Computers **2014**, *3*, 36-57; doi:10.3390/computers3010036

OPEN ACCESS

computers

ISSN 2073-431X

www.mdpi.com/journal/computers

Article

A Scalable and Highly Configurable Cache-Aware Hybrid Flash Translation Layer

Jalil Boukhobza *, Pierre Olivier and Stéphane Rubini

University Bretagne Occidentale, UMR 6285, Lab-STICC, F-29200 Brest, France;
E-Mails: pierre.olivier@univ-brest.fr (P.O.); stephane.rubini@univ-brest.fr (S.R.)

* Author to whom correspondence should be addressed; E-Mail: boukhobza@univ-brest.fr;
Tel.: +33-2-98-01-69-73; Fax: +33-2-98-01-80-11.

Received: 23 January 2014; in revised form: 7 March 2014 / Accepted: 18 March 2014 / Published: 24 March 2014

Abstract: This paper presents a cache-aware configurable hybrid flash translation layer (FTL), named CACH-FTL. It was designed based on the observation that most state-of-the-art flash-specific cache systems above FTLs flush groups of pages belonging to the same data block. CACH-FTL relies on this characteristic to optimize flash write operations placement, as large groups of pages are flushed to a block-mapped region, named BMR, whereas small groups are buffered into a page-mapped region, named PMR. Page group placement is based on a configurable threshold defining the limit under which it is more cost-effective to use page mapping (PMR) and wait for grouping more pages before flushing to the BMR. CACH-FTL is scalable in terms of mapping table size and flexible in terms of Input/Output (I/O) workload support. CACH-FTL performs very well, as the performance difference with the ideal page-mapped FTL is less than 15% in most cases and has a mean of 4% for the best CACH-FTL configurations, while using at least 78% less memory for table mapping storage on RAM.

Keywords: NAND flash memory; hybrid-mapping; flash translation layer; cache; solid state drives

1. Introduction

Semiconductor-chip-based nonvolatile memories (NVM) are becoming more widely used and are no longer confined to embedded systems. According to MarketResearch.com [1], the NVM market will

increase by a mean of 69% annually until 2015. This is partly due to the extensive usage of flash memories, but is also caused by the emergence of new NVM technologies, such as ferroelectric RAM (FeRAM), phase change RAM (PCRAM), and magneto-resistive RAM (MRAM). Even though some firms are beginning to move toward mass production of MRAM, the most mature and commonly used NVM for data storage in various consumer electronics is still NAND flash memory.

Its attractive performance, energy efficiency and shock resistance features have made NAND flash memory increasingly popular in smartphones, tablet PCs, multimedia players and even in enterprise storage systems. In fact, the growing adoption of flash memories is considered the most important technology change relevant to the field of data-centric computing [2]. As the gap between the processing power of computer systems and the performance of traditional storage systems continues to grow, it becomes necessary to insert a new memory technology into the traditional memory hierarchy to carry on feeding the growing number of processing elements with data. The current best candidate is flash memory, as it gives a very interesting performance-per-cost ratio.

Even though NAND flash memory presents some very interesting characteristics, it also has some limitations caused by its internal intricacies. Basically, the smallest addressable data unit in flash memory is a page (two to 8 KB), and a fixed set of pages (usually 64) composes a block. Some NAND flash memory operations can be executed on pages, while others are executed on blocks. The main constraints are: (1) write/erase (W/E) asymmetry; writes are performed on pages, whereas erasures are realized on blocks; (2) erase-before-write limitation; a costly erase operation is necessary before data can be modified; (3) limited number of W/E cycles; the average number is between 5,000 and 10^5, depending on the technology used.

From the performance point of view, it is commonly accepted that flash memories often outperform traditional secondary storage hard disk drives (HDD), except for random write operations, where performance depends highly on the flash memory internals. Random write operation performance is the Achilles' heel of flash memory: if not efficiently managed, it can even be worse than that of HDDs.

The flash translation layer (FTL) is a hardware/software layer intended to overcome the aforementioned limitations: (1) The erase-before-write and the W/E granularity asymmetry constraints imply that data updates should be performed out-of-place. Hence, the logical-to-physical mapping scheme, which is a critical issue, is used to manage these updates. Mapping tables are stored in an embedded RAM. (2) Out-of-place data updates require the use of a garbage collector to recycle blocks enclosing invalid pages in order to recover free space. (3) To minimize the limitation on the number of W/E cycles, FTLs try to evenly distribute the wear over the memory cells. This wear leveling prevents some memory cells from wearing out more quickly than others.

FTL mapping schemes depend on the granularity with which mapping information is managed. They can be classified into three groups: page, block and hybrid mappings [3]. (1) The page-mapping scheme maps each logical page into a physical page independently of the other pages of the same block. It is very flexible and gives good performance, but it requires too large of a mapping table to fit into the embedded RAM. (2) The block-mapping scheme considers the granularity of a block rather than a page. The logical page address is composed of the logical block number and a fixed page offset that is not modified by the mapping process. The block-mapping scheme is more feasible in terms of table size; however, its main drawback is that a page update systematically triggers a whole block erase operation and several valid page copies to another block. (3) The hybrid-mapping scheme was proposed to

overcome the above-mentioned shortcomings by combining both types of mapping. It is generally based on a block-mapping scheme and uses page-mapping for a small number of blocks (see the Related Work section below). Throughout this scheme, designers try to get as close as possible to the performance of a page-mapping scheme, while keeping the mapping table RAM usage as close as possible to the block-mapping table size.

Many caching mechanisms [4–11] have been designed to facilitate the work of the underlying FTL. These cache systems absorb part of the data updates and attempt to reveal sequentiality in order to evict the largest set of pages from the same block at the same time. Grouping pages allows a reduction in the number of block erasures performed. State-of-the-art caching systems are designed independently of FTLs for reasons of genericity, and FTL designs do not take into account upstream caches. However, flash-specific caches have a common feature consisting of flushing groups of pages from the same block.

This paper describes CACH-FTL [12], a cache-aware configurable hybrid FTL designed to optimize write performance and embedded memory usage. This optimization is achieved through a flexible and efficient data placement mechanism. With CACH-FTL, the flash memory is partitioned into two regions: (1) a page-mapped over-provisioning region, named PMR; and (2) a block-mapped data region, named BMR. CACH-FTL selectively places data flushed by the above cache either in the PMR or in the BMR, depending on the number of flushed pages. If this number is above a (configurable) threshold, written data are considered as sequential and, thus, directed toward the BMR; as block-mapping is well suited for large write operations with a high spatial locality (pages from the same block). If the number of flushed pages is below the predefined threshold, the small dataset is sent to the PMR, which works as a second-level in-flash buffer. Small groups of pages are temporarily placed into the page-mapped PMR in order to: (1) generate less ineffective erase operations and; (2) collect more pages before moving them to the BMR.

In the present study, CACH-FTL was tested on a large number of real and synthetic I/O workloads and performed well for both random and sequential ones. Its performance was close to the ideal page-mapping scheme performance, but it used a much smaller mapping table, which makes it much more scalable.

The next sections present an overview of flash memories and some related works. The fourth section of this paper then details the CACH-FTL structure and algorithms. This is followed by a description of the performance evaluation method and a presentation of the results. Finally, the conclusions and some perspectives for future work are given.

