electronics

MDPI

Article

A Crash Recovery Scheme for a Hybrid Mapping FTL in NAND Flash Storage Devices

Jong-Hyeok Park [1], Dong-Joo Park [2,*], Tae-Sun Chung [3] and Sang-Won Lee [1]

[1] College of Software, Sungkyunkwan University, Suwon 16419, Korea; akindo19@skku.edu (J.-H.P.); swlee@skku.edu (S.-W.L.)
[2] School of Computer Science and Engineering, Soongsil University, Seoul 06978, Korea
[3] Department of Artificial Intelligence, Ajou University, Suwon 16499, Korea; tschung@ajou.ac.kr
* Correspondence: djpark@ssu.ac.kr

Abstract: An FTL (flash translation layer), which most flash storage devices are equipped with, needs to guarantee the consistency of modified metadata from a sudden power failure. This crash recovery scheme significantly affects the writing performance of a flash storage device during its normal operation, as well as its reliability and recovery performance; therefore, it is desirable to make the crash recovery scheme efficient. Despite the practical importance of a crash recovery scheme in an FTL, few works exist that deal with the crash recovery issue in FTL in a comprehensive manner. This study proposed a novel crash recovery scheme called FastCheck for a hybrid mapping FTL called Fully Associative Sector Translation (FAST). FastCheck can efficiently secure the newly generated address-mapping information using periodic checkpoints, and at the same time, leverages the characteristics of an FAST FTL, where the log blocks in a log area are used in a round-robin way. Thus, it provides two major advantages over the existing FTL recovery schemes: one is having a low logging overhead during normal operations in the FTL and the other to have a fast recovery time in an environment where the log provisioning rate is relatively high, e.g., over 20%, and the flash memory capacity is very large, e.g., 32 GB or 64 GB.

Keywords: crash recovery; address mapping; logging; NAND flash memory

check for updates

Citation: Park, J.-H.; Park, D.-J.; Chung, T.-S.; Lee, S.-W. A Crash Recovery Scheme for a Hybrid Mapping FTL in NAND Flash Storage Devices. *Electronics* **2021**, *10*, 327. https://doi.org/10.3390/electronics10030327

Academic Editor: Manuel E. Acacio
Received: 18 December 2020
Accepted: 26 January 2021
Published: 1 February 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

As we have witnessed in the past decade, flash memory is deployed as an alternative data storage for mobile devices, PC/laptops, and even enterprise servers. However, mainly because of the "erase-before-write" characteristic of flash memory, most flash-based storage devices come with a core software module, namely, a flash translation layer (FTL). Since the performance of a flash memory device is heavily determined by the FTL, numerous FTLs have been proposed [1–8]. In particular, taking into account the ever-increasing capacity of flash memory devices, the efficiency of FTLs becomes more and more important. FTL is basically responsible for address mapping, garbage collection, and wear-leveling, and according to the address-mapping method, the existing FTLs can be largely categorized into block mapping, page mapping, and hybrid mapping [9].

Meanwhile, most existing works on FTLs have been mainly focused on issues such as performance and wear-leveling. In contrast, despite its practical importance, the recovery from a power-off failure in FTLs has not been paid much attention to.

In particular, there is no work that deals with crash recovery in FTLs comprehensively. Moreover, many flash-based solid state drive (SSD) vendors supplement their own crash recovery algorithms, but none of these have been published since they are reluctant to disclose their solutions. We believe that there is great room for improvement of the tightly coupled crash recovery algorithm with FTL, and it is promising to satisfy both the efficiency of crash recovery and provide a lightweight overhead during normal execution periods.

In this study, we proposed a new crash recovery algorithm called FastCheck for a well-known hybrid mapping FTL, namely, Fully Associative Sector Translation (FAST) [5]. FastCheck exploits the characteristics of FAST, where the log blocks in a log area are used in a first input first output (FIFO) manner and guarantee FTL's metadata integrity from a sudden power failure by employing a checkpoint approach.

In comparison to the existing FTL recovery schemes [10–12], FastCheck makes three major contributions, as given below:

- The first is the low metadata logging overhead during normal operations in FTL. In fact, FastCheck requires a nominal size of mapping metadata to be written per data page write. In terms of the additional write overhead for recovery, FastCheck is more efficient than the existing algorithms.
- The second is the fast recovery time. By periodically writing FTL metadata in a designated metadata area, its recovery phase finishes as early as possible without as many metadata read operations, and then FAST can resume in its normal mode.
- Finally, this research is a first comprehensive study of an intuitive and efficient crash recovery scheme for a hybrid mapping FTL.

The rest of this paper is organized as follows. Section 2 describes the related work of flash memory and the address-mapping table management, and then Sections 3–5 explain the proposed crash recovery scheme. Section 6 shows the simulation results with a comparison between the proposed method and the existing ones. Finally, Section 7 concludes this paper.

2. Background and Related Work

2.1. FAST FTL: An Overview and Recovery Issues

With flash memory, no data page can be updated in place without erasing the block containing the page. This characteristic of flash memory is called the "erase-before-write" limitation. In order to alleviate the erase-before-write characteristic, most flash memory storage devices are equipped with a software or firmware layer called an FTL. A flash memory storage device can be seen as a hard disk to the host (e.g., file system) by an FTL. One key role of an FTL is to redirect a logical page write from the host to a clean physical page, and to remap the logical–physical page address mapping when the write operation is an update operation and is predicted to incur an erase operation.

Via the granularity of address mapping, FTLs can be classified into three categories: page-mapping FTLs [1,13,14], block-mapping FTLs [15], and hybrid mapping FTLs [2–8]. FAST [5] is one of the hybrid FTL schemes, which was originally designed to improve the performance of small random writes. In FAST, 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 [3] as well.

Figure 1 shows the architecture of FAST. Flash memory in FAST is logically partitioned into data blocks, log blocks, and a metadata area. User data are stored in data blocks and log blocks are used as a buffer for new writes. Additionally, system information including mapping information is stored in the metadata area, which is divided into a map directory and map blocks. Section 3.2 will introduce a new FAST architecture for the crash recovery, where the checkpoint blocks, data block merge sequency number (dbMSN) and log block merge sequence number (lbMSN) (red-colored parts in Figure 1), are added.

There are two kinds of address-mapping tables used in the FAST scheme: a block-mapping table and a log-page-mapping table. The block-mapping table is for logical-to-physical block address-mapping translations. For example, in Figure 1, the logical block with logical block number (LBN) = 1000 is mapped to the physical block with physical block number (PBN) = 30. Next, the log-page-mapping table stores page-level mapping information for the log blocks. Whenever a new write is directed to one of the log blocks, its logical page number (LPN) is added to the LPN list of this log block in the log-page-mapping table.

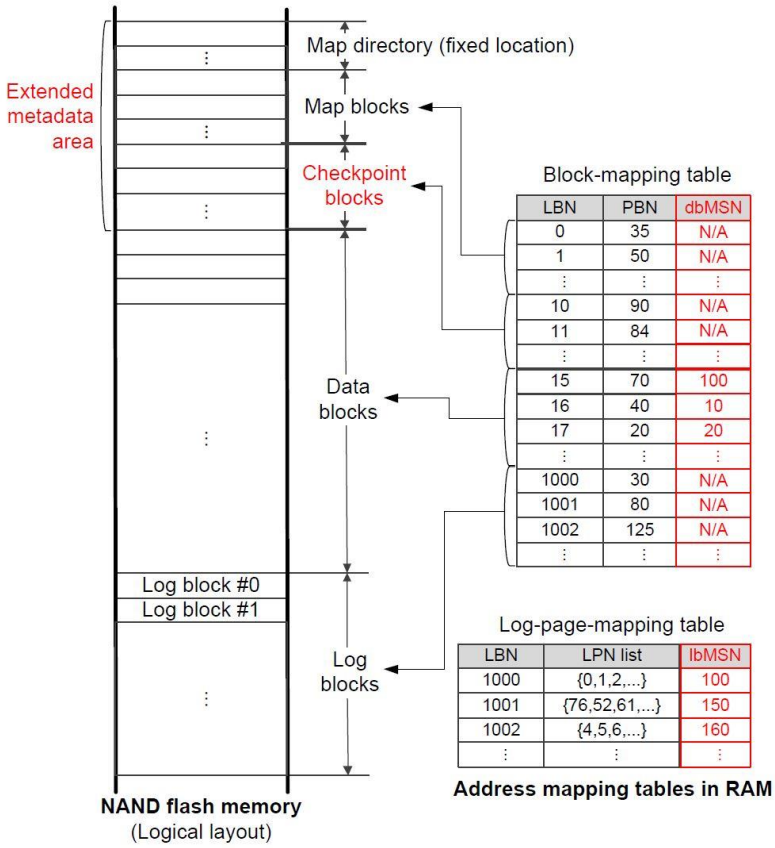


Figure 1. FAST architecture. dbMSN: data block merge sequence number, lbMSN: log block merge sequence number, LBN: logical block number, LPN: logical page number, PBN: physical block number.

