

[16] S. J. Cho and S. S. Han, "Recovery Model Improvement Using BISLD in Mobile Computing Environment," *The Journal of Korea Academia-Industrial Cooperation Society*, vol. 13, pp. 4786-4793, 2012.

[17] H. C. Park and C. Yoo, "A Design of Efficient Crash Recovery Technique for NAND Flash File System," *Journal of Korea Information Science*, vol. 35, pp. 470-475, 2008.

[18] Jung S. and Song Y. H., "Data loss recovery for power failure in flash memory storage systems," *Journal of Systems Architecture,* vol. 61, pp. 12-27, 2015.

[19] Chung J. H. and Chung T. S., "HYFLUR: recovery for power-off failure in flash memory storage systems using HYbrid FLUsh recovery," *Information science and applications (ICISA 2016)*, vol. 376, pp. 501-510, 2016.

[20] Ziv J. and Lempel A., "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, pp. 337-343, 1977.

[21] Indilinx Jasmine Platform Specification. http://www.openssd-project.org/wiki/The_OpenSSD_Project/

[22] Ma D., Feng J. and Li G., "A Survey of Address Translation Technologies for Flash Memories," *ACM Computing Surveys (CSUR)*, vol. *46*, issue 3, article no. 36, 2014**.**

[23] Lee H.-S., Yun H.-S. and Lee D.-H., "HFTL: Hybrid Flash Translation Layer based on Hot Data Identification for Flash Memory". *IEEE Transactions on Consumer Electronics*, vol. *55*, pp. 2005-2011, 2009.

[24] Zhang C., Wang Y., Wang T., Chen R., Liu D. and Shao Z., "Deterministic crash recovery for NAND flash based storage systems," *In: Proceedings of the 51th design automation conference (DAC 2014)*, 2014.

[25] Zheng M., Tucek J. and Lillibridge M., "Understanding the robustness of SSDs under power fault," *In: Proceedings of 11th USENIX conference on file and storage technologies (FAST 2013)*, 2013.

BIOGRAPHIES OF AUTHORS

Van Dai Tran, he received his BS degree in the Informatics Department at Da Nang University in July 2002 and an MS degree in the Computer Science Department at Vietnam National University-Ho Chi Minh City in June 2008. He is currently a Ph.D. candidate in the Department of Computer Science and Engineering at Soongsil University. His research interests include flash memory-based DBMSs and database systems.

Dong-Joo Park, he received his BS and MS degrees in the Computer Engineering Department at Seoul National University in February 1995 and February 1997, respectively, a Ph.D. in School of CS&E from Seoul National University in August 2001. He is currently an associate professor in the Department of Computer Science and Engineering at Soongsil University. His research interests include flash memory-based DBMSs, multimedia databases, and database systems.