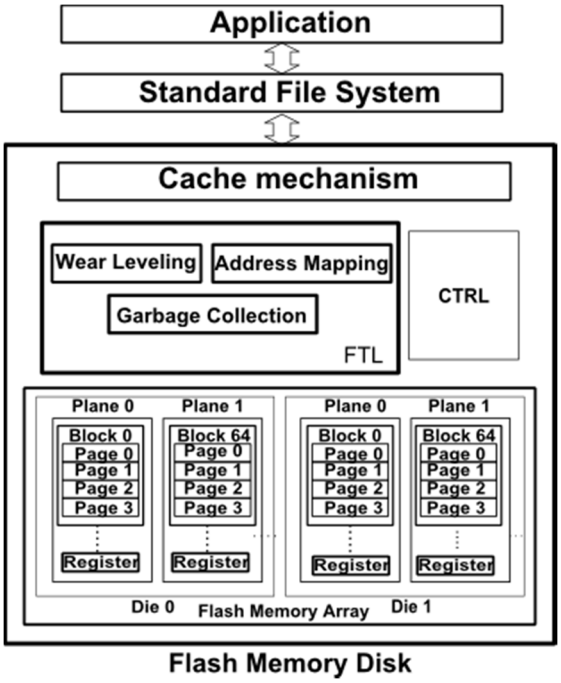
2. Overview on NAND Flash Memory

Flash memories are based on floating gate transistors and include two main implementations: (1) NOR and (2) NAND flash memories. NOR flash memories are reliable (no need for error correction code), support byte random access and have a lower density and higher cost than NAND flash memories. NOR flash memories are used for storing code [13]. NAND flash memories, in contrast, are block addressed, offer a higher storage density at a lower cost and provide good performances for large read/write operations. They are used as secondary storage [13]. Other types exist, such as divided bit line NOR, AND-type and some specific embedded flash technologies. This paper only concerns the NAND-type.

Basically, there are three main types of NAND flash memories: (1) single-level cell (SLC); (2) multi-level cell (MLC); and (3) triple-level cell (TLC). In SLC flash memories, only one bit can be stored per cell, whereas two bits can be stored in MLC and three in TLC. From the point of view of bit density and cost per bit, TLC is the best, followed by MLC and, finally, SLC. From a performance and reliability point of view, SLC outperforms MLC, which further surpasses TLC. While TLC is more frequently used for low-end media players, mobile GPS and, more generally, non-critical data applications that do not require frequent updates, MLC and SLC are mostly used for more data intensive appliances, such as Solid State Drives (SSDs) and mobile phones.

Flash memory is structured as shown in Figure 1: a chip is composed of one or more dies; each die is divided into multiple planes. A plane is composed of a fixed number of blocks, each of which encloses a fixed number of pages that is typically a multiple of 64. Current versions of flash memories have between 128 and 1,024 KB blocks (with pages of two, four or 8 KB). A page consists of a data space and a metadata out-of-band (OOB) area containing the page state, information on the error correction code (ECC), *etc.* Three operations can be carried out on flash memories: reads and writes, which are realized on pages, and erases, which are performed on blocks. As shown in Figure 1, a flash memory disk contains a flash translation layer managing wear levelling, address mapping and garbage collection and other services in the controller part (controller CTRL in the figure).

Figure 1. Flash disk logic components. FTL, flash translation layer.



3. Related Work

3.1. FTL Schemes

Hybrid-mapping schemes aim for the performance of a page-mapping scheme with the memory usage of block-mapping. Most are based on a primary block-mapping scheme, whereas others partition the flash space according to I/O characteristics into a page and a block-mapped space.

Hybrid-mapping schemes with primary block-mapping are mainly based on the use of log-blocks. Log-blocks are spare blocks used to avoid a block copy for each page update. At least one block copy is generated when the log-block is full and a new update is required. A log-block can be either dedicated to one data block (such as M-Systems [14], AFTL, [15], CNFTL [16] and BAST [17]) or shared between many (such as RNFTL [18], FAST [19] and KAST [20]). The associativity of pages in log-blocks is also crucial for mapping performance. While the first FTLs used log-blocks that are directly mapped to data blocks, as in ANAND [14], most recent FTLs are fully associative (FAST [19], LAST [21], HFTL [22], BlogFTL [23]). In these hybrid FTLs, page-mapping is generally used to map the pages of the log blocks (FAST, BAST).

The second type of hybrid FTL partitions the flash memory into a page-mapped and a block-mapped space (such as WAFTL [24], CFTL [25]). According to the data access type and/or pattern, pages are directed toward the adequate space. Indeed, as noted in [24], page-mapping is better suited (even if more expensive) to random writes than is block-mapping, which is nonetheless sufficient for sequential read and write operations.

From the partitioning point of view, CACH-FTL can be considered as closer to the latter type, as it divides the flash space into a page-mapped region and a block-mapped region. However, from the type of data stored point of view, CACH-FTL resembles log-block based FTLs, as it stores temporary data into the page-mapped region. While WAFTL and CFTL use some more or less complex mechanisms to detect I/O pattern characteristics in order to direct data toward the more adequately mapped space, CACH-FTL abstracts the applicative layer and uses a simpler algorithm that relies only on the above cache output to decide whether to write data temporarily into the page-mapped or directly to the data block-mapped space. The parameter on which CACH-FTL relies is the number of evicted pages. However, unlike CFTL and WAFTL, and like log-block-based FTL schemes, such as FAST, in CACH-FTL, the page-mapped space is used only to buffer small page sets before merging them with the block-mapped data.

CACH-FTL allows the use of information coming from the cache system above it to simplify the FTL structure, to abstract the higher layers and to be agnostic to the cache system and, at the same time, giving better performance. It also provides a high degree of flexibility and can be (re)configured according to upper layers (cache and I/O workload).

3.2. Flash-Specific Cache Systems

In order to optimize the performance of write operations on flash memories, different cache systems can be placed above the FTL. Most of them reflect the granularity of the erase operations by dealing with groups of pages (FAB [4], CLC [5], BPLRU [6], BPAC [7], LB-Clock [8], PUD-LRU [9], REF [10] and C-lash (cache for flash) [11]). These caches try to achieve two goals: (1) maximizing the number of flushed pages (from a given block); and (2) evicting data that are unlikely to be accessed (temporal and spatial locality). To realize Goal 1, most caches evict the largest set of pages belonging to the same block. For Goal 2, caches generally use LRU algorithms while dealing with page groups.

One common characteristic of flash-specific caches is that they all flush groups of pages belonging to the same in-flash block. The idea behind CACH-FTL is to make use of this parameter as an indicator for deciding whether to evict the group of pages in the page-mapped or the block-mapped space. The

number of pages in the flushed group is a valuable indicator of the performance cost of the generated block update operation. The smaller the group of pages is, the higher the cost (for a given set of valid data still residing in the flash memory). CACH-FTL uses this indicator so that small groups of flushed pages are mapped by page, thereby temporarily eliminating the cost of the block merge operations, while large groups are mapped by block. Indeed, in CACH-FTL, the page-mapped area is considered a second-level buffer that allows more space to group several sets of page updates.

The information gap between FTLs and the flash memory buffer has been discussed in [26], where the authors proposed to handle this by making the cache and the FTL cooperate. This is done through a modification of the cache in order to provide many candidates for flushing data, and on the other hand, upgrading the FTL by making it achieve a decision on the data to flush according to the cleaning cost. The proposed solution is pertinent and can be complementary to the work presented in this paper; however, it induces modifying both the buffer and the FTL layers, while we propose in this paper a FTL that imposes no modification on the cache and only uses flushed data information. Both approaches try to cope with the same information gap issue.

