Stony Brook University

OFFICIAL COPY

The official electronic file of this thesis or dissertation is maintained by the University Libraries on behalf of The Graduate School at Stony Brook University.

© All Rights Reserved by Author.

LFSM – a system to optimize the random write performance of FLASH memory

A Thesis Presented

by

Goutham Meruva

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Master of Science

in

Computer Science

Stony Brook University

May 2010

Stony Brook University

The Graduate School

Goutham Meruva

We, the thesis committee for the above candidate for the

Master of Science degree, hereby recommend

acceptance of this thesis.

Professor Tzi-cker Chiueh – Thesis Advisor
Department of Computer Science

Associate Professor Erez Zadok – Chairperson of Defense
Department of Computer Science

Assistant Professor Jie Gao
Department of Computer Science

This thesis is accepted by the Graduate School

Lawrence Martin
Dean of the Graduate School

Abstract of the Thesis

LFSM – a system to optimize the random write performance of FLASH memory

by

Goutham Meruva

Master of Science

in

Computer Science

Stony Brook University

2010

In recent years, the storage medium behind enterprise servers and personal computers has started migrating from the traditional magnetic disks to solid state disks (FLASH memory). There are many compelling reasons behind this shift like superior access speeds, high robustness, and lower power consumption due to lack of any mechanical moving parts. However, there are two very important drawbacks of FLASH which are causing severe concerns. First, the cost-per-GB is still very high and second, FLASH's random write performance is many folds slower than its sequential write performance. Log Structured Flash Storage Manager (LFSM) is a storage system for FLASH which effectively alleviates this problem to a great extent. The basic idea is to treat every write as sequential write and it sequentially to FLASH and maintain the mapping information. To solve the three common problems with this kind of design – random write to the

mapping information, lookup of the mapping, and generation of free space for new incoming writes – LFSM incorporates three authentic techniques called Batching Updates and Synchronous Commit (BUSC), Temperature-based Garbage Collection and Interval-based Caching.

In this report we explain the fundamental architecture of FLASH memory, techniques to leverage the best performance out of it, a system to convert all writes to sequential writes with respect to FLASH, BUSC principle, Temperature-based Garbage Collection, Interval-based Cache and finally the results of experiments conducted to evaluate FLASH's performance.

To

My Family and Teachers

Table of Contents

1 Introduction........ 1

2 Overview 3

2.1 General description 3

2.2 LFSM Architecture 3

2.2.1 Main thread........ 6

2.2.2 Background thread 6

3 Design of LFSM 7

3.1 Disk Write Logging........ 7

3.2 BMT 7

3.3 BMT Update Log 10

4 Interval-based Cache for in-memory BMT 12

4.1 Data structure........ 12

4.2 BMT cache LRU list: 16

5 BMT Commit........ 18

5.1 BUSC Principle........ 18

5.2 BMT Commit Manager 18

5.3 Crash Recovery........ 20

5.4 Signature Sector 21

6 Synchronization 22

6.1 Active List 22

7 Garbage Collection........ 23

7.1 Utilization 24

7.2 Temperature........ 25

7.3 Procedure........ 26

8 Distribution of BMT log and BMT Update log EUs (Floating) 28

8.1 Super map........ 28

8.2 Dedicated map 29

8.3 BMT Update log 29

8.4 BMT log 29

8.5 BMT Lookup 30
8.6 BMT Commit 30
8.7 Crash Recovery 30
9 Performance Evaluation 31
9.1 Methodology 31
9.2 Programs 31
9.3 Environment 31
9.4 Results 32
10 Conclusion and Future Work 34
Bibliography 35

Acknowledgements

First of all, I wish to sincerely thank Prof. Tzi-cker Chiueh for his guidance and support all through the project. I would also like to thank all the present and past colleagues at Rether Networks Inc (RNI). In particular, Mr. Pi-Yuan Cheng for his insightful advice and help throughout my thesis work and Sheng-I Doong, President of Rether Networks, for her kind support.

Chapter 1

1 Introduction

The recent commoditization of USB-based flash disks, mainly used in digital cameras, mobile music/video players and cell phones, has many pundits and technologists predict that flash memory-based disks will become the mass storage of choice on mainstream laptop computers in two to three years. In fact, some of the ultra mobile PCs, such as AsusTeks Eee PC, already use flash disks as the only mass storage device. Given the much better performance characteristics and enormous economies of scale behind the flash disk technology, it appears inevitable that flash disks will replace magnetic disks as the main persistent storage technology, at least in some classes of computers.

Compared with magnetic disks, flash disks consume less power, take less space, and are more reliable because they don't include any mechanical parts. Moreover, flash disks offer much better latency and throughput in general because they work just like a RAM chip and don't incur any head positioning overhead. However, existing flash disk technology has two major drawbacks that render it largely a niche technology at this point. First, flash disk technology is still quite expensive, approximately $10-15/GB, which is at least 20 times as expensive as magnetic disks. Indeed, at this price point, it is not uncommon that a flash disk costs as much as the computer it is installed on. Second, flash disks performance is better than magnetic disk when the input workload consists of sequential reads, random reads, or sequential writes. Under a random write workload, flash disks performance is comparable to that of magnetic disk at best, and in some cases actually worse. We believe the cost issue will diminish over time as the PC industry shifts its storage technology investment from magnetic to flash disks. However, flash disks random write performance problem is rooted in the way flash memory cells are modified, and thus cannot be easily addressed. This document describes the design and implementation of a log-structured flash storage manager (LFSM) that effectively solves this problem.

A flash memory chip is typically organized into a set of erasure units (EUs) (typically 256 Kbytes), each of which is the basic unit of erasure and in turn consists of a set of 512-byte sectors, which correspond to the basic units of read and write. After an EU is erased, writes to any of its sectors can proceed without triggering an erasure if their target addresses are disjoint. That

is, after a sector is written and before it can be written the second time, it must be erased first. Because of this peculiar property of flash memory, random writes to a storage area mapped to an EU may trigger repeated copying of the storage area to a free EU and erasing of the original EU holding the storage area, and thus result in significant performance overhead.

Moreover, flash disks typically come with a flash translation layer (FTL), which is implemented in firmware, maps logical disk sectors, which are exposed to the software, to physical disk sectors, and performs various optimizations such as wear leveling, which equalizes the physical write frequency of EUs. This logical-to-physical map will require 64 million entries if it keeps track of individual 512-byte sectors on a 32-GB flash disk. To reduce this maps memory requirement, commodity flash disks increase the mapping granularity, sometimes to the level of an EU. As a result of this coarser mapping granularity, two temporally separate writes to the same mapping unit, say an EU, will trigger a copy and erasure operation if the target address of the second write is not larger than that of the first write, because a commodity flash disk cannot always tell whether a disk sector in an EU has already been written previously. That is, if the N-th sector of a mapping unit is written, any attempt to write to any sector whose sector number is less than or equal to N will require an erasure, even if the target sector itself has not been written at all. Consequently, coarser mapping granularity further aggravates flash disks random write performance problem.

To address the random write performance problem, LFSM converts all random writes into sequential writes to a set of unified logs by introducing an additional level of indirection above the FTL. Because all commercial flash disks have good sequential write performance, LFSM effectively solves the random write performance problem for these disks in a uniform way without requiring any modifications to their hardware implementations. With this log-structured storage organization, LFSM needs to overcome two major challenges. First, LFSM still face random writes because it needs to maintain a separate map for the level of indirection or translation it introduces and writes to this map are random. LFSM minimizes the performance overhead of these random writes by using a technique called BUSC, batching updates with sequential commit. Second, to minimize the amount of copying whenever LFSM reclaims an EU, it needs to allocate EUs to logical blocks in such a way that logical blocks assigned to the same EU have a similar life time and each EU contains the stabilized utilization ratio, which means it is less possible the utilization ratio will be changed in the future.

Chapter 2

2 Overview

2.1 General description

LFSM is a storage manager that sits between a file system and a flash disks native driver, and can be considered as an auxiliary driver specifically designed to optimize the random write performance for existing flash disks in a disk-independent way. A property shared by all commodity flash disks on the market is good sustained throughput for sequential writes, between 30-60 MB/sec. The basic idea behind LFSM is to convert random writes into sequential writes so as to eliminate random writes from the workload that a flash disk physically faces by construction. To perform such conversion, LFSM implements the linear disk address space exposed to the file system using multiple logs, and turns every incoming logical write into a physical write to the end of one of these logs which mapped to different active EUs. Because writes to each log are sequential in an EU based nature, their performance is the same as sequential write performance.