In FAST, a merge operation results from reclaiming free space from log blocks that are currently full. In order for a new write to succeed, a clean page in the log blocks is necessary. However, if no clean pages exist, a log block is chosen as a victim in a round-robin fashion, followed by multiple merge operations for this victim log block. The number of merge operations is determined by the number of data blocks that valid pages from the current victim log block belong to. For each of these data blocks, up-to-date pages pertaining to the data block, which are in either the log blocks including the current victim block or the data block, are copied to a new data block, and then new address-mapping information is reflected by changing the block-mapping table. After all the merges are finished, the old data blocks, as well as the victim log block, are returned to the free block list.

Figure 2 shows how to perform the merge operation in FAST. In step (1), the victim log block contains valid pages belonging to three different data blocks, that is, LBN 0, 1, and 2 (we assume that a block is composed of four pages). Thus, three merge operations will proceed. In the figure, the merge operation for the data block of LBN = 2 makes a new data block to which all latest valid pages, that is, 8, 9′, 10′′, and 11 are copied. After three merge operations are completed, like step (2), old data blocks taking part in the merges are replaced with new data blocks by changing the block-mapping table. Then, the old data blocks are returned to the free block list and a new log block is allocated to the buffer of log blocks and the log-page-mapping table is changed. Furthermore, the victim log block is returned to the free block list.

While a new write from the host is being handled, some metadata, such as address-mapping information maintained by the FTL, may be modified in the main memory. However, when a sudden system crash is encountered, the changed metadata may be lost. For this reason, some FTL recovery algorithms are proposed [2,9,12,16–19]. The original FAST scheme has adopted a Block Associative Sector Translation (BAST)-like recovery

scheme [3]. In this technique, whenever the block-mapping table is modified during a merge operation, the updated block-mapping table is promptly copied to a separated metadata area called a "map block." On the other hand, in the case where a new write is performed to an empty page of log blocks, not causing merges, its logical page number is inserted into its spare area. The log-page-mapping table can be reconstructed by reading logical page numbers from all the pages of log blocks during crash recovery. However, this recovery scheme can cause a few recovery issues that were not completely taken into account in the original FAST paper.

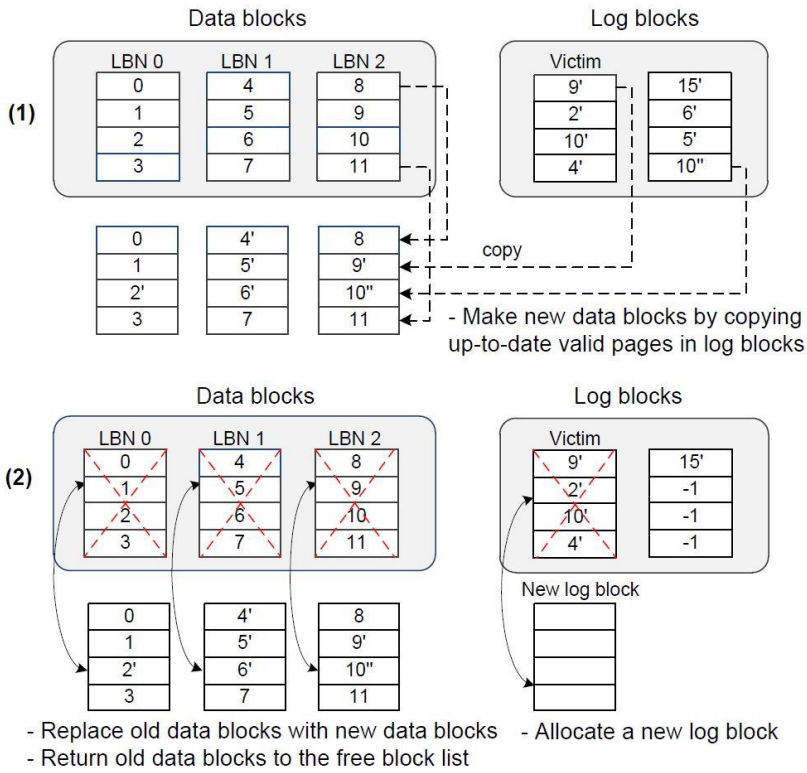


Figure 2. Merge operations in FAST.

If a power-off failure occurs, in order to rebuild the log-page-mapping table, FAST must scan whole pages in the log blocks during the recovery. This requires a very long time. With this traditional recovery technique, FAST could get up-to-date valid pages in the log blocks by scanning entire pages from them and reading the logical page number in the spare area of each page. However, there is a limitation to recognizing whether the obtained page is currently valid or already merged out. This problem makes the recovery costly and gives rise to unnecessary merge operations during the recovery. Furthermore, one concern that makes the recovery in FAST more complex concerns multiple merge operations resulting from a new write, such as that given in Figure 2. Since a system failure can happen at any time during steps (1) and (2) in Figure 2, it is difficult for the traditional recovery technique to guarantee the system's safety

Based on these observations, this study proposed a more efficient and comprehensive recovery scheme for FAST, which takes into account its architectural characteristics and complex internal merge operations

2.2. Related Work: Crash Recovery in FTLs

Several recovery mechanisms [17–20] have been proposed to roll back the mapping information, but since they are tightly coupled with the upper software layer (e.g., the file system), they can be applied only to the dedicated flash file systems, such as the log-structured file system. Of course, decoupled FTLs like FAST cannot adopt those recovery

mechanisms. This is because, in the decoupled FTLs, the file system layer cannot access the FTL's internal data structures, such as address-mapping tables.

In order to ensure the data consistency in flash memory, BAST, one of the hybrid mapping FTLs, writes metadata, namely, logical page numbers, to the spare area of each of the pages in the provisioning area [13]. When a system failure occurs, BAST recovers the address-mapping information by reading all the metadata stored in the pages of the provisioning area. However, during the recovery, BAST has to read lots of pages belonging to the provisioning area; therefore, it takes a long time to finish recovering. Moreover, considering the trend of high-capacity flash memory storage, this recovery policy seems to be unattractive. Hence, a new efficient recovery policy needs to be devised for the hybrid address-mapping FTLs like FAST.

On the other hand, some researchers have addressed FTL-decoupled recovery schemes [8,10,11,21–24]. First, Lightweight Time-shift Flash Translation Layer (LTFTL) [11] is a page-mapping FTL with an internal recovery scheme. Whenever address mapping is changed, LTFTL keeps the changed address log on buffer in the RAM, and periodically, it performs checkpoints by writing a set of logs in the RAM to the log section of flash memory in a page unit. In this way, when a crash occurs, LTFTL can rapidly restore the latest address-mapping information through a small number of page accesses to flash memory. Since FAST uses a page-mapping table for log blocks, it seems that this logging approach with checkpoint can be applied to FAST. However, there exist the following problems. First, due to the limited size of the log section, FAST cannot store all the log records under an environment where many updates in the address mapping are occurring. To solve this problem, whenever the overall size of the current log records exceeds a given threshold value, FAST dumps up-to-date page-mapping information to the flash memory. Unfortunately, this incurs performance degradation. Second, as system failure may occur during each step of the merge operation of FAST, to guarantee the system's safety, the corresponding log records should be flushed to flash memory, although the total size of all log records does not reach the page size. These flush operations result in not only low space utilization in the log section area but also performance degradation.

Power-Off Recovery Scheme (PORCE) [10] is a power-off recovery scheme for flash memory. PORCE divides write operations in the FTL into two categories: writes without reclamation and writes with reclamation, and provides recovery protocols in each case. In particular, during the reclaiming operation, PORCE writes the reclaiming start log before the reclaiming operation and the reclaiming commit log after the reclaiming operation in the transaction log area of flash memory. However, PORCE would not be applicable to the case of a complex merge operation in FAST because a merge operation may incur many address-mapping changes in data blocks and log blocks.

Sudden Power-Off Recovery (SPOR) [22] was proposed as an efficient crash recovery scheme for page-mapping FTLs, e.g., Demand-based Flash Translation Layer (DFTL) [21]. It maintains three types of caches, that is, a root page, L2/L3 mapping pages, and summary pages in order to synchronize the on-RAM and on-NAND data promptly and efficiently, leading to fast crash recovery. However, different from SPOR based on a DFTL, which is one of page-mapping FTLs, our crash recovery scheme is optimized to FAST, one of the hybrid mapping FTLs.

Motivenga [23] proposed an efficient, fault-tolerant crash recovery scheme called Fault Tolerant Recovery Mechanism (FTRM), which is coupled with an effective hot/cold page separation algorithm. FTRM focuses on efficiently handling inconsistencies between page mappings in RAM and flash memory in page-mapping FTLs for fast crash recovery. However, we propose a crash recovery scheme that is appropriate for a hybrid address-mapping FTL like FAST. The main difference between the two recovery schemes is that our recovery scheme is based on logging with checkpoints.