3.3. Motivation

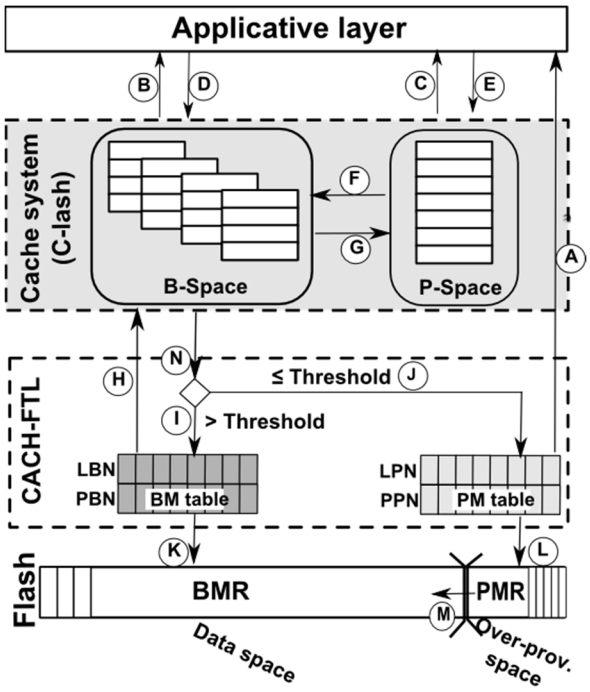
The design of CACH-FTL was motivated by the need to exploit the most pertinent information in a simple way to perform data placement in a hybrid FTL. Indeed, many state-of-the-art solutions focused on I/O pattern-based heuristics to choose the adequate mapping granularity for data placement. In CACH-FTL, we used a simpler, yet very efficient, solution that relies on a realistic architectural model supposing a cache system on top of the FTL layer, which is the architecture used in most real-world applications. CACH-FTL uses information coming from the cache to perform data placement on a page-mapped or block-mapped region. The objective, through CACH-FTL, was also to bridge the information gap between FTLs and cache systems, as both mechanisms were developed separately in state-of-the-art work. Nevertheless, this was done by staying as cache-agnostic as possible, since CACH-FTL can work with all cache systems that flush groups of pages, and most caches for flash fall into this category. In addition, CACH-FTL is scalable, as it can scale on both small and large cache proportions, as the mapping table size can be adapted accordingly.

4. Cache-Aware Configurable Hybrid FTL Design

The CACH-FTL system architecture is illustrated in Figure 2. CACH-FTL splits the flash memory into two regions: (1) an over-provisioning region managed with a page-mapping scheme (called PMR); and (2) a data region managed by the use of a block-mapping scheme (called BMR). The BMR size dictates the addressable space as seen by the applicative layer.

The CACH-FTL scheme is generic, as it can be used together with any cache system, provided that it flushes groups of pages from the same block. In this paper, C-lash (cache for flash) was considered as the cache system example [11] for the performance evaluation part. Using some other cache mechanisms proved to have similar results.

Figure 2. Cache-aware configurable hybrid (CACH)-FTL system architecture. C-lash, cache for flash; b-space, block space; p-space, page space; BMR, block-mapped region; PMR, page-mapped region; BM, block-mapping; PM, page-mapping (LBN, PBN: logical and physical page number respectively).



4.1. Overview of the C-Lash System

Cache systems *per se* are beyond the scope of this paper, as its main contribution is the FTL scheme implemented below the cache (see Figure 2). However, for reasons of clarity, a very brief overview of C-lash is given in this section.

In C-lash [11], a typical cache for flash, the cache space is partitioned into two spaces, a page space (p-space) and a block space (b-space). P-space consists of a set of pages that can come from different logical blocks, whereas b-space is composed of blocks (which are further composed of pages). Pages and blocks have the same size as those of the underlying flash memory. Both b-space and p-space have fixed sizes. The C-lash system is hierarchical, as it has two levels of eviction policies: one that evicts pages from p-space to b-space (F in Figure 2); and another in which blocks from b-space are evicted into the flash media (N in Figure 2).

When a read request arrives from upper layers, requested data are first searched for in the cache (B and C in Figure 2). If the data are not available, the read request is forwarded to the underlying FTL (A in Figure 2).

When a write request is issued, and the data to be modified are in the cache, then they are updated in place (E or D in Figure 2). When a write miss occurs, data are written into a free page of the p-space (E in Figure 2). If there is no space available, the first-level eviction policy is triggered to flush the largest group of pages from p-space to b-space (F in Figure 2). If no space is available in the b-space to receive pages from the p-space, the second eviction policy is launched to flush a block from the b-space to the flash (N in Figure 2) following an LRU algorithm.

4.2. CACH-FTL Scheme Management

As stated above, CACH-FTL partitions the flash memory into two separate regions with two different mapping tables that are both maintained within the embedded RAM.

4.2.1. Read Operation Management

When requested data are not present in the cache, the cache system forwards the request to CACH-FTL. Data can be located either in the BMR or in the PMR if not yet flushed to the BMR. CACH-FTL first checks for data in the BMR by looking at the validation bit of the mapping table. If data are not valid, the read request is forwarded to the PMR.

4.2.2. Write Operation Management

From the CACH-FTL point of view, write operations always come from the above cache system when an eviction occurs (N in Figure 2). CACH-FTL defines a redirection threshold according to the number of evicted pages. This threshold allows making a decision on to where the group of victim pages from the cache should be flushed (see Algorithm 1, Functions 1, 2 and 3). The redirection threshold gives the number of pages above which the evicted group of pages is sent to the BMR (I then K in Figure 2) and under which they are sent to the PMR (J then L in Figure 2) for a later update to the BMR. Indeed, a low number of flushed pages means that the system has undergone a burst of random write operations. The pages are then directed toward the PMR to avoid a costly update/merge operation on the BMR (valid page read, block erase and page write operations; see Algorithm 1, Function 2).

The over-provisioning space used (PMR space) is generally larger than the cache; its size can vary between 5% and 30% of the flash memory (this is the interval used in state-of-the-art studies). In CACH-FTL, the objective of the PMR is to buffer as many random writes as possible in order to group larger sets of pages (from the same block), with the objective of reducing the update cost before moving data blocks into the BMR data-space (M in Figure 2). When facing a long random write burst, the PMR rapidly becomes full of dirty or busy blocks. Therefore, in order to recycle some free space, a garbage collection (GC) mechanism is implemented. The designed GC works in two stages: first, it tries to recycle blocks by compacting valid pages within the PMR (and recycling invalid blocks); if this is not sufficient, data are then moved to the BMR to free some blocks in the PMR. Each of these mechanisms is detailed in the following sections.

Algorithm 1 shows the write operation algorithm followed by CACH-FTL. Function 1 shows the test on the redirection threshold value and switches to Function 2 in case of a BMR write or Function 3 in case of a PMR write operation.

In Function 3, Line 29 tests whether there is enough free space; indeed, if the free space falls under a given limit, the GC is launched (see Line 35). Otherwise, the group of pages is written to the PMR, and the mapping table is updated.

Algorithm 1. CACH-FTL write algorithms.

1: **input**:
2: Redirection page group size threshold: $NbPages_{Threshold}$
3: Number of evicted pages: $NbPages_{Evict}$
4: The group of pages to evict: $GroupPages_{evicted}$
5: The block in the flash corresponding to data to evict: $Block_{evict}$
6: Free pages in the PMR: $FreePages_{PMR}$
7: PMR free pages synchronous GC threshold: $SyncFreeP_{Threshold}$ // see the following GC section for details on this variable
8: **FUNCTION 1: CACH-FTL_Write_To_Flash** ($NbPages_{Threshold}$, $NbPages_{Evict}$, $GroupPages_{evicted}$)
9: **if** ($NbPages_{Evict} > NbPages_{Threshold}$)
10: *Flush_Pages_BMR*($GroupPages_{evicted}$)
11: **else**
12: *Flush_Pages_PMR*($GroupPages_{evicted}$)
13: **end if**
14: **FUNCTION 2: Flush_Pages_BMR**($GroupPages_{evicted}$)
15: **if** ($GroupPages_{evicted} < NumberOfValidPagesInABlock$)
16: //Read valid pages in flash from the BMR and/or PMR
17: Read ($Block_{evict} - GroupPages_{evicted}$)
18: Erase $Block_{evict}$ // erase the block in the flash memory
19: Flush the block from the cache → $Block_{evict}$
20: Update page-mapping tables (PMR if needed and BMR)
21: Delete $GroupPages_{evicted}$ from the cache
22: **else**
23: Erase $Block_{evict}$
24: Write the $GroupPages_{evicted}$ → $Block_{evict}$
25: Update the block-mapping table of BMR
26: Delete $GroupPages_{evicted}$ from the cache
27: **end if**
28: **FUNCTION 3: Flush_Pages_PMR**($GroupPages_{evicted}$)
29: **if** ($FreePages_{PMR} - GroupPages_{evicted} > SyncFreeP_{Threshold}$)
30: Flush $GroupPages_{evicted}$ → PMR
31: Update the page-mapping table PMR
32: **else**
33: // space available in PMR is < $SyncFreeP_{Threshold}$
34: *Activate Noninterruptible GC*
35: *PMR_GC(*$NbPages_{Evict}$*)* // see Algorithm 2.
36: Flush $GroupPages_{evicted}$ → PMR
37: Update page-mapping table PMR
38: **end if**

4.2.3. BMR Block-Mapping Scheme

Function 2 shows how pages are flushed into the BMR in the case of a page group, the size of which is greater than the threshold. The algorithm first checks if there are some valid pages in the target on-flash data block. If so, it reads first all the valid pages into a dedicated cache block before erasing the block. Then, it flushes the whole block (from the cache) into the on-flash block and, finally, updates the mapping tables.

As one can see from Lines 17–19, 23 and 24, for the sake of this study, a simple direct (block) mapping in-place update algorithm was used. Therefore, when updated, a given block is written in-place. Direct block-mapping does not allow the available flash memory free space to be used to optimize the performance of CACH-FTL. This is not an optimal solution in terms of performance, but it allows a better performance evaluation, as results could be biased by flash memory free space usage, depending on the applied I/O workload. Modifying the direct mapping scheme would allow a better FTL performance (this is to be achieved in future work).

Adding a wear leveler into CACH-FTL is also beyond the scope of this paper, as the evaluation of such a mechanism would require a separate performance evaluation study. However, several state-of-the-art wear levelers can be adapted to CACH-FTL either locally on each region (BMR or PMR) or globally all over the flash memory space.

4.3. CACH-FTL Garbage Collection (GC) Mechanisms

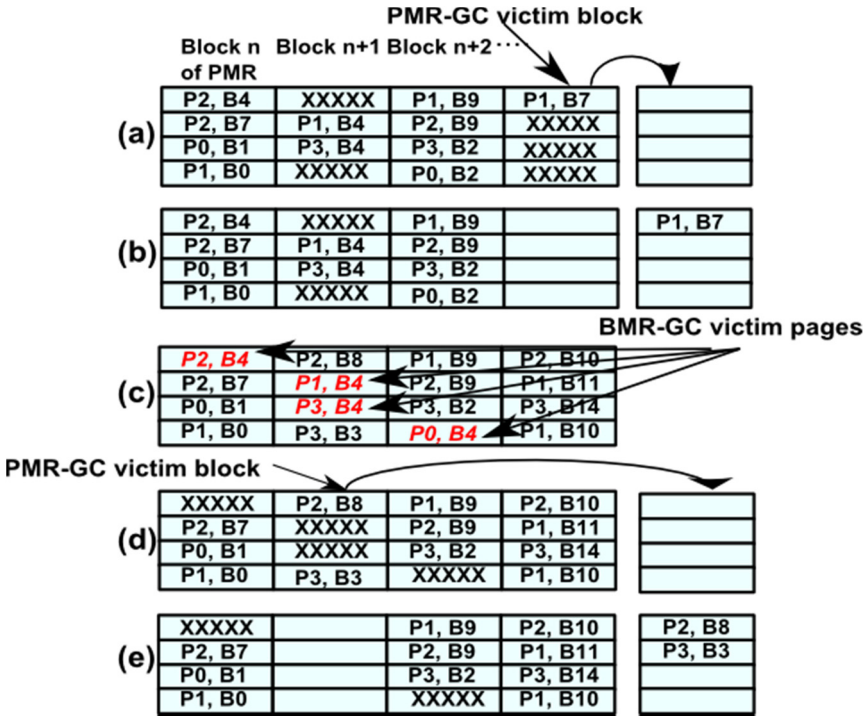
Two garbage collection mechanisms are used in CACH-FTL, one for each flash region. The PMR garbage collector (PMR-GC) recycles invalid pages inside the PMR in order to recover free space within the PMR. The BMR garbage collector (BMR-GC) moves data from the PMR to the BMR when there is no free space available in the PMR, even after the PMR-GC is performed.

4.3.1. PMR Garbage Collector

When the number of free blocks in the PMR goes under a predefined threshold, the PMR-GC is launched (see Algorithm 1, Line 35). CACH-FTL uses a simple greedy reclamation algorithm that selects the physical block from the PMR containing the least number of valid pages. Valid pages from the chosen block are copied to a free block, and then, an erase operation is triggered to recycle the victim block.

Algorithm 2, Function 4 then Function 5, describe the PMR-GC behavior. While there are still pages to evict from the cache (Line 20), the PMR-GC carries on its work. The system scans all the flash blocks of the PMR to detect the one containing the larger set of invalid pages. (Lines 21 and 22). If there are no invalid pages to recycle (all data in the PMR are valid), the BMR-GC is launched (Lines 24 and 25; see the next section). Otherwise, the system reads the valid pages from the selected block, copies them into a free block, recycles the selected block and, finally, updates the mapping table (Lines 27–31). Once done, it checks whether enough space has been recycled to copy the whole set of pages to evict, and if not, it goes for another round of PMR-GC. An example of the PMR-GC behavior is given in Figure 3a,b.

Figure 3. Examples of garbage collection (GC): each page of PMR can belong to a given data block (for instance, in the top left, Block n contains Page 2 of Data Block 4). In (**a**), when a PMR-GC is launched, CACH-FTL chooses the block with the maximum number of invalid pages (marked "XXXXX"). It copies the valid pages to a new block and erases the original one, which finishes with State (**b**). In (**c**), the PMR is full. A BMR-GC is launched, and the system chooses the largest group of pages belonging to the same block; in this example, B4 (four pages). It then copies those pages to the BMR and invalidates them (State (**d**)). Once done, the PMR-GC is launched to free a block. It then chooses the second block (State (**d**)), copies the two valid pages and erases the first block (State (**e**)).



4.3.2. BMR Garbage Collector

The BMR-GC is launched whenever the PMR-GC cannot find any physical block containing enough invalid pages to recycle. BMR-GC also uses a greedy reclamation algorithm selecting the largest group of PMR pages belonging to the same data block (from BMR), as shown in the example in Figure 3c. Once the pages in the PMR are identified, the system searches for valid pages from the same block in the BMR in order to launch a merge operation (note that valid pages in the cache are not merged during GC). If some pages are found, the valid block pages (from the PMR and BMR) are moved into a dedicated block of the cache. The related block of the flash memory is then erased, and all the pages are flushed (see Figure 3).

Function 6 of Algorithm 2 describes the actions performed during the BMR-GC. The loop in Lines 36 and 37 searches in the PMR for the largest set of pages from the same data block (BMR). Once found, pages are read and copied to the BMR (through the cache), and finally, the mapping tables are updated (Lines 38–42).

Algorithm 2. CACH-FTL garbage collection algorithms.

input:
Number of evicted pages: $NbPages_{Evict}$
The group of pages to evict: $GroupPages_{evicted}$
The block in the flash of data to evict: $Block_{evict}$
Free pages in the PMR: $FreePages_{PMR}$
PMR-GC free page asynchronous threshold: $AsyncFreeP_{Threshold}$
PMR free page synchronous GC threshold: $SyncFreeP_{Threshold}$
The PMR block with the largest number of invalid pages: $Block_{MaxInvalid}$
Number of invalid pages in $Block_{MaxInvalid}$: $NbPages_{Invalid}$
Maximum number of pages in PMR belonging to the same data block: $MaxGroupPages_{PMR}$
// FUNCTION 4 is launched at the end of each I/O request
FUNCTION 4: PMR_Garbage_Collector()
if ($FreePages_{PMR} \leq AsyncFreeP_{Threshold}$)
  Activate Interruptible GC
  PMR_GC($NbPages_{Evict}$*)*
else
  Wait for next PMR write request
end if
FUNCTION 5: PMR_GC($NbPages_{Evict}$)
while ($NbPages_{Evict} > 0$)
  for i ← 1 **to** *sizeOfPMRinBlocks*
    Find $Block_{MaxInvalid}$
  end for
  if (no $Block_{MaxInvalid}$ found) //no invalid pages in PMR
    BMR_GC()
  else
    Find $NbPages_{Invalid}$
    Read valid pages from $Block_{MaxInvalid}$
    Copy $Block_{MaxInvalid}$ → freePMRBlock
    Update page-mapping table
    Erase $Block_{MaxInvalid}$
  end if
  $NbPages_{Evict} = NbPages_{Evict} - NbPages_{Invalid}$
end while
FUNCTION 6: BMR_GC()
for i ← 1 **to** *sizeOfPMRinBlocks*
Find $MaxGroupPages_{PMR}$
Read all pages $MaxGroupPages_{PMR}$ → specific block in the cache
Read valid pages of the same block as $MaxGroupPages_{PMR}$ *from BMR*
Invalidate data from the PMR and BMR mapping tables
Erase the data block in the BMR
Flush the whole block of $MaxGroupPages_{PMR}$ from the cache and update the mapping table

4.3.3. PMR-GC and BMR-GC Asynchronous Design

In CACH-FTL, both GCs can be launched synchronously or asynchronously. PMR-GC is launched synchronously whenever there is not enough free space to write the flushed pages (from the cache), while BMR-GC is launched synchronously whenever PMR-GC is not sufficient to recycle enough free space to write the flushed pages.

In order to benefit from the I/O idle times, GCs can be launched asynchronously in order to recycle blocks, to generate free space and, thus, to anticipate page reclamation in PMR. This allows one to reduce the I/O request mean response times.

PMR-GC is launched asynchronously whenever there is less than a predefined percentage of free space in the PMR (Line 13, Algorithm 2). This percentage threshold was fixed at 10% for the performance evaluation part of the present study. On the other hand, PMR-GC is launched synchronously whenever the free space falls under a minimal predefined threshold (fixed to three blocks in the performance evaluation part) and when the FTL needs to free some space on the PMR (in order to perform a write operation: Line 33–35, Algorithm 1). When the PMR-GC is launched synchronously, it naturally delays the response time of the current write request.

BMR-GC is launched asynchronously whenever it is triggered by an asynchronous PMR-GC (Line 25, Algorithm 2); otherwise, it is launched synchronously.

In order to allow the design of asynchronous GCs, one has to define interruption points in the GC process. Indeed, both asynchronous GC systems (PMR and BMR GC) are interruptible, which is not the case for synchronous GC operations. PMR-GC is performed in three phases (Lines 28–30, Algorithm 2) and is interruptible at the end of each phase. Each phase is atomic to avoid system inconsistency. If the system performs the action on Line 28, the valid pages are read and put into a dedicated block in the cache and maintained there until the PMR-GC is complete, even if interrupted. Line 29 consists of copying the pages of the candidate block to a free PMR block. This is transparent to the functioning of CACH-FTL, as the mapping table is changed only at Line 30.

BMR-GC is performed in five phases (Lines 38–42, Algorithm 2) and can also be interrupted at the end of each phase. As for the asynchronous PMR-GC, the first step consists of reading the data and putting them into the cache (Line 38). In Line 39, the valid pages from the BMR are read in order to have the whole valid date on that block in the cache. Those two steps are interruptible, as data are still consistent in the PMR and BMR. In Line 40, data are invalidated from the PMR and BMR, so that they can be read from the cache. In Line 41, the block is erased, and finally, in Line 42, data are flushed from the dedicated cache block to the BMR in an atomic step.

The two GC algorithms described in this section are specific to over-provisioning space management and do not concern traditional FTL GCs performed on a data space (in our case, in the BMR), due to wear-leveling implementation. Once again, direct mapping is used in the BMR, which operates without a GC. If ever out-of-place modifications are allowed in the BMR (which is not currently the case), an additional GC algorithm would be necessary to recycle invalid pages. In that case, many state-of-the-art GC systems can be used.

5. Performance Evaluation

The first step of the performance evaluation of CACH-FTL consisted of comparing it with three other FTL schemes: (1) page-mapping (PM); (2) an optimized block-mapping scheme (BM); and (3) FAST hybrid FTL. The same cache system (C-lash in our case) with the same configuration was placed on top of each of the tested FTLs.

In the first step, the CACH-FTL configuration was fixed during all the tests. Cache on pure PM represents the ideal performance one can obtain, since PM is the optimal FTL [22]. The drawback of such a scheme is that it is definitely non-scalable, as the PM table size would become impractically large when the flash memory size becomes high. Cache on BM represents the lower bound of performance that CACH-FTL should achieve. The simple BM scheme gives very bad performance, but uses a very small mapping table. The BM used in this part was optimized, so that when a group of sequential pages was flushed from the cache, writing this group required only one erase operation (rather than as many block erasures as the number of pages flushed). This is a very important optimization in BM, as it reduces drastically the mean response times. FAST, a good performing, very popular log-block hybrid FTL, was also compared. To summarize, CACH-FTL was compared with an ideal, a lower bound and a very popular and good performing FTL.

This first experimental step ends up with some figures comparing the best CACH-FTL configuration with the other FTLs. The purpose was to show that one can always find a configuration to apply for CACH-FTL, giving very good performance even when compared with PM and better than BM and FAST. The objective of CACH-FTL is to approach PM performance, while using a smaller amount of embedded RAM.

The second step consisted of evaluating the impact of tuning both the redirection threshold and the over-provisioning space (PMR) size according to the I/O workload. The purpose of this part was to investigate CACH-FTL's flexibility in comparison with standard hybrid FTLs. This helps to find the optimal configuration points according to the I/O workload characteristics.

5.1. Storage System and Performance Metrics

A modified version of the FlashSim [27] simulator was used, based on the DiskSim [28] discrete event simulator, the most popular disk-based storage simulator.

Two main performance metrics were considered: the mean response time and the number of erase operations. Response times were captured at the I/O driver level, including all intermediate delays: caches, controllers, I/O queues, *etc.* The impact of intermediate elements was, however, minimized to focus on the flash memory subsystem behavior. The second metric is the number of erase operations, which gives an idea of the global wear out, even though it does not provide information on the wear leveling (as no wear leveler is implemented).

The simulated NAND flash memory characteristics were as follows: a 4 KB page size and a 256 KB block size. The three basic operations had the following delays: 25 µs for a page read, 200 µs for a page write and 1.5 ms for a block erasure. Those numbers are related to a real flash storage system [29]. The chosen cache size in all the tests was 2 MB (the simulated flash memory address spaces were

less than 8 GB for real traces). For this study, the cache system was configured and fixed to six blocks in the b-space and 128 pages in the p-space.

A fixed default configuration for CACH-FTL was used for the first set of tests. It had the following characteristics: a redirection threshold of four pages and an over-provisioning space (PMR) representing 10% of the flash space (the configuration can be optimized according to the workload). Similarly, FAST was set to use 10% for over-provisioning space for log-blocks (PM and BM do not use over-provisioning space).

Each simulation trace was preceded by a warm up phase consisting of playing the simulated trace twice in order to have a representative flash memory (in terms of valid, invalid and clean pages) and cache state. From our tests, we noticed that playing the I/O traces twice during warm up is enough, as the performance does slightly change when passing from two to ten trace executions for the warm-up (less than 2% performance difference).

5.2. Simulated I/O Workloads

Both real and synthetically generated I/O workloads were considered. For real traces, some widely used I/O traces available from the Storage Performance Council (SPC) [30] were chosen. These workloads describe traces of Online Transaction Processing (OLTP) applications obtained from two financial institutions [31]. Another tested trace subset was Cello99 [32], which is issued from the activity of a workgroup file server used in HP labs. For Cello99, eight-day traces beginning on 20 February 1999, were selected from five different disks (see Table 1).

Table 1 gives the mean write rate of all the disks of the financial trace and the minimum and maximum write rate in the set of disks per trace. As for the sequentiality rate, both the inter-request rate and the per-page (4 KB) sequentiality are shown, the latter being more representative, as it takes into account both inter- and intra-request sequentiality.

Table 1. Synthetic I/O trace characteristics (the first line contains the varied metrics).

Sequential Rate	**Request Number**	**Inter-Arrival Times**
40%, default value (10% → 90%)	250,000 default value (10,000 → 5,000,000)	exponential (0, 200 ms) default value (50 → 500 ms)
Spatial Locality	**Write Rate**	**Mean Request Size**
20%	100%	1 page (4 KB)

Table 2 presents the synthetic I/O workload characteristics. For sequentiality, spatial locality and inter-arrival time rates, the observed financial (Financials 1 and 2) trace average rates were considered as default values. Then, we launched simulation changing one parameter at a time. The sequentiality rate, request number and inter-arrival times were varied on a 1-GB size flash memory. A small size was chosen to rapidly saturate the flash memory without affecting performance compared with larger sizes. This ensures the launch of the GCs, without which the performance would not be representative.

Table 2. Financial 1, Financial 2 and Cello99 I/O trace characteristics.

Number format: Mean, (Min, Max)	Financial 1 (24 Volumes)	Financial 2 (19 Volumes)	Cello99 (5 Volumes)
Write rate	77%, (4%, 100%)	18%, (0%, 98%)	34%, (19%, 53%)
Sequentiality per request/per page	23%/46% (1%, 99%)	9%/38% (3%, 96%)	9%/45% (4%, 27%)
Mean request. size (KB)	5.6	5.3	4.5
Trace time (h)	12	12	168

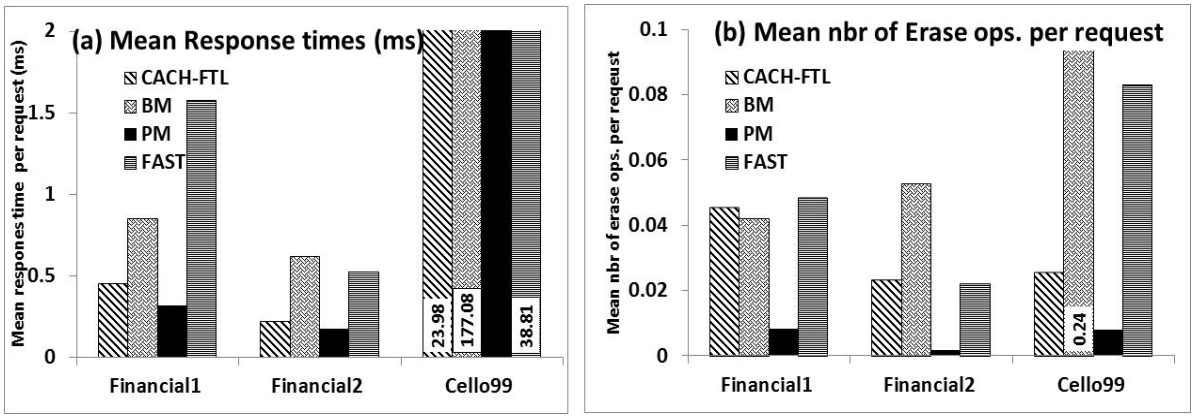
6. Results and Discussion

This section describes the results of the tests conducted.

6.1. CACH-FTL versus PM, BM and FAST

Figure 4a shows the mean response time per request for CACH-FTL compared with BM, PM and FAST. One can observe that for the chosen configuration, CACH-FTL approaches the performance of the ideal PM better in most cases and always performs better than BM and FAST. For the Financial 1 traces, CACH-FTL improves BM by 47% and FAST by 71%. For Financial 2, CACH-FTL improves BM by 65% and FAST by 58%. Finally, for the Cello99 trace, CACH-FTL improves BM by 86% and FAST by 38%. The average differences between the ideal PM and CACH-FTL are: 39% for Financial 1, 21% for Financial 2 and 35% for Cello99. Thus, CACH-FTL drastically improves response times and is closer to the ideal PM case.

Figure 4. Performance of CACH-FTL. (**a**) The mean response times for the studied traces; (**b**) The number of generated erase operations. Note that some results are outside the scale of the graph, values are reported on the figure.



One can see that the tested CACH-FTL configuration generates 8% more erasures than BM for Financial 1 and 6% less than FAST. For the Financial 2 trace, CACH-FTL performs 55% better than BM and 5% less then FAST. Finally, for the Cello workload, CACH-FTL performs 89% better than BM

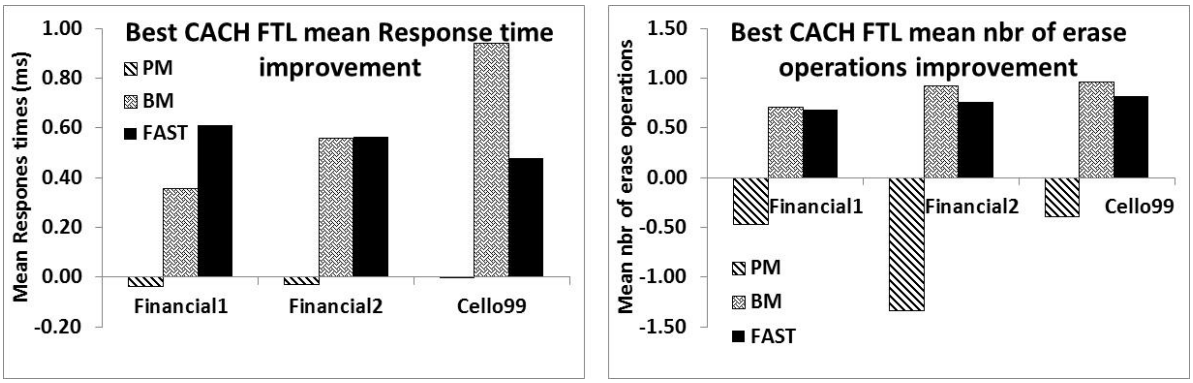
and 69% better than FAST. For all erase operation results, one can observe that PM performs, by far, better than the other FTLs.

As compared to BM, CACH-FTL performs better in terms of response times for 80% of the tested disks of the Cello trace, 79.2% of the Financial 1 trace and for 94.7% of the Financial 2 trace. For the number of erase operations metric, CACH-FTL enhances BM for 80% of the Cello disks, 91.7% of the Financial 1 disks and 89.5% of the Financial 2 disks.

As compared to FAST, CACH-FTL performs better in terms of response times for all the tested disks of the Cello trace, 87.5% of the Financial 1 trace and for 84.2% of the Financial 2 trace. For the number of erase operations metric, CACH-FTL improves BM for all the Cello disks, 66.7% of the Financial 1 disks and 68.4% of the Financial 2 disks.

Figure 5 presents the best results that CACH-FTL can give when varying the redirection threshold from 1 to 32 and the over-provisioning space from 5% to 25% of the total flash memory space. Those two parameters were varied for each single disk of each trace, and the best configuration was chosen for each disk. For FAST FTL, the log-block space has been varied accordingly. The main observation one can draw from the mean response times figure is that CACH-FTL performs approximately as good as PM, the ideal FTL, with a difference of 4%, 3% and zero for Financial 1, Financial 2 and Cello, respectively. CACH-FTL improves the performance of BM by 36%, 56% and 94% for Financial 1, Financial 2 and Cello, respectively. Finally, CACH-FTL enhances the performance of FAST by a factor of 61%, 56% and 48% for Financial 1, Financial 2 and Cello, respectively.

Figure 5. CACH-FTL's best configuration improvement over BM and FAST.



For the mean number of erase operations improvement, one can observe that the best CACH-FTL configuration always performs better than BM and FAST by more than 69% (less erase operations are generated). One can infer that this would prevent the flash memory from wearing out quickly, as the total number of erase operations is drastically reduced. However, compared to PM, CACH-FTL performs very poorly, as it generates much more erase operations. This is mainly due to the additional number of erase operations generated, because of the garbage collection mechanism. As this mechanism is asynchronous in CACH-FTL, it does not always impact response times. One must also keep in mind that PM uses approximately 80% more memory for storing the mapping table than CACH-FTL, with a 25% over-provisioning space.

Figure 6 presents the results of the tests with the synthetic I/O workloads described in Table 2. In Figure 6a, one can observe the mean response times according to the sequentiality rate variation. It can

be noticed that the more sequential the workload, the closer CACH-FTL gets to PM (only a 4% difference for a sequential rate of 90%); it generates less garbage collections, as sequential writes are directed towards BMR. However, even for very random workloads, CACH-FTL is far better than BM and FAST (a 41% and 64% improvement factor, respectively). FAST gives very poor performance for random workloads, due to the merge operation overhead. Indeed, the number of costly merge operations triggered is equal to the number of pages coming from different data blocks (at random) in the victim log-block [19].

Figure 6. Performance of CACH-FTL on synthetic workloads. **(a)** Sequentiality rate variation. **(b)** Number of generated writes (on a fixed flash size) variations. **(c)** Inter-arrival time variation, the values represent the mean of an exponential distribution.

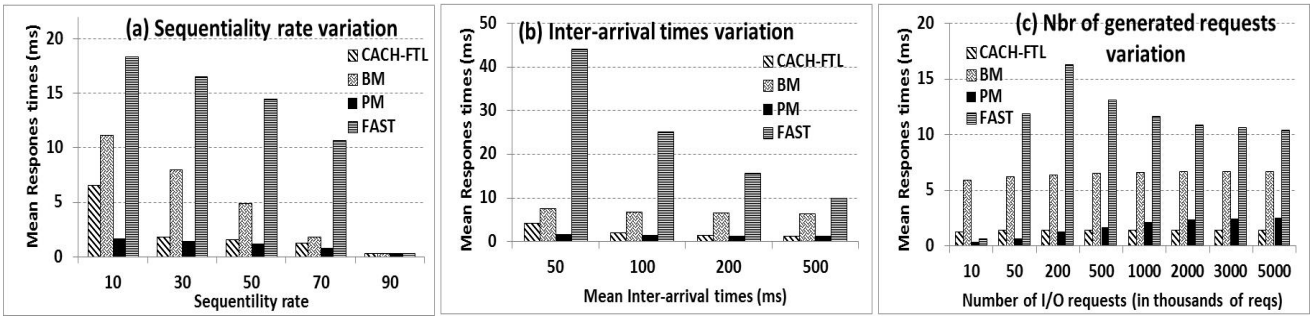


Figure 6b shows the results when inter-arrival times are varied; this variation does not impact the performance of BM and PM. For FAST, short inter-arrival times lead to saturated I/O queues, delaying response times. However, for CACH-FTL, higher inter-arrival times allow GC times to be better absorbed, leading to better performance. CACH-FTL performs better than FAST and BM for all cases and approaches PM with a 15% difference for higher inter-arrival times.

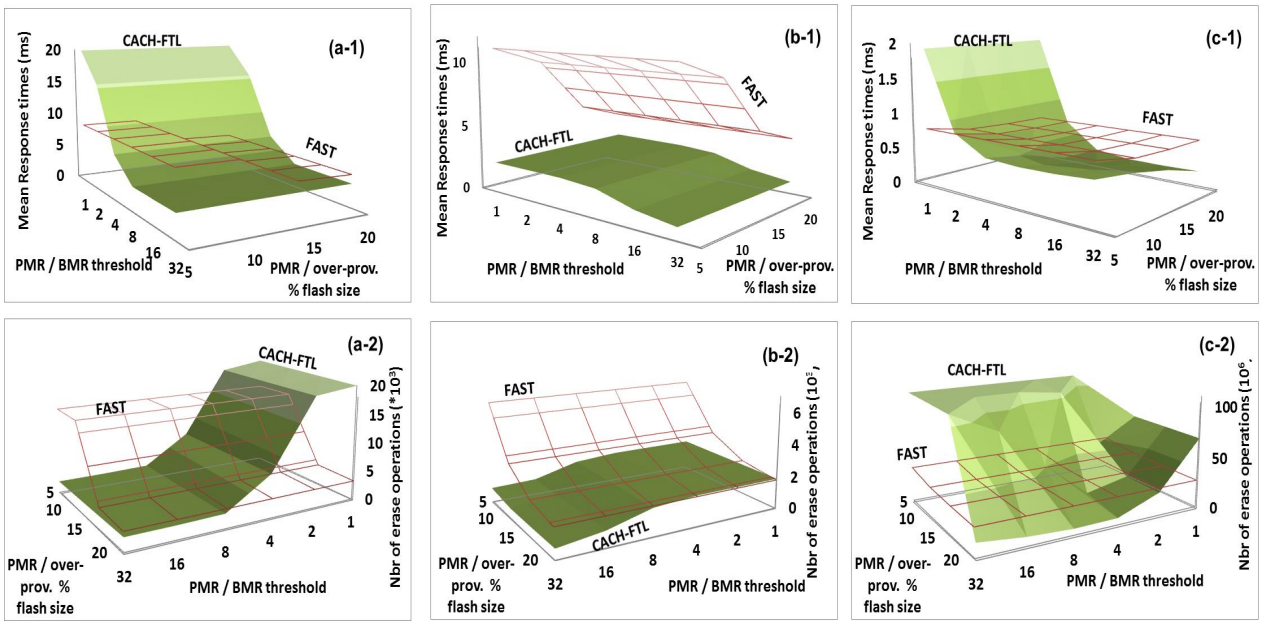
Figure 6c shows the average response time variation according to the number of generated writes. Increasing the number of writes tends to saturate the flash memory and forces the FTL to perform several garbage collections. In fact, the simulated flash space is 1 GB, and the largest set of generated writes (five million) covers 19 GB. CACH-FTL performs even better than PM (for request numbers greater than 500,000), as the latter is saturated with garbage collections, while CACH-FTL can better handle them, thanks to the asynchronous implementation.

6.2. CACH-FTL Adaptability: Redirection Threshold and Over-Provisioning Space Configuration

Figure 7 depicts the I/O performance variation of CACH-FTL with different configuration points for both the redirection threshold and the PMR size compared to FAST for three sample disks (one from Cello99, one from Financial 1 and one from Financial 2). For nearly all the tested disk volumes (around 50 for the real I/O traces), simulations showed that there is always at least one CACH-FTL configuration that surpasses, or at least matches, both FAST and BM for a given over-provisioning space size. Figure 7(a-1) and 7(c-1) shows that even though CACH-FTL gives very bad performance when poorly dimensioned (very small values of the BMR/PMR redirection threshold), it is capable of optimal performance. In Figure 7(a-1), the optimal configuration of CACH-FTL is given for high values of the redirection threshold and improves FAST performance by 27%. In Figure 7 (c-1), the best CACH-FTL

performance is also reached by high values of the threshold (32) and improves it by 68%, and by 90% for Figure 7 (b-1). Optimal performance depends highly on the applied workload, as can be observed in Figure 7. The same conclusions can be drawn based on the number of erase operation curves in Figure 7 (a-2),(b-2),(c-2). The optimal points are always given by CACH-FTL for all the tested disks. Note that the best performances are not always achieved at high threshold values and high over-provisioning space.

Figure 7. CACH-FTL and FAST performance comparison on three different volumes of Financial and Cello real traces. We varied the over-provisioning space for FAST and the PMR region and threshold for CACH-FTL. The curves show one volume from the Cello workload (**a**) and two volumes from the financial traces (**b**, **c**). Both response times and the number of erase operations are illustrated.



CACH-FTL offers the user a large design space, providing flexibility that allows one to compromise between the performance and lifetime of the flash memory (the number of erase operations performed).

7. Conclusion and Future Work

This paper presents a cache-aware configurable hybrid FTL, named CACH-FTL. CACH-FTL manages groups of pages flushed from the upstream cache according to their size (in terms of the number of pages). Large groups are flushed into the data block-mapped data region, while small groups are buffered within an over-provisioning page-mapped space and moved to the BMR data space asynchronously during I/O idle times.

The CACH-FTL's design came from the need to better use storage architecture by exploiting some information coming from the cache to manage the hybrid mapping in a simple and efficient way. It also partly bridges the information gap between the cache and the FTL.

Some characteristics of CACH-FTL are: (1) genericity, as it can be used with any flash-specific cache system, provided that it flushes groups of pages; (2) flexibility, as it offers a large configuration space, allowing the efficient tuning of I/O performance (response time and lifetime) according to application needs; this is achieved by modifying the PMR size and the redirection threshold; (3) scalability, as CACH-FTL can work with whatever cache memory/flash storage ratio, as one can adjust the size of the mapping table accordingly (reducing the PMR size); (4) efficiency, as CACH-FTL uses much less RAM than PM to store the mapping table: with four to five times less memory usage for a PMR, occupying 25% of the flash memory space, while having a drop in performance of less than 5% when the configuration is well chosen.

One possible limitation of CACH-FTL is the additional erase operations generated (due to PMR and BMR GCs), which can result from a bad configuration (threshold and PMR size) or a change in the workload characteristics. To solve this issue, we plan to implement an adaptive version of CACH-FTL, where the over-provisioning space and/or the redirection threshold can be tuned dynamically according to the workload characteristics or the flash memory state. Another perspective is the integration of a state-of-the-art wear leveler and garbage collection into CACH-FTL, as this functionality was beyond the scope of this study. The wear leveler can be implemented in each region (PMR and BMR) and can also be used to balance the wear between the regions when too many erasures are provoked in a particular space.

Conflicts of Interest

The authors declare no conflict of interest.

References

1. Market Research. Available online: http://www.marketresearch.com/corporate/aboutus/ press.asp?view=3&article=2223 (accessed on 19 March 2014).
2. Ranganathan, P. From microprocessors to nanostores: Rethinking data-centric systems. *Computer* **2011**, *44*, 39–48.
3. Ban, A. Flash File System. US Patent No 5,404,485, 4 April 1995.
4. Jo, H.; Kang, J.; Park, S.; Kim, J.; Lee, J. FAB: Flash-aware buffer management policy for portable media players. *IEEE Trans. Consum. Electron.* **2006**, *52*, 485–493.
5. Kang, S.; Park, S.; Jung, H.; Shim, H.; Cha, J. Performance trade-offs in using NVRAM write buffer for flash memory-based storage devices. *IEEE Trans. Comput.* **2009**, *58*, 744–758.
6. Kim, H.; Ahn, S. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST), San Jose, CA, USA, 26–29 February 2008.
7. Wu, G.; Eckart, B.; He, X. BPAC: An Adaptive Write Buffer Management Scheme for Flash-Based Solid State Drives. In Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST), Incline Village, NV, USA, 3–7 May 2010.
8. Debnath, B.; Subramanya, S.; Du, D.; Lilja, D.J. Large Block CLOCK (LB-CLOCK): A Write Caching Algorithm for Solid State Disks. In Proceedings of the IEEE International Symposium on

Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), London, UK, 21–23 September 2009.

9. Hu, J.; Jiang, H.; Tian, T.; Xu, L. PUD-LRU: An Erase-Efficient Write Buffer Management Algorithm for Flash Memory SSD. In Proceedings of the IEEE International Symposium on of Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), Miami Beach, FL, USA, 17–19 August 2010.
10. Seo, D.; Shin, D. Recently-evicted-first buffer replacement policy for flash storage devices. *IEEE Trans. Consum. Electron.* **2008**, *54*, 1228–1235.
11. Boukhobza, J.; Olivier, P.; Rubini, R. A Cache Management Strategy to Replace Wear Leveling Techniques for Embedded Flash Memory. In Proceedings the of International Symposium on Performance Evaluation of Computer & Telecommunication Systems (SPECTS), The Hague, The Netherlands, 27–30 June 2011.
12. Boukhobza, J.; Olivier, P.; Rubini, S. CACH-FTL: A Cache-Aware Configurable Hybrid Flash Translation Layer. In Proceedings of Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 27 February–1 March 2013.
13. Brewer, J.E.; Gill, M. *Nonvolatile Memory Technologies with Emphasis on Flash*; IEEE Press Series, Wiley Inter-Science: Piscataway, NJ, USA, 2008; pp. 22–24.
14. Ban, A. Flash File System Optimized for Page-Mode Flash Technologies. US Patent No 5,937,425, 10 August 1999.
15. Wu, C.; Kuo, T. An Adaptive Two-Level Management for the Flash Translation Layer in Embedded Systems. In Proceedings of the IEEE/ACM international conference on Computer-Aided Design (ICCAD), San Jose, CA, USA, 5–9 November 2006.
16. Hsieh, J.; Tsai, Y.; Kuo, T.; Lee, T. Configurable flash-memory management: Performance *versus* overheads. *IEEE Trans. Comput*. **2008**, *57*, 1571–1583.
17. Kim, J.; Kim, J.M.; Noh, S.H.; Min, S.L.; Cho, Y. A space-efficient flash translation layer for compactflash systems. *IEEE Trans. Consum. Electron*. **2002**, *48*, 366–375.
18. Wang, Y.; Liu, D.; Wang, M.; Qin, Z.; Guan, Y. RNFTL: A Reuse-Aware NAND Flash Translation Layer for Flash Memory. In Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), Stockholm, Sweden, 13–15 April 2010.
19. Lee, S.; Park, D.; Chung, T.; Lee, D.; Park, S.; Song, H. A log buffer based flash translation layer using fully associative sector translation. *ACM Trans. Embed. Comput. Syst*. **2007**, *6*, 1–27.
20. Cho, H.; Shin, D.; Eom, Y.I. KAST: K-Associative Sector Translation for NAND Flash Memory in Real-Time Systems. In Proceedings of Design, Automation and Test in Europe (DATE), Nice, France, 20–24 April 2009.
21. Lee, S.; Shin, D.; Kim, Y.; Kim, J. LAST: Locality aware sector translation for NAND flash memory based storage systems. *ACM SIGOPS Oper. Syst. Rev.* **2008**, *42*, 36–42.
22. Lee, H.; Yun, H.; Lee, D. HFTL: Hybrid flash translation layer based on hot data identification for flash memory. *IEEE Trans. Consum. Electron.* **2009**, *55*, 2005–2011.
23. Guan, Y.; Wang, G.; Wang, Y.; Chen, R.; Shao, Z. Block-Level Log-Block Management for NAND Flash Memory Storage Systems. In Proceedings of the ACM SIGPLAN/SIGBED Conference on

Languages, Compilers and Tools for Embedded Systems (LCTES), Seattle, WA, USA, 20–21 June 2013.
24. Wei, Q.; Gong, B.; Pathak, S.; Veeravalli, B.; Zeng, L.; Okada, K. WAFTL: A Workload Adaptive Flash Translation Layer with Data Partition. In Proceedings of the 27th IEEE Symposium on Mass Storage Systems and Technologies (MSST), Denver, CO, USA, 23–27 May 2011.
25. Park, D.; Debnath, B.; Du, D. A Workload-Aware Adaptive Hybrid Flash Translation Layer with an Efficient Caching Strategy. In Proceedings of the IEEE International Symposium on of Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 25–27 July 2011.
26. Liao, X.; Hu, S. Bridging the information gap between buffer and flash translation layer for flash memory. *IEEE Trans. Consum. Electron.* **2011**, *57*, 1765–1773.
27. Kim, Y.; Taurus, B.; Gupta, A.; Urgaonkar, B. FlashSim: A Simulator for NAND Flash-Based Solid-State Drives. In Proceedings of the 1st International Conference on Advances in System Simulation (SIMUL), Porto, Portugal, 20–25 September 2009.
28. Ganger, G.R.; Worthington, B.; Patt, Y.N. *The Disksim Simulation Environment Version 3.0 Reference Manual*; Tech. