Crash Recovery in FAST FTL*

Sungup Moon, Sang-Phil Lim[1], Dong-Joo Park[2], and Sang-Won Lee[1]

[1] Sungkyunkwan University, Cheoncheon-Dong, Jangan-Gu, Suwon, Gyeonggi-Do 440-746, Korea
[2] Soongsil University, Sangdo-Dong, Dongjak-Gu, Seoul 156-743, Korea
{sungup,lsfeel0204}@skku.edu, djpark@ssu.ac.kr and swlee@skku.edu

Abstract. NAND flash memory is one of the non-volatile memories and has been replacing hard disk in various storage markets from mobile devices, PC/Laptop computers, even to enterprise servers. However, flash memory does not allow in-place-update, and thus a block should be erased before overwriting the existing data in it. In order to overcome the performance problem from this intrinsic deficiency, flash storage devices are equipped with the software module, called FTL (Flash Translation Layer). Meanwhile, flash storage devices are subject to failure and thus should be able to recover metadata (including address mapping information) as well as data from the crash. In general, the FTL layer is responsible for the crash recovery. In this paper, we propose a novel crash recovery scheme for FAST, a hybrid address mapping FTL. It writes periodically newly generated address mapping information in a log structured way, but it exploits the characteristics of FAST FTL that the log blocks in a log area are used in a round-robin way, thus providing two advantages over the existing FTL recovery schemes. One is the low overhead in performing logging during normal operations in FTL. The other is the fast recovery time.

1 Introduction

For the past decade, we have witnessed that flash memory is deployed as an alternative data storage for mobile devices, laptops, and even enterprise server applications. Mainly because of the erase-before-update characteristics in flash memory, every flash memory storage comes with a core software module, flash translation layer(FTL). Since FTL can critically determine the performance of a flash device, numerous FTLs have been proposed. And taking into account that the capacity of flash memory devices drastically increases, the efficiency of FTLs becomes more and more important. According to address mapping, the existing FTLs can be categorized largely into block mapping, page mapping, and hybrid mapping.

However, most existing works on FTL has been focused on issues such as performance, wear leveling, and garbage collections. In contrast, despite of its

* This research was supported by MKE, Korea under ITRC NIPA-2010-(C1090-1021-0008) and Seoul Metropolitan Government under 'Seoul R&BD Program (PA090903).'

practical importance, the recovery from power-off failure in FTLs has not been paid much attention to. As far as we know, there is no work which deals with the crash recovery in FTLs comprehensively. When a page write is requested, a new address mapping information, in addition to the data page itself, should also be persistently propagated to flash for power-off recovery, either in page mapping or hybrid mapping. Thus, under a naive solution like this, a page write request in flash memory would at least need two physical flash writes, which could degrade the FTL performance. Of course, both the algorithmic complexity and its run-time overhead in FTL may considerably depend on the address mapping being used. For example, in case of block mapping, we have to write the changed address mapping information, only when the block mapping changes, not for each page write. Thus, the recovery solution in block mapping would be very simple and does not incur much run time overhead. In contrast, for every data page write in page mapping FTL, a new page mapping entry should be written in flash memory. In order to alleviate this overhead, we can use a checkpoint approach for flushing new page-level mapping entries periodically in a log structured manner. But, even this approach has the garbage collection overhead for the old page-level mapping entries.

In this paper, we propose a novel crash recovery mechanism for FAST, a hybrid address mapping FTL. In particular, it writes the new mapping information periodically in a log structured way, but it exploits the characteristics of FAST FTL that the log blocks in a log area are used in a round-robin way, thus providing two advantages over other FTL recovery schemes. One is the low overhead in writing the new mapping metadata during the normal operation in FTL. In fact, our recovery scheme will require a nominal size of mapping metadata to be written per data page write. In terms of additional write overhead for recovery, our scheme could outperform the existing approach by up to an order of magnitude. The other advantage is the fast recovery time. With our scheme, the recovery phase can finish as early as possible, and then the FAST FTL proceed in a normal mode. In addition, this paper deals with the duality of flash write overhead between original data page and the metadata for recovery, and we argue that, for comparing FTLs in terms of performance, we should take into account the overhead in maintaining the metadata persistently for recovery.