Choi [24] proposed a general-purpose FTL framework that includes a crash recovery module called Hierarchically Interacting Logs (HIL). HIL handles crash recovery through two phases of borrowing ideas from the database: structural recovery and functional

recovery. However, our crash recovery technique is specialized for hybrid mapping FTLs and solves crash recovery through two phases, namely, the restoration phase and the consistency check phase.

It should be noticed that this paper is an extended version of the Software Technologies for Embedded and Ubiquitous Systems (SEUS) conference paper [25], which mainly dealt with the idea of how to recover address-mapping information from the crash and shows a little qualitative performance analysis. In this paper, we provide a detailed description of the idea and the recovery algorithm to make it easy to understand them; furthermore, we added experimental results based on diverse simulations, as well as supplemented the qualitative performance analysis.

3. FastCheck: A Recovery Scheme for FAST

3.1. Checkpoint-Based Approach

An FTL-supported recovery system has to provide two essential functionalities: (1) minimizing the overhead due to additional metadata writes for the recovery and (2) fast restoration of the metadata lost by failure. In particular, with a hybrid FTL like FAST, it is very important to restore the log page-mapping table to the consistent state that existed before the failure. However, as mentioned in Section 2.1, if every change in the log-page-mapping table is not persistently saved in flash memory, it is hard to achieve a fast recovery because all the pages in the log blocks should be scanned during the recovery.

Furthermore, under the FAST scheme, when the log blocks are full, a victim block is chosen and then one or more merge operations are performed, which will result in many updates of the log-page-mapping table. Whenever the log page-mapping table changes, we may write the overall table to flash memory. However, this strategy will produce a big overhead during normal execution time. Instead, we devise a way of performing checkpoints (i.e., FastCheck) that periodically output log-page-mapping information into flash memory. Since a checkpoint is performed in the unit of a page, the overhead of writing mapping data during normal execution time would be very small. In addition, FastCheck can avoid performing checkpoints repeatedly for the same part of the log-page-mapping table; therefore, high space utilization for the metadata area of flash memory is achieved. This provides an optimized recovery to the FAST, since at the recovery time, FastCheck does not have to read data from the metadata area that is more than the size of the log-page-mapping table.

3.2. Revised FAST Architecture

In this section, we describe the metadata area in the flash memory and the mapping tables in the main memory, some parts of which should be modified for power-off recovery, in contrast with the original FAST architecture.

First, we extended the metadata area used in the FAST for power-off recovery. In Figure 1, the previous metadata area consisted of two parts: "map directory" and "map blocks." Part of the map blocks permanently stores almost all kinds of block mapping information that the block-mapping table of FAST keeps. Part of the map directory at a fixed location saves the information regarding what blocks are used by part of map blocks. As shown in Figure 1, we added a new part of the "checkpoint blocks" to the previous metadata area, where the log-page-mapping table in RAM is logged in the fashion of a checkpoint. In the following, we illustrate the above three parts of the metadata area in detail.

- Map directory: The map directory stores the block-mapping information for map blocks, that is, the information of what physical blocks are allocated to the map blocks. The FAST accesses the map directory first when it starts up. Since it has to access the map directory without any mapping information, the map directory uses some fixed physical blocks (in order to make provisions for bad blocks in the map directory, it is necessary to have extra blocks or to adopt the redundant array of independent disks (RAID)-0 technique, which divides the map directory into two logical areas). There

may be no space in the map blocks when the FTL writes the mapping information modified, for example, by updating the block-mapping table due to multiple merge operations. In this case, it is required to allocate a new block to the map blocks, leading to the update of the block-mapping information for them. Accordingly, this modified information should be written to the map directory.

- Map blocks: The map blocks preserve all block-mapping information that is maintained for the FAST. The block-mapping table in the main memory can be updated due to merge operations or block-mapping information can be changed because the checkpoint blocks become full and thus a new block is allocated to it. Whenever these events happen, modified block-mapping information should be recorded somewhere on flash memory, which is the map blocks.
- Checkpoint blocks: The checkpoint blocks periodically record partial information of the log-page-mapping table in a checkpoint fashion. FAST writes data in the log area sequentially, and during a merge, selects a victim log block in a round-robin way. Therefore, we can restore the log-page-mapping table before the failure by reading sequentially written mapping information in the reverse order.

Furthermore, the block-mapping table and the log-page-mapping table used in the original FAST scheme was slightly modified. As shown in Figure 1, two new columns, namely, dbMSN and lbMSN, were added to the block-mapping table and the log-page-mapping table, respectively. These additional data are used to guarantee the correctness of merges during the recovery, which is discussed in detail in Section 5.2.

4. Logging

In this section, we describe how to perform logging operations against system crash on flash memory during a normal execution period, where the operations are divided into three types: (1) logging on the checkpoint blocks, (2) logging on the map blocks, and (3) logging on the map directory. The first type of logging involves outputting the information on the log-page-mapping table to the checkpoint blocks and the second logging involves outputting the information on the block-mapping table to the map blocks, except for block-mapping information on the map blocks, which is written to the map directory by the third logging. For reference, in the case of system failures during logging, a metadata write with the unit of a page is required for guaranteeing the atomicity of the metadata write (generally, this issue can be solved by adopting a CRC/ECC engine in the flash controller). Figure 3 illustrates the above three types of logging on the metadata area of the flash memory.

4.1. Logging on the Checkpoint Blocks

As shown in Figure 3a, the log blocks are logically divided into several log regions. The size of a log region is determined by how much information in the log-page-mapping table is packed up into a page. After pages in the log blocks belonging to the first log region are all used by a sequence of writes from the upper file system, part of the log-page-mapping table corresponding to this log region is output to the checkpoint block area. For the rest of the log regions, the same actions are done one after another. This checkpoint approach is different from the strategy that involves all information in the log-page-mapping table being flushed out whenever an update happens to this table. It may be impossible to adopt the latter approach in a large-scale storage system since a large log-page-mapping table should be maintained, and accordingly, the cost of a flush-out is very expensive. The unit of a checkpoint is equivalent to the size of a log region, which is, as mentioned before, configured to guarantee a one-page write when outputting the metadata to the flash memory. Suppose that the size of a page is 4 KB, the size of a block is 512 KB [26], and the space size of an LPN value kept in the log-page-mapping table is 4 B. A page can keep about 1024 LPN values, and thus, a log region consists of eight log blocks, each of which has 128 pages. Since the last checkpoint, newly updated mapping information in the log-page-mapping table would be lost upon a system failure because

it is kept only in the RAM, not persistently stored in flash memory. However, this lost information can be easily and promptly recovered during the crash recovery.

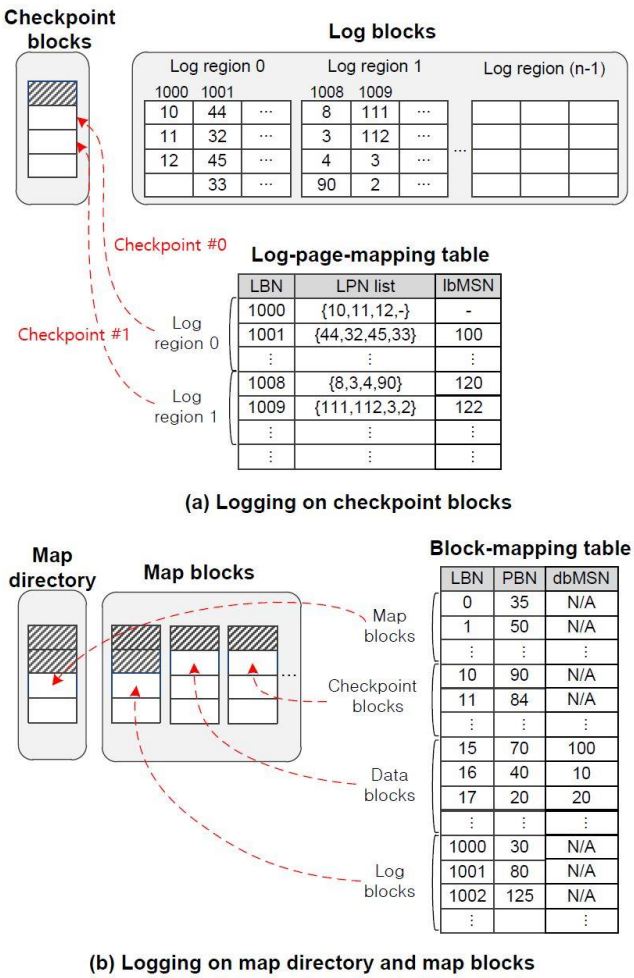


Figure 3. Logging operations in FAST.

4.2. Logging on the Map Blocks

All block-mapping information maintained by FAST need to be preserved for the recovery after a system failure. If all log blocks are exhausted by many data updates, a victim log block is selected and then numerous merge operations for valid logical data pages in the victim are performed. The victim log block is replaced with an empty block, which is allocated to the pool of log blocks from the free block list, leading to the change of an entry value on part of the log blocks in the block-mapping table. Furthermore, while a sequence of merge operations is being executed, the corresponding data blocks are useless, and thus, new data blocks are allocated. In this case, mapping information on part of data blocks in the block-mapping table may be changed. Meanwhile, frequent checkpoints make the checkpoint block area full, in which case, one of the checkpoint blocks should be exchanged with a new free block. Accordingly, part of checkpoint blocks in the block-mapping table has to be updated. As shown in Figure 3b, whenever at least one of the above three cases occurs, logging is performed in the map block. As mentioned above, for the atomicity of the metadata write, the information on each part of the block-mapping table is placed in one of the pages pertaining to the map blocks.

However, in an environment where the large-scale flash memory is used, the size of the part of the data blocks from the block-mapping table may be too large to pack this part's information into one page. In this case, we divided the overall data blocks into data regions of the same size and then allocated a block in the map blocks to each data

region. If mapping information on one of the data blocks in a data region is changed, of all metadata on the part of the data blocks in the block-mapping table, only mapping information corresponding to this data region in the block-mapping table is output to one of the map blocks, which is mapped to this data region.

4.3. Logging on the Map Directory

Figure 3b illustrates that the map directory preserves block-mapping information for the map blocks. The map blocks will be full in the future; therefore, a new block will have to be allocated to the map blocks. Accordingly, part of the map blocks in the block-mapping table will be changed and hence, new mapping information needs to be recorded to the map directory.

4.4. The Input/Output (I/O) Cost in Logging

The write cost on flash memory is expensive; therefore, the cost resulting from logging for recovery must be small compared with that of the overall normal writes. From this point of view, our logging approach is very efficient. First, logging on the map blocks is carried out for each merge, which is required when the buffer of log blocks becomes full. When logging, only an additional page write is needed. Generally, a merge operation requires a lot of page reads/writes and an erase for an old data block. Compared with this large cost of a merge, the overhead arising from logging on the map blocks is very small. Second, logging on the map directory is performed at the time when pages of the last map block are all exhausted by logging on the map blocks. Of course, the overhead is expected to be small; therefore, logging on the map directory has a small effect on the overall performance of normal writes. Finally, since it is small in the two logging cases above, the cost by logging on the checkpoint blocks is also small. This is because such logging happens whenever the space equivalent to a log region is filled with log data written by data updates.

5. Recovery

In this section, we explain how to recover two address-mapping tables using logged data after a system crash. In order to explain the recovery of the FastCheck scheme, we made a small complex situation that describes when a system crash happens. A write operation issued from the upper layer to an FAST FTL may give rise to an overwrite on a data block; therefore, FAST will redirect this write to one of the log blocks. However, in the case where the log blocks are full, a victim log block among them should be selected, and then multiple merge operations within the victim should be performed. When a system crash arises, the FAST can be in a variety of situations. In various recovering situations, we focus on recovering the FAST that was involved in carrying out merge operations at the time of the crash. This is because they touch most of the metadata used in the FTL and flash memory since this makes the recovery more complex than other situations.

Now we illustrate the recovery process of FastCheck using the example in Figure 4. Before proceeding, we assume that a system crash happens right after the first merge in the victim log block completes, as shown in Figure 4. In the figure, the victim log block is currently the log block of LBN = 1004, where two merges for the logical blocks of LBN = 10 and 11 are expected to be executed. (The log pages of LPN = 40 and 41 belong to the log block of LBN = 10 since 40/4 = 10 and 41/4 = 10. Furthermore, because 46/4 = 11, the log block of LBN = 11 contains a log page of LPN = 46. For reference, LPN = −1 means the corresponding log page was already merged, so no additional merge was needed.) Another assumption was made in Figure 4. As mentioned in Section 4, log blocks are divided into logical log regions. The number of log blocks in the figure is six and they are divided into two log regions, with each having a size of three. Accordingly, the log-page-mapping table in the figure is divided into two log regions. The log region 0 of this table was already checkpointed to the checkpoint block area in the flash memory. After that, the log region 1 was being used as a buffer for overwrites issued by the upper layer. The first log block

(LBN = 1003) in this region was already selected as a victim. Then, a few merges within this victim were done and the first log block was replaced with a new physical log block, which was used as storage for dealing with overwrites from the upper layer. After that, because the first log block became full, the second log block was selected as a victim, followed by the first merge within this victim and then a system crash arose.

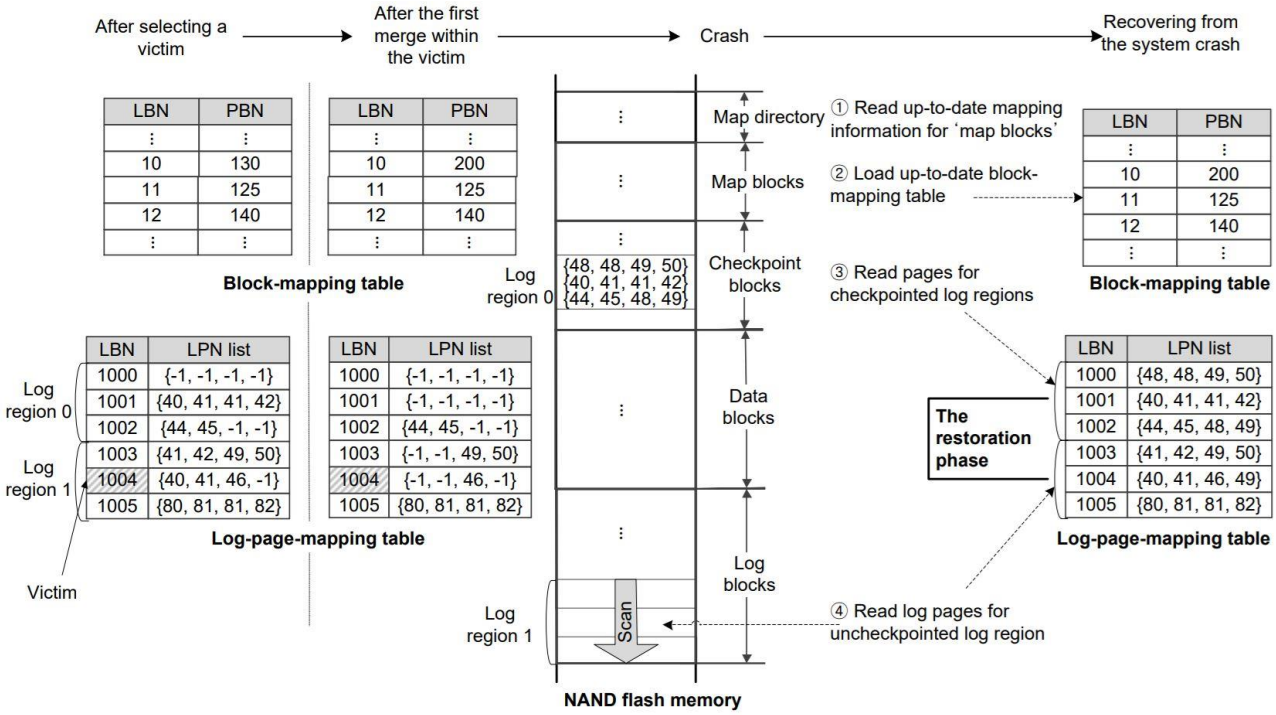


Figure 4. Recovery process.

5.1. Restoring the Block-Mapping Table

At the startup time, by accessing the map directory, which exists at a fixed location of the flash memory, the FastCheck scheme first finds out where the up-to-date block-mapping table is stored in the map block area and then loads it in the RAM (steps 1 and 2 of Figure 4).

As described in Section 4, the block-mapping table in the RAM is frequently updated due to many merges in the log blocks. After each update, it is logged in the map block area. However, a system crash may happen before finishing the logging; therefore, the block-mapping table modified by the current merge may not be recorded persistently on the flash memory. In this case, after loading the old block-mapping table from the map block area, the last merge performed before the system crash is re-executed, if necessary.

5.2. Restoring the Log-Page-Mapping Table

FastCheck uses a two-phase algorithm to restore the log-page-mapping table after a system crash. In the first phase, called the restoration phase, FastCheck reads the address-mapping data for an already-checkpointed log region from the checkpoint block area and the address-mapping data for a not-yet-checkpointed log region from the log block area and then builds a temporary log-page-mapping table, which may not be consistent. For this reason, the second phase, called the consistency check phase, is required, where FastCheck checks the consistency of the log-page-mapping table that was restored temporarily and makes it consistent, if necessary. Below, we elaborate on each phase.

5.2.1. Restoration Phase

As described in Section 4, in order to cope with a system crash in the future, FastCheck periodically leaves logged data of the log-page-mapping table on the checkpoint block area. It is noted that the unit of the checkpoint is equivalent to the size of a log region. The restoration phase works as follows. First, while scanning the checkpoint block area backward, FastCheck reads all the pages in the first $(n - 1)$ log regions and fills an empty log-page-mapping table with the address-mapping information that these pages contain (step 3 in Figure 4). In order to get a completely built log-page-mapping table consisting of n log regions, it is necessary to obtain the address-mapping information that belongs to the last remaining log region. This was the active log region, and thus, its address-mapping information resided in the main memory at the time of the system crash (not yet checkpointed to flash memory). Therefore, FastCheck restores its address-mapping information by reading log blocks in the log block area corresponding to the remaining log region (step 4 in Figure 4). As a result, FastCheck can build a log-page-mapping table like the table at the bottom-right in Figure 4. However, it should be noted that the restored table at this stage is not exactly the same as the one at the time of the failure.

The log-page-mapping table restored above might still be inconsistent because it may not reflect the modifications caused by all merge operations that were executed after the last checkpoint. In Figure 4, it can be seen that the table at the bottom-right is different from the second table at the bottom-left. Furthermore, it can be observed from Figure 4 by comparing these two tables that two merge operations for LBNs 10 and 12 were executed since the last checkpoint (for log region 0) was taken. In order to restore the log-page-mapping table exactly to the status at the time of the failure, we need a further phase called the consistency check phase.

5.2.2. Consistency Check Phase

In the consistency check phase, FastCheck makes the temporary log-page-mapping table restored by the restoration phase consistent by invalidating log pages in the log-page-mapping table, if necessary. More specifically, it sets LPN values of invalid log pages as -1 in the log-page-mapping table. For reference, after a log block is chosen as the victim and multiple merge operations for the victim log block are executed, all the log pages in the log block area that participate in those merge operations have to be marked as invalid. That is, all the corresponding LPN values in the log-page-mapping table are set to be -1. For example, in the log-page-mapping table at the bottom-left of Figure 4, the current victim is the log block of LBN = 1004, where a merge operation for LPN = 40 and 41 is executed first. As a result of the merge, the LPN values of all the log pages belonging to the logical block of LBN = 10 are set to be -1.

However, a log-page-mapping table built by the restoration phase has no information regarding what log pages in the log blocks were already merged into new data blocks. Therefore, the consistency check cannot be performed correctly. For this reason, in order to decide the validity of each log page in log blocks during the recovery, we introduce the MSN (merge sequence number) and devise a recovery algorithm using MSNs. Basically, MSN, which is a global variable, plays the role of a timestamp. It represents the total count of the merges carried out, and thus, monotonically increases upon each merge operation. According to the block type to which MSN is used, there are two types of MSN, namely, lbMSN and dbMSN:

- lbMSN: For each log block, when its last page is written to flash memory, the current MSN value is also stored in it, which is called lbMSN. The lbMSN information is also kept in the RAM-resident log-page-mapping table. Of course, during checkpointing, the lbMSN information is stored in the checkpoint block area, along with the log-page-mapping data.
- dbMSN: Just as each log block has its lbMSN, each data block keeps its dbMSN, which is in accordance with the current MSN value when a merge for a data block is completed. Similar to the lbMSN, the dbMSN information is kept in the RAM-resident

block-mapping table. At the end of the merge, this block-mapping table containing dbMSNs is logged in the map block area.

Similar to the timestamp's function, the lbMSN value attached to a log block represents when this log block was created and the dbMSN value attached to a data block makes it possible to know when this data block was created, together with when all the pages in this data block were merged into it. Figure 5 illustrates what lbMSN and dbMSN mean. In the log-page-mapping table, the meaning of lbMSN = 44 is that a log block of LBN = 1003 was newly created, and after that, new log pages were stored to this log block immediately after the 44th merge was processed. Likewise, in the block-mapping table, a data block of LBN = 10 was newly created as a result of the 45th merge, and at this moment, all valid log pages in the log block area pertaining to this data block were merged into it.

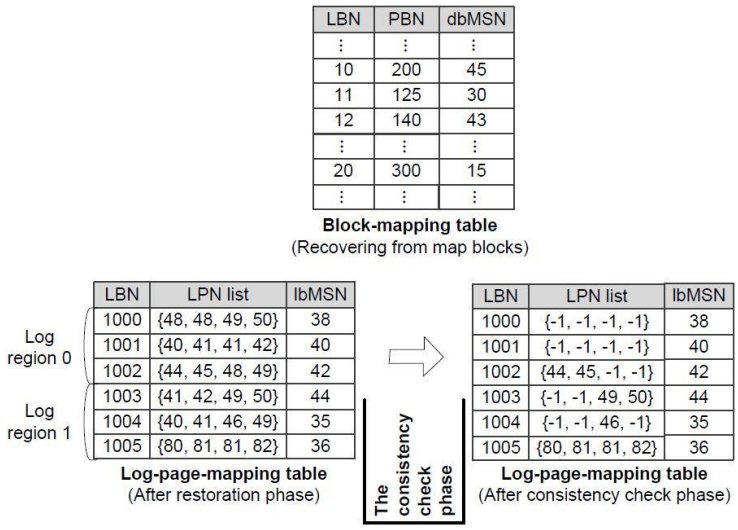


Figure 5. Consistency check phase.

The consistency check phrase starts by finding the victim log block at the time of the system crash. More specifically, it looks up the log-page-mapping table and gets the log block with the smallest lbMSN value. This is because the FAST selects a victim log block in a round-robin fashion, that is, the oldest log block. If necessary, we refer to the block-mapping table and search for the largest dbMSN value. This additional look-up informs us of how many merges were done for the found victim log block. In Figure 5, a log block of LBN = 1004 in the log-page-mapping table satisfies the condition for the victim, and in the block-mapping table, since the largest dbMSN value is 45, one merge within the found victim log block was done. In other words, log pages of LPN = 40 and 41 in the found victim log block (LBN = 1004) gave rise to the merge, resulting in the new data block of PBN = 200 in the block-mapping table.

In the following, we explain how to use lbMSN and dbMSN to check the validation of log pages in the log-page-mapping table. In order to determine the validity of log pages, we derived a simple condition using MSN as follows:

$$\text{dbMSN} \leq \text{lbMSN}. \tag{1}$$

Equation (1) means that for any log page in the log block area, if its MSN value (i.e., lbMSN) is greater than or equals to the MSN value of its corresponding data block (i.e., dbMSN), it was valid at the point of the system crash. Suppose that a log page in the log block area satisfies the condition, dbMSN $\leq$ lbMSN. As mentioned earlier, MSN plays the role of a timestamp. Therefore, it is certain that this log page was written in one of the log blocks after the last merge for its corresponding data block was executed, that is, this log page is not yet merged into its corresponding data block. As a result, this log page is currently valid. In a reverse case where a log page satisfies the condition, dbMSN > lbMSN,

this log page is undoubtedly currently invalid. For instance, we can see in Figure 5 that the log page of LPN = 40 was written to the log block of LBN = 1004 at the time of MSN = 35 and this page's corresponding data block, LBN = 10, was created by the merge at the time of MSN = 45. Therefore, we can conclude that the log page with LPN = 46 in the log block of LBN = 1004 is valid because (dbMSN = 30) $\leq$ (lbMSN = 35).

Equation (1) makes it very simple to check the validity of log pages in the log blocks. Starting from the victim log block, we scan the log-page-mapping table backward and examine whether log pages within each log block are valid or not using Equation (1). Whenever we test Equation (1) for a log page in the log-page-mapping table, we have to refer to the block-mapping table to obtain the dbMSN value of its corresponding data block. Hence, the cost is expensive. In order to reduce this cost, it is desirable that we make a group of log pages in the log-page-mapping table that share the same data block and test the validation for this group only with one look-up of the block-mapping table. For example, when testing the validation of the log page with LPN = 40 in the log block with LBN = 1004, we need a reference to the block-mapping table to get the dbMSN value of its corresponding data block, LBN = 10. By using the same dbMSN value continuously, we continue to perform the validation test for other log pages in the log-page-mapping table that share the same corresponding data block, whose LPN value ranges from 40 and 43. Because all log pages ranging from 40 to 43 do not satisfy Equation (1), they are all marked as invalid, that is, set to be -1 in the log-page-mapping table.