2.2 LFSM Architecture

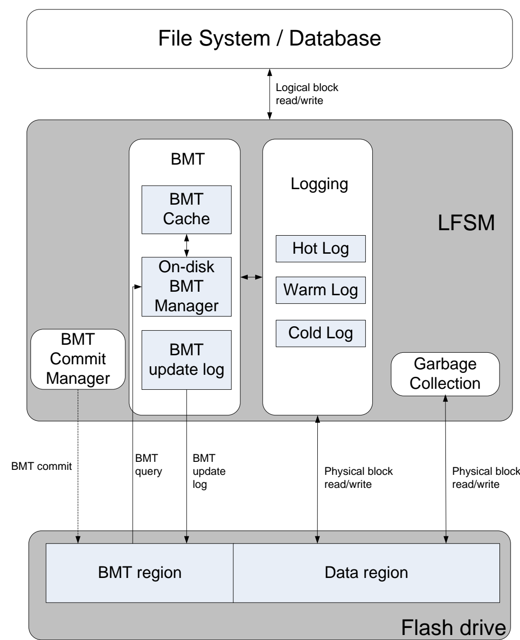


Figure 1.1 LFSM architecture

As Fig 1.1 demonstrates, LFSM sits below the file system and above the flash drive. Accordingly, there are two address spaces in this design. The file system and/or user applications see a linear sequence of logical blocks exposed by LFSM. The native flash disk driver exposes a linear sequence of physical blocks to LFSM.

LFSM consists of two threads. The main thread is responsible for synchronous flash disk logging whereas the other background thread is responsible for asynchronous BMT commit and garbage collection.

The main function of logging block is to transfer the random write to the sequential write. While receiving the random logical write request, the logging block converts it to the sequential write address in one of the three different temperature logs, hot, cold and warm logs. The three different temperature logs store different life time data. Colleting the data with the similar life time together will ease the garbage collection performance penalty.

Log-structured file system (LSF) was one of the earliest works on organizing the entire file system as a log in order to mitigate the disk I/O bottleneck problem. LFSM borrows ideas from LFS, particularly in the area of garbage collection. LFS maintains a single log of segments, and uses a product of segment age and segment utilization ratio as the metric to determine the order in which segments are reclaimed. In contrast, LFSM advocates multiple logs, each of which is mapped to an EU with a distinct estimated life time range. Then, LFSM maintains a fixed-sized LRU, Hlist (also referred as hot_list), to move the least recently used log EU to the least valid page heap (LVP_heap). LVP_heap sorts the non-recently used EUs by the utilization ratio, and the root of LVP_heap has the minimum utilization ratio. LFSM chooses to reclaim the EU with the additional stabilization information instead of reclaiming EUs only according to their utilization ratio (e.g. smaller first), as in the case of LFS.

As LFSM transfers the logical block address (LBA) to physical block address (PBA), in block mapping table (BMT) it stores the look up table for LBA to PBA. The BMT data is stored on disk, and the on-disk BMT manager gets the BMT record by additional disk I/O. To mitigate the performance penalty of disk I/O, the BMT cache use interval-based data structure to cache the most recently used BMT record in the memory. The BMT update log is a circular log to record the pending BMT entries, the BMT entries which are not committed to the disk yet. LFSM uses BMT update log to retrieve the pending BMT entries from system crash.

BMT commit manager brings the pending BMT records to the on-disk BMT. To ensure BMT commit is sequential process, LFSM brings in an EU worth of BMT entries, commit pending updates to it, and write the new BMT entries to back to the EU brought in. Thus, the BMT commit also invalidates the corresponding pending records in BMT update log. The order in which BMT EUs are brought in is determined based on the following two considerations: (a) effective number of BMT updates committed and thus queue space freed, and (b) the extent to which the global frontier is moved and thus the extent to which the BMT update log is freed. Sometimes one has to focus exclusively on (b) to free up enough space in the BMT update log to continue. The BMT popularity commit aims to handle (a) while the BMT critical commit focus on (b) condition.

To reclaim unused space on the logs, LFSM performs garbage collection in the background, whose associated performance impact could be quite substantial. The performance cost of reclaiming an EU mainly comes from copying out the live physical blocks in it and is thus proportional to the number of such blocks at the time of reclamation. To minimize the performance overhead associated with garbage collection, the LFSM garbage collection implements the intuition that picks for garbage collection the least utilized EU whose temperature is cold.

Following we give an example of a write request. While receiving the read/write request associated to logical block address (LBA), LFSM performs the block map table (BMT) query to identify the temperature of the data. To accelerate the BMT look-up procedure, LFSM uses an in-memory BMT cache. According to the temperature, LFSM logs the content to the physical block address (PBA), and update the PBA to the BMT. To prevent BMT corruption due to the crash, LFSM logs the pending BMT record to the BMT update log. Finally LFSM returns the write success hardware interrupt.

The physical layout of LFSM on disk is a little different from what is depicted in the Fig. 1. That is, although conceptually we maintain the impression of linear BMT and BMT Update log but actually the erasure units of both the logs are distributed on the disk. We do this for purposes of “wear-leveling” and “erasure” which will be explained in detail later. Hence without loss of generality, for all our following discussions, we consider BMT and BMT Update logs as contiguous.

2.2.1 Main thread

The main thread comprises of the reads or writes from the file system. Let's see in detail how to handle each of them in detail.

For WRITE, four actions are supposed to be performed: BMT lookup to get the current physical block number of the request, log the write buffer to the data log sequentially, log the BMT update to the BMT update log sequentially and modify the in-memory data structure for the BMT update.

For READ, two actions are supposed to be performed: BMT lookup to get the current physical block number of the request and read from the disk from the desired location.

2.2.2 Background thread

The background thread is responsible for asynchronous BMT commit and garbage collection. The garbage collection is handled by gc_collect_valid_blocks function while the BMT commit is handled by BMT_commit_manager function.

Chapter 3

3 Design of LFSM

3.1 Disk Write Logging

The logging part of LFSM converts the random logical LBA to sequential PBA belonging to one of the active logs depending on the temperature of the LBA.

The temperature logging idea eases the garbage collection overhead. LFSM categorizes write data into AGE_GROUP_NUM+1 different temperature levels, e.g. AGE_GROUP_NUM is equal to 2, which are hot, warm and cold data. The cold data is expected to have the longest life time, warm data has medium, and hot data has the shortest life time respectively. If the LBA of the data has never been written before, the data will be treated as the cold data as default. If the LBA of the data has been written before, the temperature of the data is increasing, meaning the temperature is hot if it was warm, and is warm if it was cold, and remain hot if it was hot. The temperature level of the data only drops one level when it is copied during the garbage collection as a live data. The hot log should be invalidated in a short period of time and becoming a good target for garbage collection. LFSM keeps AGE_GROUP_NUM+1 active EUs, HListGC.active_eus [], to store the incoming data with different temperature.

All EUs are categorized into three different groups in LFSM, which are free, recently used, and the one with fixed utilization ratio. LFSM links all free EUs in a linked list, HListGC.free_list. Active EUs are picked from the free list. While one active EU is full, it is moved to the HListGC.hot_list, which stores the recently used EUs, and might have different utilization ratio in the future. If the hot_list is full, the least recently used EU in the hot_list will be moved to the HListGC.LVP_Heap in which the EUs are considered having the fixed utilization ratio.

3.2 BMT

The main function of the BMT module is the LBA to PBA mapping. The BMT module can be

divided into three sub systems, the ***on-disk BMT***, ***BMT cache*** and ***BMT update log*** as shown in Fig 3.1.

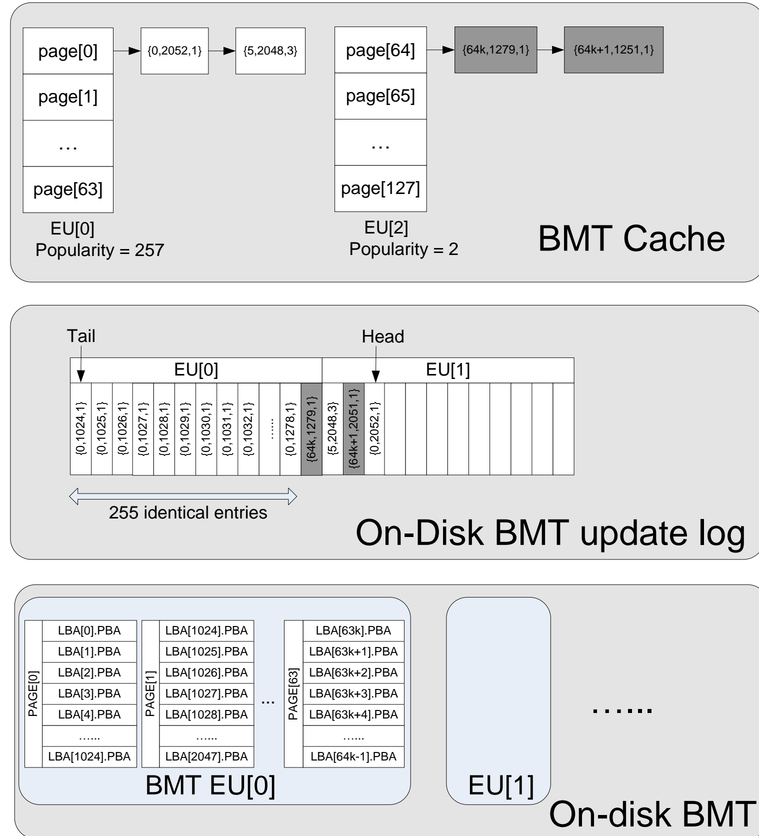


Figure 3.1 BMT models. Assuming EU size is 128KB, page size is 4KB, sector size is 512B, and on-disk BMT entries is 4B indexed by LBA.