This paper is organized as follows. Section 2 describes the related work of flash memory and the address mapping table management, and then we explain our recovery method in section 3. In Section 4, we evaluate the management cost arithmetically. Finally, Section 5 concludes this paper.

2 Background and Related Work

2.1 Background: Flash Memory, FTL, and Address Mapping

With flash memory, no data page can be updated in place without erasing a block of flash memory containing the page. This characteristic of flash memory is called "erase-before-write" limitation. In order to alleviate the erase-before-write problem in flash memory, most flash memory storage devices are equipped

with a software or firmware layer called Flash Translation Layer (FTL). An FTL makes a flash memory storage device look like a hard disk drive to the upper layers. One key role of an FTL is to redirect each logical page write from the host to a clean physical page, and to remap the logical page address from the old physical page to the new physical page. In addition to this address mapping, an FTL is also responsible for data consistency and uniform wear-leveling.

By the granularity of address mapping, FTLs can be largely classified into three types: page mapping FTLs [2], block mapping FTLs [1], and hybrid mapping FTLs including BAST[4] and FAST [5]. In a block mapping FTL, address mapping is done coarsely at the level of block. When the data in a block is to be overwritten, FTL assigns a new empty block for the block. Although this inflexible mapping scheme often incurs significant overhead for page writes, we need to change the block mapping information and store the new mapping information persistently only when a new physical block is assigned to a logical block.

In a page mapping FTL, on the other hand, address mapping is finer-grained at the level of pages, which are much smaller than blocks. The process of address mapping in a page mapping FTL is more flexible, because a logical page can be mapped to any free physical page. When an update is requested for a logical page, the FTL stores the data in a clean page, and updates the mapping information for the page. When it runs out of clean pages, the FTL initiates a block reclamation in order to create a clean block. The block reclamation usually involves erasing an old block and relocating valid pages. However, a page mapping FTL requires a much larger memory space to store a mapping table. Because it has a large address mapping table. In order to guarantee the consistent mapping information against the power-off failure, a page mapping FTL basically stores the whole page mapping table for every page write and this overhead is not trivial if we consider the large page mapping table. In a page mapping table, another way to recover the consistent page mapping information from failure is to store the logical page address in the spare area of each physical page, but this approach will suffer from reconstructing the mapping information by scanning all the pages.

Hybrid mapping FTLs have been proposed to overcome the limitations of both page and block mapping FTLs, namely, the large memory requirement of page mapping and the inflexibility of block mapping. In FAST FTL [4], one of the popular hybrid mapping FTLs, flash memory blocks are logically partitioned into a data area and a log area. Blocks in a data area are managed by block-level mapping, while blocks in a log area are done by page-level mapping. If there is a data write request, the FAST scheme will write the data on clean pages at the end of the log area which are managed by the page level address table. Thus, this hybrid FTL scheme retains higher space utilization than the block level mapping scheme and smaller table size than the page level mapping scheme. But, as in the page mapping FTL scheme, every page write will change the page mapping for the log area so that FAST also has the overhead to store, in addition to the data page itself, the new page address mapping entry persistently for each write. However, as will be described later, there is a unique opportunity in FAST FTL for maintaining the page mapping information consistently with lower overhead.

2.2 Crash Recovery in FTLs

Many recovery mechanisms have been proposed to maintain the mapping information, but those mechanisms should work with the upper software layer like the file system. In the decoupled FTL, filesystem layer cannot access the FTL internal data structure like the page mapping table. On the other hand, except for some paper like LTFTL[6] and PORCE[3], the FTL-specific recovery mechanism has not been researched for the SSD's decoupled FTL.

LTFTL [6] is a page mapping FTL with internal recovery scheme. LTFTL keeps the changed address log at the RAM and checkpoints the logs by a unit page. In this scheme, LTFTL can restore not only the latest address table but also other previous address tables through few page accesses. But LTFTL has to merge the logs with the large address table and write the newly whole address table at the mapping block, if the logs become over the log threshold. This transaction affects the normal I/O request performance because of their large-size address table.

PORCE [3] only focused on the recovery scheme after the power-failure. PORCE divided the power-failure problem into two situations; the normal write operation consistency and the reclaiming operation consistency. Especially in the reclaiming operation, PORCE writes the reclaiming-start-log before the reclaiming operation and the reclaiming-commit-log after the all reclaiming operation at the transaction log area. But FAST FTL always performs their merge operation with many associated data block, and each associated block state will be changed every merging step. Thus reclaiming operation of FAST FTL should be traceable and robust against the repetitive recovery-failure, but PORCE did not mentioned about this.

3 Crash Recovery in FAST FTL

3.1 Overview of FAST FTL

FAST FTL is one of the hybrid FTL schemes which was originally designed to improve the performance of small random write. In FAST FTL, flash memory blocks are logically partitioned to a data area and a log area. Blocks in a data area are managed by block-level mapping, while blocks in a log area are done by page-level mapping. Since no single log block is tied up with any particular data block due to its full associativity, any page update can be done to any clean page in any one of the log blocks. This improves the block utilization significantly and avoids the log block trashing problem as well.

By the way, every write in a log area will change the page-level mapping entry for the logical page being written, and thus each write in a log area requires at least *one additional flash write* for storing the page-level mapping information persistently, thus guaranteeing the consistency of the page-level mapping information against the power-off failure. This additional flash write for storing mapping information will degrade the performance by half. Thus, a more clever

solution for crash recovery in FAST is necessary, and we propose a checkpoint-based recovery scheme for FAST FTL.

One concern which makes the recovery in FAST FTL more complex is the full merge operation in FTL. When the log area is full, a log block is chosen as a victim for reclamation and the victim log block may contain valid pages belonging to many different data blocks. In FAST, log blocks are managed in a FIFO queue, and thus the oldest log block is chosen as a victim block. Those valid pages should then be merged with their corresponding data blocks. This type of merge operation is called a full merge, and is usually very expensive as it involves performing merge operations as many as valid pages in the victim log block. Please see [5] for details. Besides its performance overhead, we should note that one victim log block in FAST may incur many merges, and a crash can occur at any point while handling them. For each block being merged, we need to change its block mapping information and save it in flash memory, too. That is, the reclamation of a victim block can change the block mapping information of several blocks being merged, and the crash between the multiple merges makes it hard to achieve the atomic propagation of the block-level mapping changes from multiple full merges. We will revisit this issue later.

Before closing this subsection, let us explain two assumptions we make in this paper. First, both the free block list (i.e. the list of free flash blocks) and the associated-data-block list (i.e. the list of data blocks to be merged when a victim log block is reclaimed) are maintained as sorted lists. Second, the victim log block should be returned to the free block list, instead of being reused as the new log block immediately after reclamation.

3.2 Revised FAST FTL Architecture

In order to support crash recovery functionality to FAST FTL, we need to revise the previous FAST architecture a little, which is described in this subsection.

Address-mapping tables. FAST FTL maintains two address-mapping tables, a block mapping table for data blocks and a page mapping table for log blocks. For merge correctness, two new columns, MT(Merge Timestamp) and VT(Victim Timestamp) are added to the two mapping tables, respectively, which will be handled in details in Section 3.3.

Logging areas on flash memory. For the fast recovery from system failures, address mapping information in main memory has to be persistently stored somewhere in flash memory. This is why two logging areas in the front of flash memory exist in Fig. 1. The block mapping logging area stores two classes of 'block mapping' information for the data and log blocks. First, whenever the block mapping table for data blocks is updated due to page writes or merge operations, the whole table is stored in the block mapping logging area. This logging cost is not so expensive since such a update does not happen frequently. Second, the recovery module in FAST FTL is required to know what physical

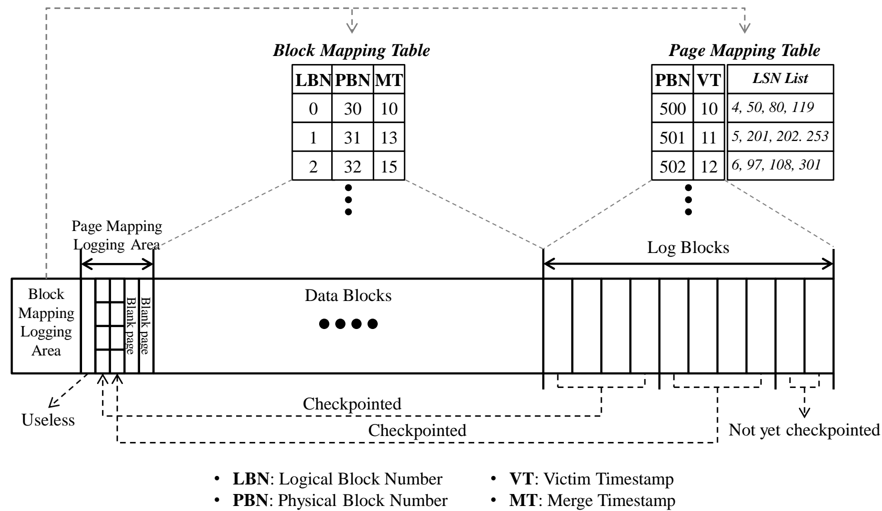

Fig. 1. Revised FAST FTL Architecture

block each of the log blocks are mapped to before the crash, so this information should be stored onto flash memory whenever address mapping between log blocks and physical blocks changes. Of course, the frequency of these changes is not so high, and thus the cost is low.

Next, let us explain how to use the page mapping logging area. The rest of address mapping information, that is, the information of what logical pages each of the log blocks buffers has to be saved up somewhere of flash memory. In fact, the size of this information is very large, therefore it is too expensive to store this information wholly whenever the page mapping table is updated. This is why we devise a new logging technique to store this-like page mapping information efficiently, which is described in details in Section 3.3.

3.3 Logging in FAST FTL

We described why the page mapping logging area on flash memory is needed in Section 3.2. Here, we explain how the page mapping information is stored and managed in this logging area. It is not efficient to store the whole data of the page mapping table for log blocks every update. Instead, we employ the checkpoint technique, which will acts as follows. As page updates from the file system are done onto the log blocks, the page mapping table for log blocks will grow. The moment the first N log blocks are exhausted, the first N entries of the page mapping table are output into the page mapping logging area. Here N determines the number of these entries whose total size approximates to the page size. In this way, the next N entries will be written into the page mapping logging area. This log containing N entries is called 'FASTcheck'. If later the log

block buffer becomes full, the first victim log block selection has to be performed. Accordingly, the entry information in the first FASTcheck corresponding to this victim log block becomes invalid and afterwards, if the first N victim selections are done, the first FASTcheck in the page mapping logging area becomes invalid completely. If the logging area becomes full by writing a lot of FASTchecks, a new free block is allocated for the new page mapping logging block and then all valid FASTchecks of the old logging block are copied into the new logging block. The old logging block is returned to e.g., the garbage collector.

Since checkpoints are performed per N log blocks, that is, N entries of the page mapping table, the entire page mapping table cannot be persistently stored in flash memory. If a system failure occurs at the time when the last N entries of the page mapping table is not yet accumulated after the last checkpoint is done, this uncheckpointed entries has to be recovered by scanning the corresponding log blocks and extracting their metadata. We will handle this-like recovery issue in the next subsection.

3.4 Recovery in FAST FTL

Prior to the description of crash recovery in FAST, we first assume that a crash never happens while FAST performs a full merge operation after a victim log block is selected. It is noted that a full merge operation consists of multiple individual merge operations. If a crash occurs during full merge, consistency between the block mapping table and the page mapping table cannot be guaranteed and therefore we have to make more efforts to solve this problem. So we plan to handle this complicated issue in the future work. The crash recovery is processed as follows. First, we recover the block mapping table for data blocks from the block mapping logging area and then block mapping information belonging to the page mapping table for log blocks. It is noted that the page mapping table consists of block and page mapping information. Next, we recover the rest of the page mapping table, that is, page mapping information by reading only valid FASTchecks from the page mapping logging area. As mentioned in Section 3.3, the uncheckpointed entries of the page mapping table do not exist in the page mapping logging area. Accordingly, we have to scan log blocks corresponding to these entries and then extracting page mapping data from them. As a result, we can get a complete page mapping table for log blocks.

3.5 Reforming the Merge Operation in FAST FTL

During crash recovery, we must make two block and page mapping tables equivalent with ones just before the crash occurs. However, for the page mapping table, the recovered table may be different from the last table in main memory. This is because in-memory page mapping table can be updated due to full merge operations for victim log blocks even after a series of checkpoints are performed. The problem resulting from this-like difference is performing merge operations for 'invalid' pages in the victim log blocks. In [4], the merge operation generates a new data block, followed by setting all pages joining in this generation

LBN	PBN	MT
0	30	10
1	52	***20***
2	32	15

iv: invalid

PBN	VT	*LSN List*
520	**20**	*iv, iv, iv, iv*
501	11	*iv, 201, 202, 253*
502	12	*iv, 97, 108, 301*

(a) Merge operation before the crash.

LBN	PBN	MT
0	30	10
1	52	**20**
2	32	15

iv: invalid

PBN	VT	*LSN List*
520	**20**	*iv, iv, iv, iv*
501	11	*5, 201, 202, 253*
502	12	*6, 97, 108, 301*

(b) Merge operation after the crash.

Fig. 2. Reforming the Merge Operation in FAST FTL

with invalid on the page mapping table for log blocks. However, these invalid marks cannot be reflected on already checkpointed entries of the page mapping table. So, there cannot exist invalid marks in the newly recovered page mapping table after crash recovery, resulting in useless merges on invalid pages. In order to prevent useless merges like this, we try to reform the merge operation in FAST FTL by devising two terms, VT(Victim Timestamp) and MT(Merge Timestamp). Each log block's VT in the page mapping table points to its turn in which it will be a victim log block, while each data block's MT in the block mapping table indicates which merge generated it. We define that MT is VT plus the number of log blocks. For example, in Fig. 1, VT=10 of the first entry in the page mapping table indicates that current log block will be tenth victim later. MT=13 of the second entry in the block mapping table means that the data block of PBN=30 was generated due to the third victim, since the number of log blocks is ten and so VT is three.

Now let us reform merge operations occurring after crash recovery by using VT and MT. Fig. 2(a) shows that before a crash occurs, the log block of PBN=500 in Fig. 1 was chosen as a victim, next a full merge operation was done for this victim log block, and finally a new log block was allocated from the free block list. The full merge operation consists of multiple individual merges, so in Fig. 1, the pages of LSN=4, 5, and 6 in all log blocks will be used in generating a new data block of LBN=1(assuming that a block has four page). Accordingly, the data block of LBN=1 in Fig. 1 allocate a new physical block of PBN=52 and also MT changes from 13 to 20, because VT of the victim log block is 10. In the page mapping table, invalid marks are made for the participants for the new data block. Let us assume that after Fig. 2(a), a system failure happened. According to our recovery strategy, two tables of Fig. 2(b) will be made. However, the page mapping table of Fig. 2(b) differs in that of Fig. 2(a). This is because invalid marks did not be reflected on the former. In case that the log block of PBN=501 in Fig. 2(b) will be selected as a victim later, useless data block will be created for LSN=5 and 6 in all log blocks. This problem can be solved by using VT and

MT. Before create a new data block by performing individual merge operation like LSN=5 and 6 of Fig. 2(b), we first find the data block with the same LBN in the block mapping table(LBN=1 in Fig. 2(a)) and then compare MT with VT. If MT is larger than VT, then we skip current individual merge operation, since this indicates that some victim already made current data block for same LSNs.

4 Evaluation

Runtime Overhead for Mapping Management. In this section, we evaluate the mapping management overhead using the total write overhead between LTFTL based FAST scheme and our FAST scheme under the OLTP workload.

In the case of LTFTL scheme in the FAST FTL, each N_{lt-log} logs will be written as an mapping log entry. Moreover, if the total log size exceeds the $N_{lt-threshold}$, FTL must store the whole mapping table consisted with N_{table} pages. Therefore, $Cost_{LTFTL}$, the write overhead of LTFTL based FAST, is as follow:

$$Cost_{LTFTL} = \frac{N_{log}}{N_{lt-log}} + \left\lfloor \frac{N_{log}}{N_{lt-threshold}} \right\rfloor \times N_{table} \tag{1}$$

If the FAST FTL uses FASTcheck scheme, each N_{latest} merge operation will write a page as the N_{latest} block's mapping table in the FASTcheck block. In this case, $Cost_{FASTcheck}$, the write overhead cost using FASTcheck, is hear:

$$Cost_{FASTcheck} = \frac{N_{merge}}{N_{latest}} \tag{2}$$

From this notation, we calculated the mapping management cost between the LTFTL based FAST and FASTcheck based FAST. Under the total 3,594,850 write operation, $Cost_{LTFTL}$ was 30,544 and $Cost_{FASTcheck}$ was 3,511.

Recovery Overhead. At the recovery phase, any FTL scheme must read the address table and its changed table log. In our approach, FTL has two phase read operation as we mentioned at Section 3.4. The N_{table} FASTcheck area pages must be read at the first read phase, and average $(N_{latest} \times N_{page})/2$ pages has to be read for the uncheckpointed information. The LTFTL approach also read N_{table} table pages, $(N_{lt-threshold}/N_{lt-log})/2$ log pages, and $N_{lt-lot}/2$ pages' spare area at the recovery time. However the $N_{lt-threashold}$ would be increased if the log area size became large, our approach has suitable read count for the recovery operation.

5 Conclusion

In this paper, we proposed an efficient power-off recovery scheme for a hybrid FTL scheme, FAST. Instead of writing the new address mapping information for

every page write operation, our FASTcheck scheme checkpoints the address mapping information periodically, and can recover the mapping information which is generated since the last checkpoint. As shown in Section 4, the saving in writing the metadata during the normal mode in FASTcheck far outweighs.

We plan three future works. The first one is to implement our scheme in real boards and to verify its efficient. The second is to optimize the excessive read overhead in incremental recovery phase. And finally, we will investigate whether our scheme can be applicable to other FTLs.

References

1. A. Ban. Flash file system, April 4 1995. US Patent 5,404,485.
2. A. Ban. Flash file system optimized for page-mode flash technologies, August 10 1999. US Patent 5,937,425.
3. Tae-Sun Chung, Myungho Lee, Yeonseung Ryu, and Kangsun Lee. PORCE: An efficient power off recovery scheme for flash memory. *J. Syst. Archit.*, 54(10):935–943, 2008.
4. Jesung Kim, Jong Min Kim, S.H. Noh, Sang Lyul Min, and Yookun Cho. A space-efficient flash translation layer for CompactFlash systems. *Consumer Electronics, IEEE Transactions on*, 48(2):366 –375, May 2002.
5. Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embed. Comput. Syst.*, 6(3):18, 2007.
6. K. Sun, S. Baek, J. Choi, D. Lee, S.H. Noh, and S.L. Min. LTFTL: lightweight time-shift flash translation layer for flash memory based embedded storage. In *Proceedings of the 8th ACM international conference on Embedded software*, pages 51–58. ACM, 2008.