The two-phase recovery process of FastCheck is given as a pseudoalgorithm in Figure 6. In summary, our FastCheck scheme has two advantages. First, its recovery overhead is quite low since it requires one scanning pass for the log-page-mapping table in the RAM and a small number of NAND flash accesses. Second, the recovery algorithm is highly scalable, even for a large-scale flash SSD.

```
procedure RECOVERY_PROCESS()
    /* 1. Load address-mapping tables */
    load block-mapping table for map blocks from map directory
    load block-mapping table for checkpoint/log/data blocks from map blocks
    load checkpointed-page-mapping table (PMT) from checkpoint blocks
    /* 2. Two-phase recovery */
    /* 2-1. Restoration phase */
    for all P ∈ uncheckpointed log blocks of the log block area do
        get LBN, LPNs, and lbMSN from spare area of each page in P and then insert
        them in PMT
    end for
    /* 2-2. Consistency check phase */
    for all P ∈ log pages in PMT which are marked as valid do
        LB_P ← get LBN of log block where P exists from PMT
        DB_P ← get LBN of P
        if dbMSN(DB_P) > lbMSN(LB_P) then  // Equation (1)
            mark PMT(P) as invalid  // set PMT(P) to be −1
        end if
    end for
end procedure
```

Figure 6. Two-phase recovery algorithm.

5.3. Crashes during Recovery in FastCheck

In any recovery system, it is possible to encounter crashes repeatedly during recovery, and thus it must be able to handle repeated system crashes. Fortunately, FastCheck simply restarts the recovery process in the event of a crash during recovery because it does not log any update in the metadata in either recovery phase. That is, even if our two-phase

recovery algorithm in FastCheck is repeatedly executed, the effect is the same as if executed once. In this sense, it is an idempotent recovery algorithm.

5.4. Applicability for Other Hybrid Mapping FTLs

In this study, we focused on the recovery algorithm for FAST, but we believe that the main idea of FastCheck is easily applicable to other hybrid mapping FTLs, particularly taking into account the fact that FastCheck can successfully recover any crashes in the middle of very complicated internal operations in FAST, such as multiple merges for a victim log block.

For any hybrid mapping FTL to be recoverable, it is essential to guarantee that both the block-mapping table and the log-page-mapping table are reliably stored. In this respect, we believe that FastCheck provides a model case of recoverable hybrid mapping FTLs. FastCheck guarantees the recoverability of both mapping tables against any kind of crashes. First, FastCheck ensures the consistency of the block-mapping table by logging any block-mapping change at the map blocks for every merge operation. Second, it can also ensure the consistency of the log-page-mapping table with its two-phase algorithm, which maintains additional timestamp metadata for detecting the sequence of written data in log blocks.

6. Performance Evaluation

In this section, in terms of the overhead for logging during a normal execution period and the time cost during recovery, we qualitatively evaluate the two recovery policies of the LTFTL and FAST FTLs (as mentioned in Section 2.1, the PORCE scheme is excluded in the evaluation since it has some limitations when applied to the FTLs where complex multiple merge operations are internally executed, such as FAST). We first represent an analytical model for the qualitative comparison between the two recovery schemes and then show the trace-driven simulation results.

6.1. Qualitative Analysis

First of all, as shown in Table 1, we define the notation, and based on this, we present a cost model to compare the efficiency between LTFTL's recovery scheme and the FastCheck of FAST.

Table 1. Definition of the notation of the analytical model.

Notation	Description
C_{flash}	Capacity of the flash memory
L_{log_rate}	Percentage of the log block area to the flash memory
N_{writes}	Number of write requests from the host
N_{pmap}	Number of pages to store a log-page-mapping table
N_{log_recs}	Number of log records generated by data page writes in LTFTL (Note1)
$N_{lrecs_in_page}$	Number of log records that a single page can accommodate in LTFTL
$N_{lr_threshold}$	Threshold value of the number of log records allowable in LTFTL (Note2)
P_{ltftl}	Checkpoint period in pages for the log records in LTFTL (Note3)
$P_{fastcheck}$	Checkpoint period in pages for the log-page-mapping table in FastCheck (Note4)

(Note1) In case of the LTFTL's recovery scheme adapted in FAST, two log records would be generated per data page write: (1) when a data page is written to the log block area and (2) when a data page is merged out from the log block area to the data block area. Thus, the total number of log records to be generated would be $2 \times N_{log_recs}$. (Note2) In LTFTL, if the number of log records that have been generated so far in the RAM exceeds $N_{lr_threshold}$, the whole log-page-mapping table should be persistently written to flash memory at once. (Note3) The interval between checkpoints is determined by the page size of the flash memory. Thus, P_{ltftl} equals $N_{lrecs_in_page}$. (Note4) $P_{fastcheck}$ equals the number of pages in the log blocks that a log region can manage, that is, the number of LPN values in the log-page-mapping table that a log region can retain. Thus, $P_{fastcheck}$ can be calculated as x/y, where x is the size of a page in the flash memory and y is the size of the space necessary to store an LPN value. For reference, a log region is the same size as a page of flash memory.

6.1.1. Logging Overhead during Normal Read and Write Operations

Assuming that the LTFTL's recovery algorithm is applied to FAST instead of FastCheck such that many pages with the amount of $N_{lrecs_in_page}$ log records each would be generated in RAM and later, those would be checkpointed on flash memory every Pltftl. If the total number of log records generated so far exceeds $N_{lr_threshold}$, then N_{pmap} pages, capable of storing a whole log-page-mapping table should be written to flash memory at once. Therefore, the overhead cost for normal read and write operations in the LTFTL's recovery policy, $Cost_{ltftl}$, would be as below:

$$\begin{aligned} Cost_{ltftl} &= \frac{N_{\log_recs}}{N_{lrecs_in_page}} + [\frac{N_{\log_recs}}{N_{lr_threshold}}] \times N_{pmap} \\ &= \frac{2\times N_{writes}}{P_{ltftl}} + [\frac{2\times N_{writes}}{P_{ltftl}\times N_{pmap}}] \times N_{pmap} \\ &\cong \frac{4\times N_{writes}}{P_{ltftl}} . \end{aligned} \tag{2}$$

Meanwhile, in the case of FastCheck of FAST, the checkpoint operations occur every $P_{fastcheck}$ in process of normal read and write operations. Thus, $Cost_{fastcheck}$ for the total number of write requests from the host, N_{writes}, would be as below:

$$Cost_{fastcheck} = \frac{N_{writes}}{P_{fastcheck}} . \tag{3}$$

Under a reasonable assumption that the size of a page is 4 KB, the size of a block is 512 KB, and the chip size of the flash memory is 8 GB [20], according to Table 1, $P_{fastcheck}$ would be 1024 (if we assume that when storing LPN values in the log-page-mapping table, the space of 4 B is needed for each value, the number of LPN values that a log region in the log-page-mapping table can hold is 4 KB/4 B = 1024.). On the other hand, each log record is of the size 16 B in LTFTL; therefore, P_{ltftl} would be 256. Therefore, because $P_{fastcheck} = 4 \times P_{ltftl}$, the overhead costs during a normal execution period for FastCheck and LTFTL's recovery scheme can be compared using the following equation:

$$Cost_{fastcheck} \cong \frac{1}{16} \times Cost_{ltftl}. \tag{4}$$

Consequently, we can realize that the write overhead due to the logging during a normal execution period in FAST is almost 16 times less than in LTFTL.

6.1.2. Recovery Time Cost

In the following, we describe how long it takes for each recovery scheme to restore its page-mapping table at the recovery time. The LTFTL's recovery approach has to read a large number of pages storing the page-mapping table, log pages containing log records, and a large number of log pages in the log block area for the uncheckpointed log records, which reside in the main memory when a system failure occurred. On the other hand, as mentioned in Section 5, the FastCheck scheme must read the number of pages storing the page-mapping table from the checkpoint blocks and a large number of log pages in the log block area for the uncheckpointed log region that is not yet checkpointed from the main memory at the occurrence of the system failure. From these features, we count the maximum number of pages read for the restoration of the log-page-mapping table:

$$\begin{aligned} Cost_{ltftl} &= N_{pmap} + \frac{N_{lr_{threshold}}}{N_{lrecs_{inpage}}} + N_{lrecs_{inpage}} \\ &= N_{pmap} + \frac{P_{ltftl}\times N_{pmap}}{P_{ltftl}} + P_{ltftl} \\ &= 2 \times N_{pmap} + P_{ltftl} , \end{aligned} \tag{5}$$

$$Cost_{fastcheck} = N_{pmap} + P_{fastcheck} . \tag{6}$$

If we make the same assumption as with Equation (4), we can also get $P_{fastcheck} = 4 \times P_{ltftl}$ and the following equation by using the above two Equations (5) and (6):

$$\frac{Cost_{fastcheck}}{Cost_{ltftl}} = \frac{C_{flash} \times L_{\log_rate} + 4}{2 \times C_{flash} \times L_{\log_\mathrm{rate}} + 1} \,. \tag{7}$$

In Equation (7), depending on the capacity of the flash memory and the percentage of the log block area, the FastCheck scheme can have a shorter or longer recovery time compared with the LTFTL's recovery scheme. Suppose that the capacity of the flash memory is 16 GB and the percentage of the log block area is 20%. The two recovery schemes show almost equivalent recovery times. Under a situation where the flash memory has a capacity of less than 16 GB (namely 8 GB), the FastCheck scheme needs a longer recovery time than the LTFTL's recovery scheme. However, under the reverse situation, the predicted result would be in favor of the FastCheck scheme. Meanwhile, if we lower the percentage of the log block area, namely, 10%, the FastCheck scheme would show a favorable result on the condition that the capacity of the flash memory is very large, for instance, 64 GB. In summary, under an environment where the flash memory size is relatively small and the ratio of log blocks to flash memory is also a little low, in terms of recovery time, the LTFTL's recovery scheme has an advantage over FastCheck, which is, however, better in the reverse environment.

The following Sections 6.2 and 6.3 provide a comparison between the simulation results of FastCheck's and LTFTL's recovery methods.

6.2. Simulation Environment

In order to evaluate and compare the performances between the two recovery schemes, we modified a trace-driven FAST simulator [5] for each. We assumed that the NAND flash memory had a storage capacity of 8 GB, a page was 4 KB in size, and a block was 512 KB in size [26]. This is because the range of logical page addresses for the workloads obtained from the two benchmarks described below was within 8 GB. We also assumed that the simulator had enough main memory to store all address-mapping information maintained by each recovery scheme. The simulator counted the numbers of page reads/writes and block erase operations for the given workloads.

To generate simulation workloads, we used two sorts of benchmark programs. First, to obtain an online transaction processing (OLTP) trace, we used a commercial TPC-C benchmark tool [27] and made it perform the benchmark workload on a commercial database server. The TPC-C benchmark is a mixture of five types of read-only transactions and update-intensive transactions that simulate the activities that are commonly found in real-world OLTP applications [22]. During the benchmark execution time, we traced all the write requests submitted to a raw storage device. As a result of analyzing the traced data, we got 8,707,364 write requests, most of which were relatively small random writes, that is, 4 KB.

Second, we also ran a Postmark benchmark tool [28] on the Linux ext2 file system and captured all the write requests at the raw level of a flash storage device. The Postmark benchmark is used to measure the performance of applications such as email, netnews, and e-commerce [29]. Unlike an OLTP trace, the write pattern of the Postmark benchmark was in the form of a mixture of sequential writes and random writes, where the degree of randomness for the writes was not severe. Furthermore, the number of write requests was 1,116,874 and we observed that the data size of the writes ranged from 512 B to 64 KB.

In implementing the LTFTL's recovery algorithm on FAST, we made a slight modification to this algorithm. Since the LTFTL does not easily support the consistency of merge operations that the FTLs like FAST have to perform, we added a log-flush function module that plays the role of flushing out log records in the RAM to flash memory immediately after a merge operation. Furthermore, we applied the same rule to the LTFTL's recovery scheme with respect to the logging and recovery for the block-address-mapping table, which does not exist in LTFTL.

6.3. Simulation Results

Figure 7 shows a comparison between the write overheads for logging using FastCheck and the LTFTL's recovery methods during a normal execution period. We measured them by varying the rate of the log block area to flash memory from 3 to 20%. As shown in Figure 7a,b, in both cases, FastCheck clearly incurred a lower write overhead cost than the LTFTL's recovery method. This is because FastCheck carries out the checkpoints for the log-page-mapping data using a relatively long checkpoint period, as shown in Equation (3). On the other hand, LTFTL needs to flush out log records in the RAM more frequently to ensure the correctness of the merge operations. In particular, with respect to the overhead due to the logging for the page-address-mapping information, LTFTL should perform more additional writes for the logging than FastCheck by 60 to 120 times.

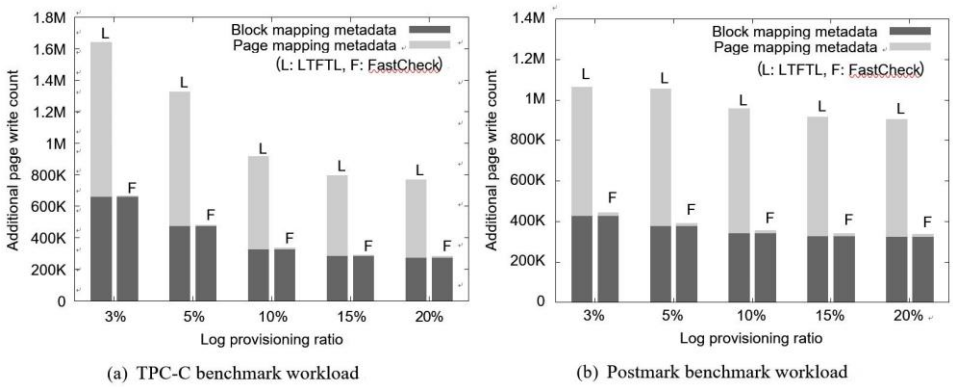
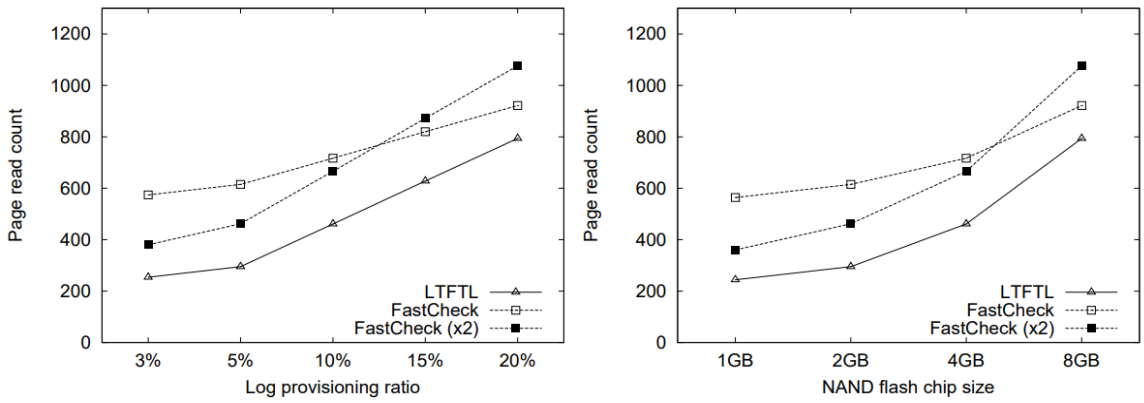


Figure 7. Overhead cost during a normal execution period.

Interestingly, we can observe in Figure 7 that aspects of the two write overheads due to the logging for the page-address-mapping table differ from each other. FastCheck shows very uniform overheads in both of the subfigures, while in LTFTL, the overheads are significantly affected by the provisioning rate and the workload pattern. With O-FAST's advantage [5], as the provisioning log space is enlarged, a lot of merges could be avoided because many log pages in the log blocks would be invalidated by the enlarged log window. For this reason, the more merges that are skipped, fewer metadata (like mapping information) modifications occurred. As a result, the additional write overhead could be reduced. Another reason for this observation is that the write pattern of the TPC-C benchmark workload was quite skewed in some data pages, that is, it had a high temporal locality. As shown in Figure 7a, the write overhead resulting from logging the page-mapping metadata decreased in LTFTL because it can get more chances to save many log-flush operations. However, in the Postmark benchmark workload, sequential writes are dominant, thus LTFTL does not obtain much of a benefit from FAST's characteristics. Of course, Figure 7b shows that the burden of additional writes for the logging in LTFTL is quite high regardless of the provisioning log space. On the other hand, FastCheck seems to be scalable in terms of the provisioning size. In FastCheck, only write requests from the host affects the write overhead during a normal execution period. Therefore, FastCheck shows the uniform overhead cost irrespective of the flash memory configuration and even the workload pattern.

In addition, we evaluated the performance of the two recovery schemes in terms of the recovery time, which was definitely related to how many pages were read during the recovery. Figure 8 shows the performance results after varying the provisioning log area rate and the flash chip size. As shown in Figure 8a,b, FastCheck required more recovery time than LTFTL's recovery scheme on average. In fact, if we assumed that the latency of a single page read was 30 µs, the recovery time gap between the two recovery schemes was only about 5 ms. Furthermore, we can see in the figure that as the provisioning rate and the flash memory capacity increased, the recovery time cost of FastCheck came close to one of LTFTL's recovery scheme. In particular, supposing that the flash memory capacity is very

large, e.g., 32 GB or 64 GB, FastCheck would show a recovery time cost that was similar to or less than LTFTL's recovery scheme, which we expected already in Section 6.1.2.



(a) Varying the provisioning size

(b) Varying the flash memory capacity (provisioning rate: 20%)

Figure 8. Recovery time cost.

According to the above observation, FastCheck required much more recovery time than LTFTL's recovery scheme under the configuration that the volume of a flash memory chip was not large enough and the provisioning log space size was small. This was because in Equation (6), the ratio of the time cost to read all the log pages that a log region manages, that is, $P_{fastcheck}$ to the overall time cost was greater under such a configuration compared with the LTFTL's recovery scheme. In order to reduce this cost during the recovery, we considered a faster checkpoint period, $P_{fastcheck}$, by reducing the amount of log pages that the log region managed. In Figure 8, we can see that when making the checkpoint period of FastCheck twice as fast, FastCheck revealed a small recovery time cost under the configuration above.

7. Conclusions

In this paper, we proposed a novel crash recovery scheme called FastCheck that focuses on the FAST FTL, which is one of the well-known hybrid mapping FTLs. FastCheck periodically writes newly generated FTL metadata, such as address-mapping information with a checkpoint approach, which exploits the characteristic of the FAST FTL where the log blocks in a log area are used in a round-robin fashion. We explained that FastCheck can guarantee the consistency of the modified FTL metadata against sudden system failure. Furthermore, we qualitatively and quantitatively showed that FastCheck provided a small logging overhead during a normal execution period and a good recovery time during the recovery in an environment where the log provisioning rate was relatively high, e.g., over 20%, and the flash memory capacity was very large, e.g., 32 GB or 64 GB, compared with the LTFTL's recovery scheme. On the other hand, for flash memory with a log ratio of less than 20% or a capacity of less than 32 GB, LTFLT's recovery strategy was advantageous. For future work, we plan to implement a FastCheck prototype in a real environment and evaluate it to verify its efficiency. In addition, we will propose an optimized logging algorithm, even for the block-mapping table that every hybrid mapping FTL maintains.

Author Contributions: Conceptualization, J.-H.P. and S.-W.L.; validation, D.-J.P. and T.-S.C.; writing—original draft preparation, J.-H.P. and D.-J.P.; writing—review and editing, D.-J.P. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the Institute of Information & Communications Technology Planning & Evaluation (IITP)(no. 2015-0-00314); in part by the MSIT (Ministry of Science, ICT), Korea, under the High-Potential Individuals Global Training Program)(2020-0-01592), supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation); in part

by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2019R1F1A1058548).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Gupta, A.; Kim, Y.; Urgaonkar, B. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In Proceedings of the 14th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09), Washington, DC, USA, 7–11 March 2009; pp. 229–240.
2. Kang, J.-U.; Jo, H.; Kim, J.-S.; Lee, J. A Superblock-Based Flash Translation Layer for NAND Flash Memory. In Proceedings of the 6th ACM & IEEE International Conference on Embedded Software—EMSOFT '06, Seoul, Korea, 22–25 October 2006; IEEE Press: New York, NY, USA, 2006; pp. 161–170.
3. Kim, J.; Kim, J.M.; Noh, S.; Min, S.L.; Cho, Y. A space-efficient flash translation layer for CompactFlash systems. *IEEE Trans. Consum. Electron.* **2002**, *48*, 366–375.
4. Lee, S.; Shin, D.; Kim, Y.-J.; Kim, J. LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems. *SIGOPS Oper. Syst. Rev.* **2008**, *42*, 36–42. [CrossRef]
5. Lee, S.-W.; Park, D.-J.; Chung, T.-S.; Lee, D.-H.; Park, S.; Song, H.-J. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embed. Comput. Syst.* **2007**, *6*, 18. [CrossRef]
6. Lim, S.-P.; Lee, S.-W.; Moon, B. FASTer FTL for Enterprise-Class Flash Memory SSDs. In Proceedings of the IEEE SNAPI 2010 6th International Workshop on Storage Network Architecture and Parallel I/Os, New York, NY, USA, 3 May 2010; pp. 3–12.
7. Mativenga, R.; Paik, J.-Y.; Kim, Y.; Lee, J.; Chung, T.-S. RFTL: Improving performance of selective caching-based page-level FTL through replication. *Clust. Comput.* **2018**, *22*, 25–41. [CrossRef]
8. Wu, C.-H. A self-adjusting flash translation layer for resource-limited embedded systems. *ACM Trans. Embed. Comput. Syst.* **2010**, *9*, 1–26. [CrossRef]
9. Chung, T.-S.; Park, D.-J.; Park, S.; Lee, D.-H.; Lee, S.-W.; Song, H.-J. A survey of Flash Translation Layer. *J. Syst. Arch.* **2009**, *55*, 332–343. [CrossRef]
10. Chung, T.-S.; Lee, M.; Ryu, Y.; Lee, K. PORCE: An efficient power off recovery scheme for flash memory. *J. Syst. Arch.* **2008**, *54*, 935–943. [CrossRef]
11. Sun, K.; Baek, S.; Choi, J.; Lee, D.; Noh, S.; Min, S. LTFTL: Lightweight Time-Shift Flash Translation Layer for Flash Memory Based Embedded Storage. In Proceedings of the 8th ACM International Conference on Embedded Software, Atlanta, GA, USA, 19–24 October 2008; Association for Computing Machinery: New York, NY, USA, 2008; pp. 51–58.
12. Wu, C.-H.; Lin, H.-H. Timing Analysis of System Initialization and Crash Recovery for a Segment-Based Flash Translation Layer. *ACM Trans. Des. Autom. Electron. Syst.* **2012**, *17*, 1–21. [CrossRef]
13. Ban, A. Flash File System. U.S. Patent 5,404,485, 4 April 1994.
14. Chiang, M.-L.; Lee, P.C.H.; Chang, R.-C. Using Data Clustering to Improve Cleaning Performance for Flash Memory. *Softw. Pract. Exp.* **1999**, *29*, 267–290. [CrossRef]
15. Ban, A. Flash File System Optimized for Page-Mode Flash Technologies. U.S. Patent 5,937,425, 10 August 1998.
16. Aleph One Company. YAFFS: Yet another Flash Filing System. Available online: http://www.yaffs.net/ (accessed on 1 December 2020).
17. Transaction Processing Performance Council. TPC-C Benchmark (Revision 5.8.0). 2006. Available online: http://www.tpc.org/tpcc/ (accessed on 1 December 2020).
18. Woodhouse, D. JFFS: The Journaling Flash File System. In Proceedings of the Ottwa Linux Symposium, Ottawa, ON, Canada, 30 July 2001.
19. Wu, C.-H.; Kuo, T.-W.; Chang, L.-P. The Design of efficient initialization and crash recovery for log-based file systems over flash memory. *ACM Trans. Storage* **2006**, *2*, 449–467. [CrossRef]
20. Gal, E.; Toledo, S. A Transactional Flash File System for Microcontrollers. In Proceedings of the USENIX Annual Technical Conference, General Track, Anaheim, CA, USA, 10–15 April 2005; pp. 89–104.
21. Chang, Y.-M.; Lin, P.-H.; Lin, Y.-J.; Kuo, T.-C.; Chang, Y.-H.; Li, Y.-C.; Li, H.-P.; Wang, K.C. An Efficient Sudden-Power-Off-Recovery Design with Guaranteed Booting Time for Solid State Drives. In Proceedings of the 2016 IEEE 8th International Memory Workshop (IMW), Paris, France, 15–18 May 2016; pp. 1–4.
22. Kim, N.; Won, Y.; Cha, J.; Yoon, S.; Choi, J.; Kang, S. Exploiting Compression-Induced Internal Fragmentation for Power-Off Recovery in SSD. *IEEE Trans. Comput.* **2015**, *65*, 1720–1733. [CrossRef]
23. Mativenga, R.; Hamandawana, P.; Chung, T.-S.; Kim, J. FTRM: A Cache-Based Fault Tolerant Recovery Mechanism for Multi-Channel Flash Devices. *Electronics* **2020**, *9*, 1581. [CrossRef]
24. Choi, J.-Y.; Nam, E.H.; Seong, Y.J.; Yoon, J.; Lee, S.; Kim, H.; Park, J.; Woo, Y.-J.; Lee, S.; Min, S.L. HIL: A Framework for Compositional FTL Development and Provably-Correct Crash Recovery. *ACM Trans. Storage* **2018**, *14*, 1–29. [CrossRef]
25. Moon, S.; Lim, S.-P.; Park, D.-J.; Lee, S.-W. Crash Recovery in FAST FTL. In Proceedings of the SEUS'10 8th IFIP WG 10.2 International Conference on Software Technologies for Embedded and Ubiquitous Systems, Waidhofen/Ybbs, Austria, 13–15 October 2010; pp. 13–22.

26. Samsung Electronics. 8Gx8 Bit NAND Flash Memory (K9XDG08U5M). Data Sheet. Available online: http://docs-emea.rs-online.com/webdocs/0def/0900766b80defffe.pdf (accessed on 1 December 2020).
27. TCP-C OLTPBench Code. Available online: https://github.com/oltpbenchmark/oltpbench/ (accessed on 1 December 2020).
28. Postmark Benchmark Code. Available online: https://www.dartmouth.edu/~davidg/postmark-1_5.c/ (accessed on 1 December 2020).
29. Katcher, J. *Postmark: A New File System Benchmark*; Technical Report; Network Appliance Inc.: Sunnyvale, CA, USA, 1997.