LFSM manages the on-disk BMT as an array indexed by LBA and stored on disk. The Fig 3.1 gives an example that one BMT EU can store up to 64K BMT records. On-disk BMT look-up can be simply served by a disk read I/O with LBA offset.

The ***BMT lookup*** process is defined in the bmt_lookup function, which query the BMT in the BMT cache by PPQ_BMT_cache_lookup, and if cache happened, read_small_disk_io_temp is called to obtain the BMT entry from the on-disk BMT and PPQ_BMT_cache_insert_nonpending is called to insert the BMT entry into the cache.

The ***BMT cache*** mitigates the disk I/O time for the BMT look-up procedure. BMT cache is arranged in a per-BMT-page structure as shown in Fig 2. The data in the BMT cache can be categorized as ***pending BMT entries*** and ***non-pending BMT entries***. The pending BMT entries

represent the BMT entries which haven't been committed to the on-disk BMT, while the non-pending BMT entries are identical to the one in on-disk BMT. While logging block writes the new pending BMT record to the BMT update log, it also updates the BMT cache for the same entry. During the BMT lookup process, if cache miss happened, the on-disk BMT manager performs the BMT query through the disk I/O with the minimum write unit size, sector. All of the BMT records in the sector will be added to the BMT cache. While BMT cache is full, LFSM ejects non-pending BMT entries in the least recently used BMT EU. Although the interval-based BMT cache saves memory space by aggregating the adjacent BMT entries, it also introduces additional complexity of merge and split the BMT entries. While inserting a BMT entry to the BMT cache, we have to merge the BMT entry with the adjacent entries if they have contiguous PBAs. While ejecting or updating the BMT entry, we might have to split one BMT entry apart for the different intervals. The BMT cache is defined in bmt_ppq.c and will be described in the latter section.

Although LFSM has converted the random LBA write to the consecutive PBA write, the ***BMT commit manager*** has to write to the LBA BMT entries randomly. LFSM solves this problem by using BUSC scheme to synchronously log the BMT update and asynchronously commits multiple updates to the BMT in a batched fashion. Because of ***BMT update log***, even if the system crashes, the BMT updates that have not been flushed to the on-disk BMT can be correctly reconstructed at recovery time. The ***BMT commit manager*** asynchronously commits the BMT pending records through aggregated and sequential writes to reduce the performance overhead of the random writes to the BMT.

Using BUSC to update the BMT means each logical block write operation triggers three write operations, the first being writing a new version of the logical block to a block data log, the second being logging the associated BMT update log, and the third being actually updating the corresponding on-disk BMT entry. The first two writes are done synchronously and the third write is done in an asynchronously and batched fashion. Later section will present the detail of the ***BMT commit manager***.

BMT update log manager ensures that uncommitted BMT updates can be correctly recovered when machines crash, and thus makes it possible to commit pending BMT updates in an efficient manner without compromising the BMTs integrity.

3.3 BMT Update Log

The BMT update log is a circular sequence of EUs, with two pointers tail and head. The logging block writes the pending BMT records to the BMT update log and head pointer is moved to the next free write sector. While the BMT commit manager will invalidate the pending BMT records in the BMT update log. If all of the BMT records in the tail EU are invalidated, the tail pointer is moved to the head of the next adjacent EU. The size of the BMT update log defines the maximum number of pending records in the LFSM system. While the BMT update log is full, which means the head and tail pointer are overlapped, the incoming write will be pending until the BMT commit manager invalidate the tail EU. The BMT update log entry is designed as A_BMT_E structure. The advantage of using interval-based BMT update log entry is because if LFSM receives the write I/O with more than one sector, the additional BMT update log entry can be replaced by simply increasing the run_length field by one.

When a machine crashes, LFSM scan through the BMT update log and reconstruct the pending BMT entries according to the sequence number in the BMT update log entries. To facilitate the identification of not-yet-committed BMT updates, LFSM includes the following information in the BMT update log entry associated with each logical block write operation: (1) the LBA, PBA, and the run length, (2) the writes corresponding sequence number, (3) the commit point: the sequence number of the youngest logical block write operation all BMT updates before which have already been committed to disk. With these information, LFSM reconstructs pending BMT updates by first identifying the latest or youngest BMT log entry (whose sequence number is N1), then obtaining its associated commit point (whose sequence number is N2), and finally reading in all the BMT update log entries between N1 and N2 to insert them into their corresponding per-BMT-page queues.

Logging BMT updates entails a space overhead problem: Because the minimum unit for reading and writing a flash disk is a 512-byte sector, each BMT update log entry costs a 512-byte sector even though in actuality it requires 22 bytes, which shown in the Fig2. This means the space overhead associated with BMT logging is about 12.5% (512 bytes for every 4-KB page), which is too high to be acceptable. LFSM sitting above to the firmware level cannot utilize the out-of-band area of each block. To minimize the performance overhead, LFSM preserve 10M disk space

dedicating for the BMT update log on 64GB disk. The BMT update log disk space stores up to 20K BMT update log entries.

With the above design, LFSM successfully services each logical block write operation with a single synchronous sequential write to the BMT update log and a sequential write to the active EU, and thus greatly improves the random write performance of modern flash disks. However, BMT update log introduces additional disk write penalty. One way to solve the additional write problem is simply separating the BMT update log to different disks to perform two parallel write operations.

Chapter 4

4 Interval-based Cache for in-memory BMT

The BMT cache is used to improve the performance of BMT look up. The BMT cache utilizes the data structure of the per page queue and mixed the cache entries, which are called non-pending entries, with the BMT per page queue entries, which are called pending entries, to form a sorted linked list.

In order to save the memory space, the consecutive BMT entries in per page queue with the consecutive LBN and PBN can be merged together as an aggregated BMT entry (A_BMT_E). For example, BMT{LBN:100 PBN:200}, and BMT{LBN:101 PBN:201} can be merged as A_BMT_E{LBN:100 PBN:200 runlength:2}. As normal inserting algorithm to the sorted linked list, It takes O(n) for a BMT look up in a per page queue and O(n) to insert an entry to the per page queue.

We need to hold a threshold of max. number of the non-pending entries can be co-existing in the BMT cache, which is PPQ_CACHE_T_HIGH. The background thread is responsible for detecting if the total non-pending entry count, BMT.total_non_pending_items, is larger than the threshold, and removing proper number of entries from the BMT cache when necessary. The corresponding control algorithm for the total pending entry count should be handled by the BMT commit module.

The reminder of this section will describe the A_BMT_E data structure, and go through the main algorithms and private and public functions.

4.1 Data structure

```
struct A_BMT_E{
        sector_t lbno;
        sector_t pbno;
        int run_length;
};
```

Insert one BMT entry to the BMT cache:

The purpose of PPQ_BMT_update function is to insert one single A_BMT_E entry to the ppq.bmt_cache. To insert an A_BMT_E entry to BMT cache by traversing from the head of the BMT cache list, we also need to consider about the new entry could be merged to or split by the existing entries, or it's just an independent entry. We need to adjust the non-pending count and pending count for the BMT cache as well.

The possible inserting cases for the new coming BMT entry can be categorized as following.

1.1 new.lbno < existing.lbno

1.1.1 new.lbno+new.run_len <= existing.lbno

New coming entry is added in front of the existing entry.

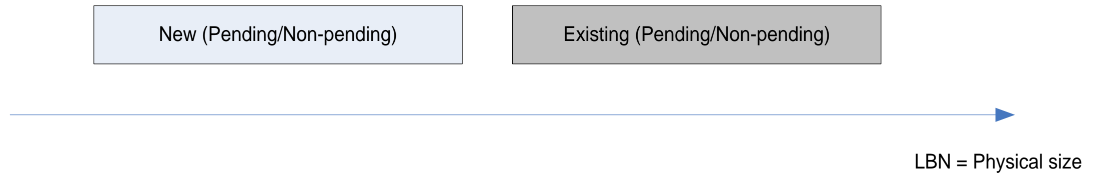


Figure 4.1

1.1.2 new.lbno_new.run_len == existing.lbno+existing.run_len

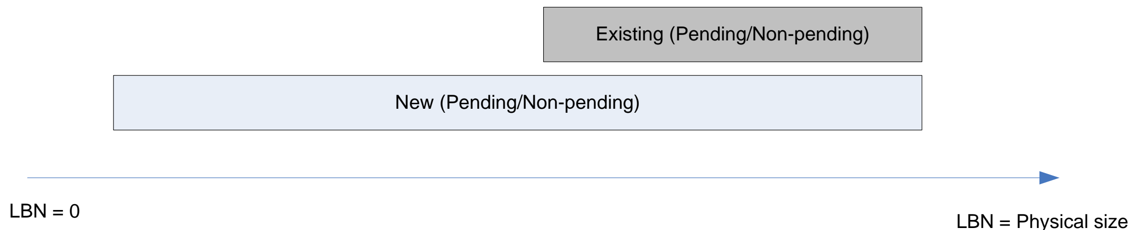


Figure 4.2

a. If the New is a pending entry, it can overwrite the existing entry.

b. If both of the New and the Existing are non-pending entry, the New can just overwrite the Existing.

c. If the New is pending and the Existing is non-pending, the New entry should not overwrite the existing entry. The result will be following.

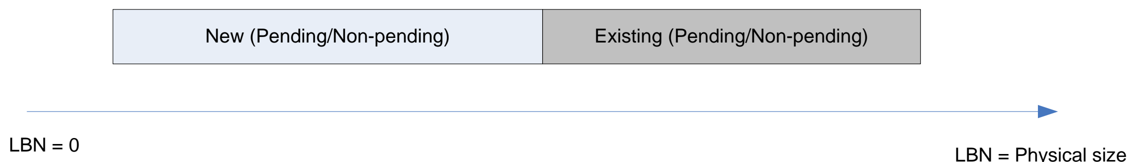


Figure 4.3

1.1.3 new.lbno_new.run_len < existing.lbno+existing.run_len

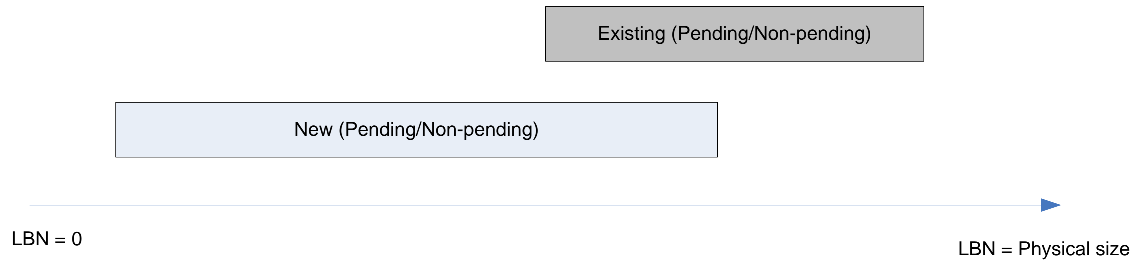


Figure 4.4

We should split the New are two part and decide if lower half of the New can over write Existing by the pending type of them.

1.1.4 new.lbno_new.run_len > existing.lbno+existing.run_len

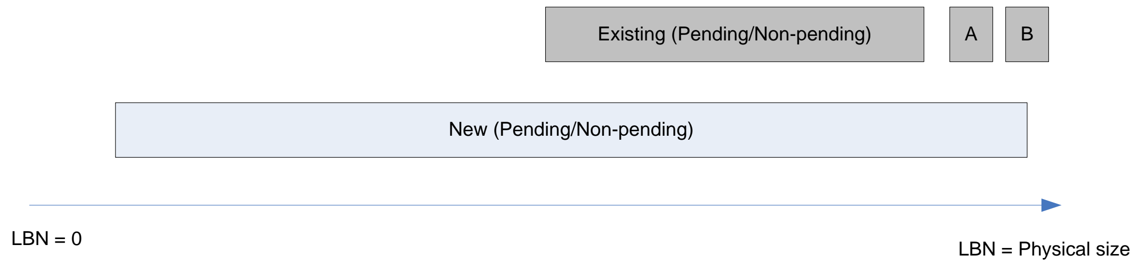


Figure 4.5

1.2 new.lbno >= existing.lbno && new.lbno <= existing.lbno+existing.run_len-1

1.2.1 new.lbno == existing.lbno && new.run_len == existing.run_len

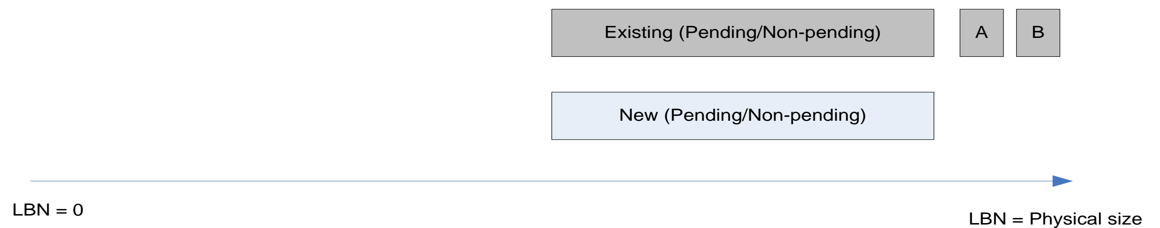


Figure 4.6

1.2.2 new.lbno+new.run_len <= existing.lbno+existing.run_len

1.2.2.1 new.lbno == existing.lbno //head aligned

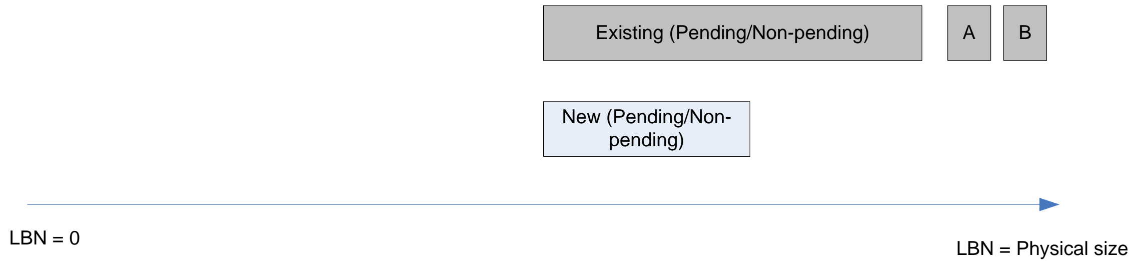


Figure 4.7

1.2.2.2 newnew.lbno+new.run_len == existing.lbno+existing.runlen //tail aligned

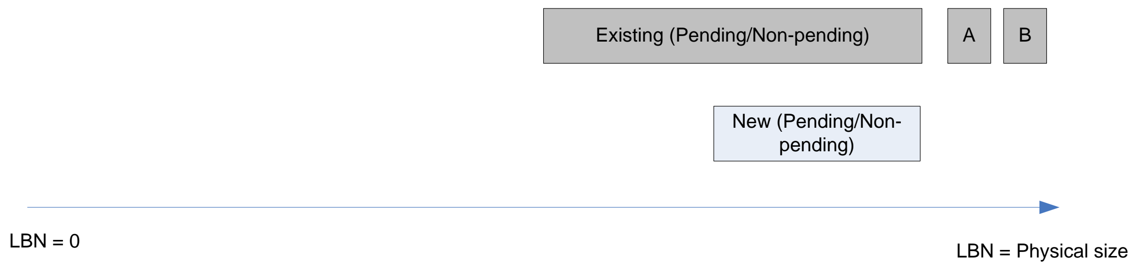


Figure 4.8

1.2.2.3 else //in the middle

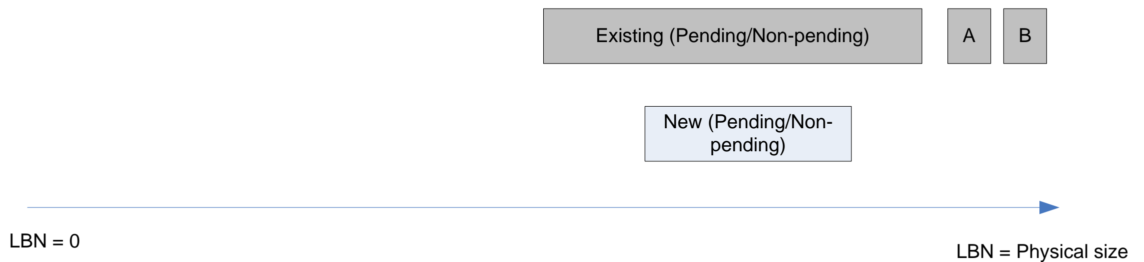


Figure 4.9

1.2.3 new.lbno+new.run_len > existing.lbno+existing.run_len

Two possible cases listed following.

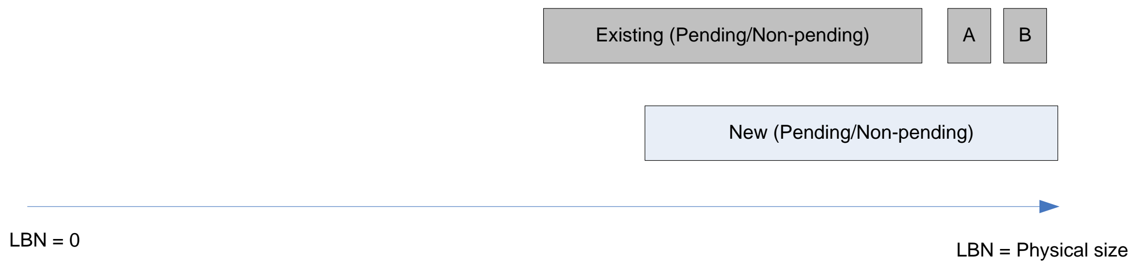


Figure 4.10

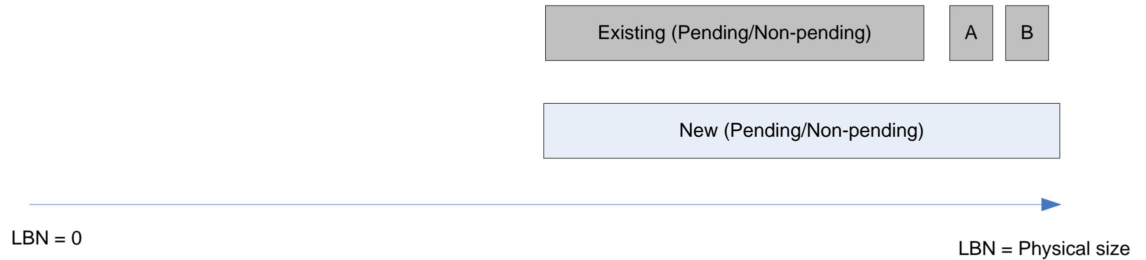


Figure 4.11

1.3 new.lbno > existing.lbno+existing.run_len-1

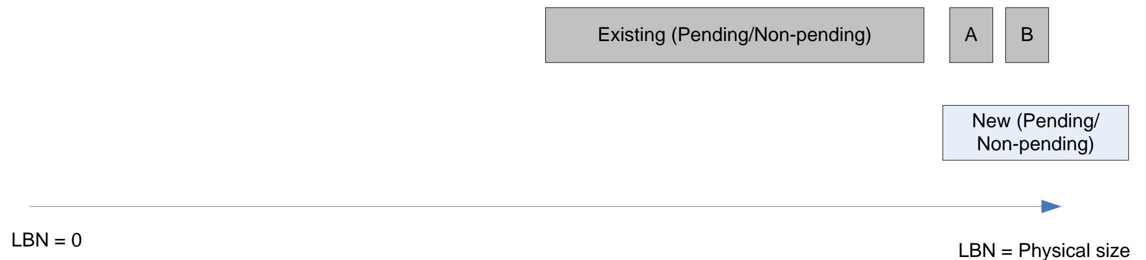


Figure 4.12

To handle the cases that (new.lbno+new.run_len) > (existing.lbno+existing.run_len), we need to consider the merge/split cases for the BMT entries followed by the existing as well.

4.2 BMT cache LRU list:

While BMT cache is full, which means the total number of the non-pending entries in cache reaches the PPQ_CACHE_T_HIGH threshold; we have to remove some non-pending entries from the BMT cache.

To remove the BMT cache entries in the LRU order, we create a LRU list of BMT per page queue cache so we can eliminate the entries according to the LRU list. Only the per page queue with non-pending BMT cache entries will be added into the LRU list. The following example shows a BMT with LRU order as page 1, page 2 and then page n.

We cannot only eliminate the smallest amount of entries to keep the total number of non-pending entries in the BMT cache less than the PPQ_CACHE_T_HIGH, because that will cause the cache full in the nearly future and impact the performance. So, we set up another threshold called PPQ_CACHE_T_LOW. While we hit the cache full condition we remove at least

PPQ_CACHE_T_HIGH- PPQ_CACHE_T_LOW entries to keep the total pending entries less than PPQ_CACHE_T_LOW.

The PPQ_CACHE_T_HIGH and PPQ_CACHE_T_LOW are default set as 10K and 8K. Further research is needed to determine the thresholds to perform the better performance.

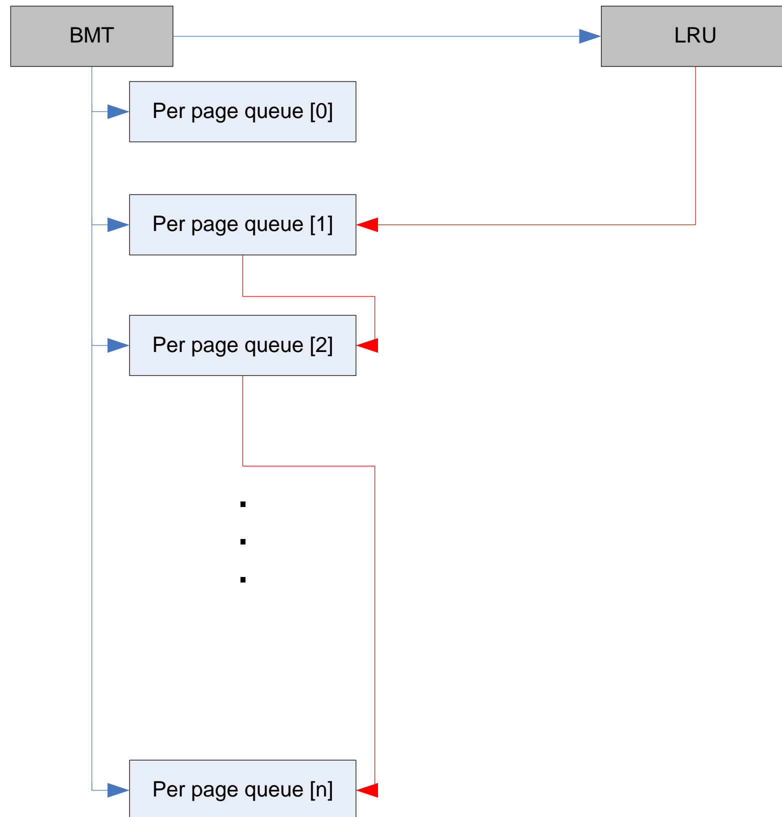


Figure 4.13 BMT Cache LRU list used for eviction

Chapter 5

5 BMT Commit

5.1 BUSC Principle

BUSC stands for Batching Updates and Sequential Commit. FLASH Translation Layer (FTL), which is the firmware for FLASH, is responsible to handle the basic operations like read, write, erasure, wear-leveling, defect management, etc among others. FTL keeps some buffers for incoming writes called “log EUs” which hold temporarily the incoming writes. When a log block is full, the contents inside are copied to their respective erasure units called “data EUs”. If we write sequentially, all the blocks inside a log EU belong to a single EU. In this case, FTL need not do any further copy – it just needs to swap the corresponding data EU with the log EU. So, when we write randomly to FLASH, FTL has no choice but to copy all the blocks from the log EU to their respective data EUs which is the primary reason behind the poor random write performance of FLASH. Keeping this in mind, BUSC principle is designed.

We keep in-memory the updates which are supposed to be written to the on-disk BMT. We keep track of these updates in a way that we know which BMT updates would go to which BMT EU. When we are supposed to write these updates to disk, we read the whole BMT EU from disk, update it with the new content in memory and write the whole EU back sequentially. This is called Batching Updates and Sequential Commit.

5.2 BMT Commit Manager

BMT Commit Manager (BCM) takes care of writing the in-memory pending BMT updates to the on-disk BMT. This process is also called **BMT commit** or simply **commit**. It is performed always in the back ground thread in the function **BMT_commit_manager ().** It is recommended to have a clear understanding of BUSC principle before proceeding further in this section.

Commit and BMT update logging are highly inter-dependent; at any given instance of time, the pending updates which are not yet committed should exist securely in the BMT update log

portion. Even if the system crashes without actually committing all the pending updates to the disk, we use this backup information from the BMT update log to recovery the BMT. For detail explanation of crash recovery, please read the crash recovery section. Since BCM implementation is interleaved with that of BMT update logging, let's see briefly how BMT update logging is done. BMT update log is treated as a circular log where the writing is done from its **head** erasure unit. Once the head reaches the end of the update log, it wraps around. So, intuitively after committing the pending BMT entries in the **tail** of the BMT update log, LFSM moves the **tail pointer** ahead to prevent over-writing.

After every commit, the sectors in the update log holding the committed updates are freed i.e. can be re-used. The central idea is to keep as much space as available in the update log minimizing the number of commits and also making sure that the tail erasure unit (EU) of the BMT update log is not over-written. Accordingly, there are two kinds of commit deployed by the BCM: **Popularity-based** and **Critical**.

Every BMT EU has to have information regarding the entire update log EUs where the BMT EU's pending updates are written to. This data structure is called a **dependency list** and is implemented as an array of lists named **ppq_2_update_log []**. This list is populated in **per_page_queue_update ()** and is released after the specific EU is committed. Similarly, every BMT update log EU has to have information regarding all the BMT EUs of whose updates it is holding. These dependency lists is the array **update_log_2_ppq []**. This list is populated in **BMT_update_log ()** and is released in **remove_from_ul2ppq_dependency ()**.

Popularity is defined as the number of BMT update log sectors occupied by updates belonging to one BMT EU. The BMT EU which has the maximum number of pending updates is called the most popular BMT EU. A simple array is used to maintain this popularity information, **bmt_eu_2_ul_popularity []**.The BCM starts operating after at least 25% of the BMT update log is full. At this point popularity-based commit happens i.e. updates of the most popular BMT EU will be committed so as to free maximum number of sectors from the update log. We bring in the desired EU to memory from the on-disk BMT using **read_bmt_page ()**, modify its content from the BMT cache using **PPQ_BMT_commit_build_page_buffer ()** and write it back using **write_bmt_page ()**.

Though popularity-based commit frees as many sectors as possible, it won't guarantee the advancement of the tail of the BMT update log, which means the re-usability of the tail EU of the BMT update log is still not safe. Critical commit happens when there's no BMT update log EU

available. In this algorithm, the BCM commits all the pending BMT entries in the tail EU of the BMT update log, which would free the tail EU of BMT update log. Thus, the critical commit remove the dependency of the tail eu's update_log_2_ppq dependency list as explained before. Intuitively, this would move the tail of BMT update log by at least one EU so that the head of the update log can advance further without any loss of data.

After every commit, we perform a check if the tail of the BMT update log can be moved further. This would help in the cases where the popularity based commit might be good enough to move the tail by itself instead depending on the critical commit.

5.3 Crash Recovery

The purpose of crash recovery is to reconstruct the system status after the crash, lost of the in memory data. The most significant information of LFSM system stored in the memory is the BMT entries in the BMT cache. Thus, the crash recovery module is responsible to reconstruct the BMT from the BMT update log after the system crash.

LFSM detecting the crash by examining the signature sector during system initialization in function generate_freemap_frontier. If the signature sector is LFSM_LOAD means LFSM was loaded before and didn't unload successfully. Thus, the BMT_crash_recovery is called to perform the crash recovery.

BMT_crash_recovery is the main function for the LFSM crash recovery. It read out all of the data in the BMT update log, whose address is BMT_update_log_start. The pending BMT entries are obtained by parsing the data of BMT update log. Finally, PPQ_BMT_cache_insert_one_pending is called to insert the pending BMT entries to the BMT cache.

LFSM completes the reconstructing procedure by calling update_ondisk_BMT to commit all of the BMT entries in the BMT cache to the on-disk BMT.

Generally, crash recovery reconstructs the pending BMT entries from the BMT update log and commits them to the on-disk BMT. Because the BMT update log is a circular buffer which guarantee for no data overwritten and all of the pending BMT entries are recorded in the BMT update log, the LFSM system status can be successfully reconstructed from the crash.

5.4 Signature Sector

When the driver is unloaded and re-loaded LFSM is supposed to get back its valid data. For this to happen, it should identify its previous head & tail positions of all active EUs. Also, LFSM should be able to decide whether the flash disk connected is a fresh disk or a previously used disk before checking for head & tail positions. To answer these questions, a signature sector is implemented.

The “signature” is a predefined ASCII value which spots the disk connected is a fresh one or an already used one and it also helps to identify if the crash recovery needs to start or not. If the LFSM is loaded, the signature field is assigned as “LFSM_LOAD”, and it will be assigned to “LFSM_UNLOAD” after the LFSM is successfully unloaded, or the LFSM is a fresh one and never been initialized before.

The “signature_successful_unload” is also a predefined ASCII value which decides whether the recovery algorithm has to start or not. If the driver is properly unloaded, no need to do a recovery. Else, recovery should start. The “physical_capacity” field is just an extra check to the above mentioned two signatures. It is redundant but enforces accuracy. “head” and “tail” fields store the value of head and tail of the logs at the time of unloading the driver so that while re-loading the driver the signature sector is read and the old head and tail could be restored.

Chapter 6

6 Synchronization

As described in main thread section, in order to convert random write to the sequential write LFSM invokes three I/Os to handle a single I/O request. That causes the race condition for conflict I/O requests, which means there're two I/O writing the same LBA and the late coming I/O might be ready to be submitted to the Flash while the previous I/O is in the BMT query stage. To solve this problem, LFSM uses active list to make sure the conflict I/O requests are processed according to their incoming order.

6.1 Active List

All of the processing I/O requests are stored in the active list, lfsm_dev_struct.datalog_active_list. Each I/O request in the active list is described by bio_container data structure. The insertion of the active list is handled by function get_bio_container(). Before processing a new incoming I/O request, get_bio_container checks if the I/O request conflicts with any items in the active list by traversing the datalog_active_list. If it's not a conflict I/O, a new bio_container is initialized and added to the active list and start to process. If it's a conflict I/O, the bio_container will be appended to the wait_list of every bio that it conflicts with, and the thread handling the conflict I/O will be added in the io_queue and status will be changed to pending.

After one I/O in the active list is finished, the status of entries in its wait_list is updated for the further process, handled in move_from_active_to_free() function**.** The thread of a conflict I/O will be woken up if all of the confliction is removed, whose number of confliction pages is equal to 0. The number of confliction pages is stored in the bio_container.conflict_pages.

Chapter 7

7 Garbage Collection

If we see the processing of a WRITE request by LFSM, we would understand that we always log the WRITES sequentially. It is noteworthy to observe that this sequential nature is with respect to one erasure unit i.e. inside one EU, keep the WRITES sequential. Though many of the blocks get overwritten with time, we cannot immediately over-write them since it would break the sequential write property of LFSM. Instead, we keep logging sequentially using the free blocks and mark the old over-written blocks as invalid. Thus, with time the number of invalid blocks increases and proportionally the number of free blocks decreases. Hence, to clean up the invalid blocks and make them re-usable (free), we do garbage collection (GC). The goal of GC is to maintain the balance between the invalid and free blocks. GC in LFSM is always done in the back ground thread in the function **gc_collect_valid_blks ()**.

Our garbage collection is EU-based. In other words, we collect valid blocks of one EU completely and move this EU to free pool and then proceed to do the same for another EU. We have a threshold (number of free EUs to the total number of EUs) to trigger the GC. Currently, it is 20% and is represented by GC_THRESHOLD_FACTOR. When this threshold is hit, GC starts in the background. Due to various reasons like scheduling, heavy IO in main thread, etc. there might be a case where the background GC might not be able to cope up to pump up the free pool and hence main thread mightn't find any free EU to process it's WRITE. In this scenario, the main thread yields to the background thread (to do the GC) and waits till it finds at least one free EU in the free pool. This we call **critical garbage collection**.

A good GC algorithm should have these features:

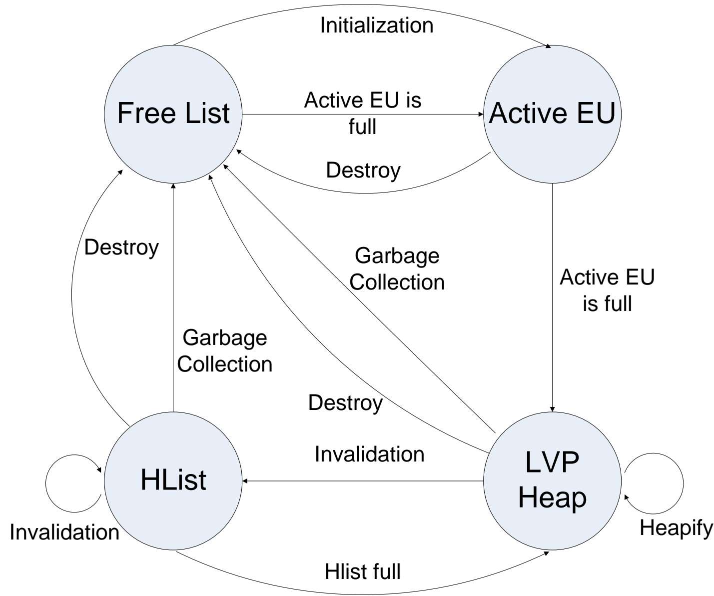
1. Minimize the number of valid pages copied. (Utilization)

2. The frequency of garbage collecting an EU should be proportional to the frequency of invalidation of blocks inside this EU. (Temperature)

LFSM satisfies the above mentioned criteria in a novel way as explained below.

7.1 Utilization

Utilization represents how many valid pages are there in a EU. So, when we want to garbage collect an EU, we pick the EU which has the least utilization so that we have to copy the minimum number of valid pages. After copying all the valid pages in the EU to new location, we move the old EU to the free pool. The data structure which is most efficient for this purpose is Min-Heap, which we call ***LVP_Heap***, where the root has the EU with the least number of valid pages. In log N time, we can insert and delete a EU from this heap. One important thing here is that we do not want to GC a EU whose pages are still being over-written (invalidated) currently. Because, this would result in copying those pages which would immediately get invalidated. Hence, we implemented something called a ***HList***. Once a EU which is in heap gets invalidated (a page gets over-written), this EU is moved from heap to HList. It is kept in HList until HList's length limit is reached. This limit is represented by HLIST_CAPACITY and is an experimentally found out to be100. This would give substantial time for the EU in HList to get invalidated as much as possible and by the time it gets moved to heap, it would be relatively inactive with respect to getting invalidated. In other words, we want to keep waiting a EU in HList to move to Heap until its utilization gets stabilized. Now, it can be safely garbage collected. Note that, if a EU in HList gets invalidated, it is moved to the head of the HList and when HList limit is reached, the EU from the tail of the HList is removed and inserted to Heap.



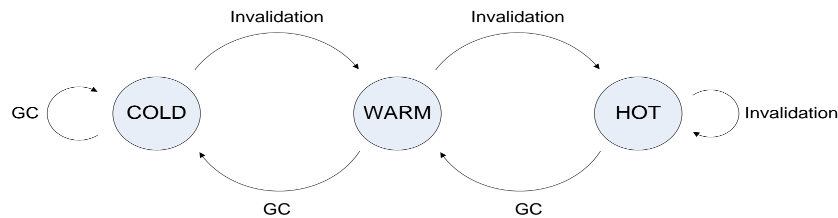
Life Cycle of an EU based on it's Utilization

Figure 7.1

7.2 Temperature

Temperature denotes the frequency of invalidation of the blocks. We assign the blocks which get frequently over-written as ***HOT***, the ones which are relatively stable as ***WARM***, and the ones which are almost never over-written as ***COLD***. For example, DLL files could be termed COLD, while TEMP files are treated as HOT. The idea is to group the blocks having the same temperature in the same EU. The assumption is that blocks having the same temperature generally die (invalidated) together. Hence, it makes sense not to garbage collect those EUs which have cold blocks as frequently as those EUs having warm blocks. Similarly, the EUs having warm blocks are garbage collected lesser number of times when compared to those having hot blocks. This will avoid the number of EUs that are actually garbage collected and also garbage collect those EUs which would give us more free blocks. Hence this improves the efficiency of the process. When a block is written for the first time, by default it goes to a cold EU. Once it gets over-written, it is moved to warm EU. If it's again over-written, it is moved to hot EU and stays there for any further invalidation. Similarly, if a hot block survives (remains

valid in the EU) a GC once, it is moved to warm EU. If it survives the 2nd GC, it is moved to cold EU and it stays there after any further GCs.



Life cycle of a block based on temperature

Figure 7.2

We keep the information regarding HList and LVP_Heap in struct HListGC and the information regarding the utilization & temperature of every EU in its respective struct EUProperty.

Since garbage collection and main thread WRITES run concurrently, there might be a possibility of conflicts i.e. both targeting the same LBN. For example, a GC WRITE finds out that the LBN its moving is already in the Active List. This means that particular EU having this LBN is being invalidated and hence would be moved to HList and shouldn't be garbage collected. Hence, we should abort the garbage collection of this EU.

7.3 Procedure

As explained earlier, the main goal of GC is to pump up the pool of free EUs. We would try and garbage collect enough EUs so that effectively we would generate at least one EU worth of free space. So we try to pick EUs from the heap one after another until we find out that garbage collecting these EUs would give us one EU worth free space. If we find that EUs in heap aren't enough to satisfy our constraint, we pick from hlist. Now we start the process of GC on this list of EUs we picked considering one EU at a time.

The information regarding the LBN of all the blocks in the EU is kept in a sector called metadata sector. This sector resides in the last block (8 sectors) of the EU. So, we will read this sector now and get the information of which LBNs are present and also how many of those are still valid using the EU bitmap. After getting this information, we would try to allocate bio containers to serve the WRITES of these valid pages. If we find any conflict with main thread WRITE, we

would stop of the GC of that EU and proceed to the next. Next, we will read the entire EU. After having the content of the EU in memory, we would move the EU to Free List from Heap/Hlist depending on the present location of the EU. Next step would be to assign new PBN to these blocks where they would end up eventually. Copy the content read before to the containers allocated and executed the WRITE one page after another. After this is done, release the containers and free the data structures promptly. Repeat this process for all the EUs in the pickup list.

Chapter 8

8 Distribution of BMT log and BMT Update log EUs (Floating)

As mentioned in the overview section of the design of LFSM, BMT log and BMT Update log are not contiguous. Their EUs are distributed across the disk. The actual layout is as shown in the below figure. The basic idea is to get all (almost) the EUs of the disk to the free pool so that some algorithms of “wear-leveling” and “erasure” could be incorporated in to the LFSM design.

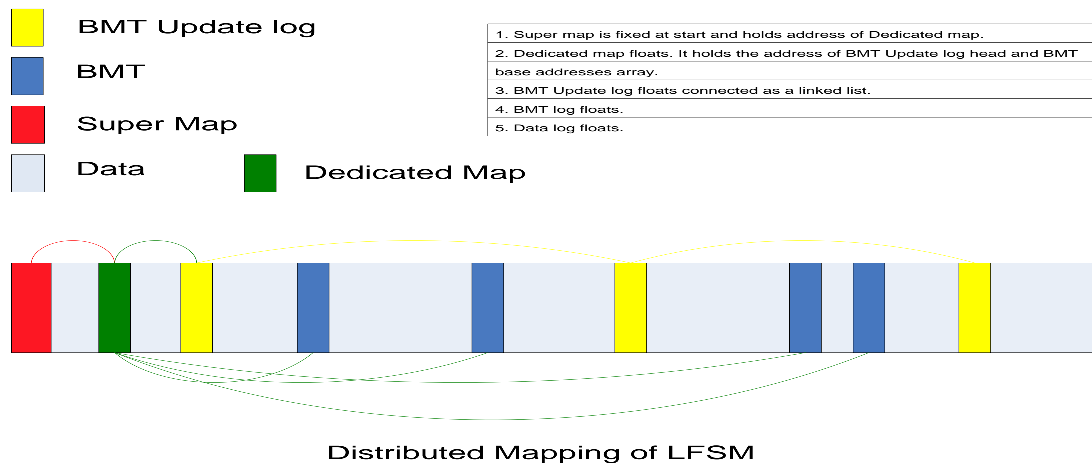


Figure 8.1

8.1 Super map

Super map is the only static section of LFSM. The first 2 EUs of the disk are reserved for this and these 2 EUs do not participate in wear-leveling. But, we will try to minimize the amount of writing to these 2 EUs so that they don’t get worn out soon. The information stored in the Super map is the address of the dedicated map EU and LFSM signature to identify fresh/re-used/crashed disk. Whenever the dedicated map EU changes, the new address of the dedicated map EU is logged in the Super map. This logging is sequential in these 2 EUs and when the 2 EUs are full, we wrap around. To the most updated super map entry, we use the latest sequence number approach.

8.2 Dedicated map

Dedicated map is a single EU which holds the address of the head EU of the BMT Update log and the array of base addresses of the BMT log EUs. We know that the head EU of the BMT update log will possibly change only after a BMT commit. Also, the base addresses of the BMT EUs will also change during that time only. So, after every commit, we have to log these two details to the dedicated map EU. Of course, we have to write sequentially inside this EU and once it's full, we move it to the free pool and allocate another dedicated EU from the free pool. In this scenario where the dedicated EU changes, the same event is logged to the Super map.

8.3 BMT Update log

As explained before, the address of the head EU of the BMT Update log is stored in the dedicated map and whenever the head changes, that event is logged to the dedicated map. BMT Update log size is statically decided and only those many maximum number of EUs can be allocated for the update log from the free pool. The EUs are connected with each other in the form of a linked list i.e. every update log EU holds in its last sector the address of the next update log EU. These EUs are linked for two reasons – 1) to read them sequentially one after another during crash recovery 2) to change the update log head after BMT commit, following the links.

8.4 BMT log

During initialization we allocate all the BMT EUs and hold their base addresses in an array called BMT_map[]. This information is logged in the dedicated map along with the update log head. Whenever a new EU replaces an existing BMT EU during commit, the change is reflected in the BMT_map[] and also it is logged to the dedicated map immediately. After a driver is re-loaded, we read the Super map to get the dedicated EU. We read the dedicated EU to get the BMT_map[] and thus we populate our BMT EUs.

8.5 BMT Lookup

The virtually linear BMT lookup is supported just by adding the base address offset from the BMT_map[] array to get the exact address of the BMT EU which we would like to fetch the BMT record from.

8.6 BMT Commit

Again, we have to maintain the linear BMT impression. When we read a BMT EU from disk, we move the EU to the free pool, allocate another EU from the free pool, write to the new EU and change the base address of that EU in the BMT_map[] array.

8.7 Crash Recovery

The crash recovery algorithm is almost similar to what was previously explained in section 2.9. Only difference is that to read the Update log, we have to do some extra steps – read super map, then read dedicated map to get the update log head and now start reading the BMT update log EUs one after another following the link. Remember that the address of the next update log EU is stored in the last sector of the current update log EU.

Chapter 9

9 Performance Evaluation

9.1 Methodology

In this section, we are going to use a synthetic work load to measure the average write and read end-to-end latencies to compare the performance of an LFSM-enabled flash disks and vanilla flash disks. Then we show the effectiveness of Interval-based Caching technique of LFSM running the block level traces of one day workloads from TPCC and CIFS servers. Finally, we will analyze the performance of the garbage collection algorithm by checking the effective time spent on copying the valid blocks per erasure unit.

9.2 Programs

Most of the tests we performed to check the read/write latencies of the flash disk are from the simulating programs written to emulate the sequential and random nature of access. This is done on purpose. Because, most of the real world IO load would never be completely sequential or completely random – it will be a mixture. Since, we know already that the performance of FLASH is excellent under sequential workload; the main challenge of LFSM is to answer the random write performance issue. Thus, we wanted to simulate 100% random work load to the flash disk and to achieve that our synthetic load generators are the best choice.

However, measuring the performance of our interval-based caching algorithm is a different issue since caching techniques should be measured not on 100% sequential or 100% random workloads. Rather, the effectiveness of any new caching technique should be measured on real world scenarios and hence we opted to use TPCC and CIFS servers' workloads.

9.3 Environment

We used a Dell Dimension 9200 system with 4GB RAM and Intel Core 2 Duo E6400 2.13GHz CPU having 2048KB L2 Cache. The disk is SAMSUNG 16GB 3.3V ATA5 UDMA66 SLC flash disk. Operating system is Fedora 10 distribution and kernel version 2.6.25.14.

9.4 Results

The below table shows the end-to-end latencies of read and write requests of size 4KB to the vanilla flash disk. The readings are taken after sending 1 Million requests.

	Sequential	Random
Read	**364**	**376**
Write	**906**	**38868**

Table 9.1 Latencies in Microseconds of a 4KB request of vanilla flash disk.

The below table shows the end-to-end latencies of read and write requests of size 4KB to an LFSM-enabled flash disk. The readings are taken after sending 1 Million requests.

	Sequential	Random
Read	**384**	**398**
Write	**1866**	**2933**

Table 9.2 Latencies in Microseconds of a 4KB request of LFSM.

If we observe the last row of the above table, it's interesting to see that the LFSM's sequential write performance is a little less when compared to the vanilla flash disk. But, if we notice the random write latency, LFSM beats the vanilla disk's performance by a magnitude. To understand these numbers in detail, we need to have a look at a more detailed version of the above table where the total latency is expanded in terms of the delaying components.

	Critical Commit	BMT Lookup	Data Logging	BMT Update logging	Total latency
Sequential	**0**	**14**	**906**	**948**	**1866**
Random	**11**	**29**	**1465**	**1435**	**2933**

Table 9.3 Latencies in Microseconds of a 4KB request of LFSM expanding on delaying components.

As it is clear in the above table, that the overhead in the LFSM design is the BMT Update logging, which has to be done to ensure data protection against system crashes. But, if we consider the overall improvement of performance by LFSM to the flash disk's vanilla performance, because of the magnitude gain in the random write performance, LFSM alleviates the performance bottle necks of FLASH memory.

Now, let's see what is the time consumed in the background thread by our temperature based garbage collection technique. The below table explains those details.

	Read EU metadata	BMT Lookup	Data logging	BMT Update loggin	Total latency per EU
Sequential	**11**	**161**	**1105**	**2708**	**4003**
Random	**57**	**417**	**929**	**2103**	**3511**

Table 9.4 Latencies in Microseconds of temperature-based garbage collection per erasure unit (EU).

We see that the garbage collection overhead of LFSM is very low because of the sophisticated temperature-based garbage collection algorithm. LFSM is incurring 4003 microseconds to do the garbage collection of 1 erasure unit which would have 128 4KB pages in general on a sequential workload. That is, 31.2 microseconds per 4KB page which is very minimal. On a random workload, the garbage collection results are still better with a 3511/128 = 27.42 microseconds per 4KB page.

Chapter 10

10 Conclusion and Future Work

On conclusion, we have presented a storage system called Log Structured Storage Manager (LFSM) which alleviates the random write performance bottleneck of flash disks to a great extent. We measured the performance numbers of LFSM over that of vanilla flash disk and found out that LFSM defeats the vanilla flash disk's overall performance by reducing the random write latency by 10 times though we incur a slight overhead in case of sequential write latency. We have designed, implemented and evaluated several new techniques in the LFSM – Interval-based Cache, Temperature-based garbage collection and BUSC principle – and came to a definitive opinion that they all work extremely wonderful for FLASH memory.

When we see table 9.3, we would observe that the major overhead which LFSM incurs is that of the BMT Update logging. To re-state, we have to do the BMT Update logging to keep LFSM design safe from any system crashes in which case, the in-memory updates to the BMT would be lost. In the current implementation, if we consider a single write request, BMT Update logging is started after the Data logging is done in order. That is, we have to wait till we get the interrupt from the disk for the data logging and then we have to do the BMT Update logging. This limitation is only because we do not have the luxury to do both the loggings concurrently from the operating system level. Therefore, if we can somehow do both data logging and BMT Update logging concurrently to the flash disk, this small overhead of BMT Update logging would also be completely nullified.

We implemented LFSM at block layer of Linux kernel at which we didn't had the luxury of exercising different planes of FLASH memory. However, FTL (Flash Translation Layer, firmware of FLASH) can definitely do it. That is, FTL can send writes to different planes inside the FLASH memory concurrently. Thus we believe that if LFSM can be ported to FTL, data logging and BMT update logging can be directed to different planes so that they both get executed concurrently. In this way, we overcome the overhead of BMT Update logging completely and the small extra time which we are incurring in case of sequential writing would not be present. Thus, for future work, we are expecting to move LFSM from block layer to FTL to get the best performance out of the FLASH memory.

Bibliography

[1] Maohua Lu and Tzi-cker Chiueh, “An Update-Aware Disk I/O System for High-Performance Database Indexes,” Technical Report, Stony Brook University, 2008.

[2] E. Harari, R. D. Norman, and S. Mehrota, Flash EEPROM System, Number 5,602,987. United States Patent, 1997 February.

[3] Intel Corporation., Understanding the Flash Translation Layer (FTL) Specification, http://www.infinibandta.org/.

[4] Andrew Birrell, Michael Isard, Chuck Thacker, and Ted Wobber, “A Design for High-Performance Flash Disks,” SIGOPS Oper. Syst. Rev., vol. 41, no. 2, pp. 88–93, 2007.

[5] Chanik Park, Jaeyu Seo, Dongyoung Seo, Shinhan Kim, and Bumsoo Kim, “Cost-Efficient Memory Architecture Design of NAND Flash Memory Embedded Systems,” in ICCD ’03: Proceedings of the 21st International Conference on Computer Design, Washington, DC, USA, 2003, p. 474, IEEE Computer Society.

[6] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy, “Design tradeoffs for SSD performance,” in ATC’08: USENIX 2008 Annual Technical Conference on Annual Technical Conference, Berkeley, CA, USA, 2008, pp. 57–70, USENIX Association.

[7] Mendel Rosenblum and John K. Ousterhout, “The Design and Implementation of a Log-Structured File System,” ACM Trans. Comput. Syst., vol. 10, no. 1, pp. 26–52, 1992.

[8] 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,” Trans. on Embedded Computing Sys., vol. 6, no. 3, pp. 18, 2007.

[9] S.H.; Sang Lyul Min; Yookun Cho Jesung Kim; Jong Min Kim; Noh, “A Space-Efficient Flash Translation Layer for CompactFlash Systems,” IEEE Transactions on Consumer Electronics, vol. 48, no. 2, pp. 366–375, May 2002.

[10] Amir Ban, Flash File System, Number 5,404,485. United States Patent, 1993.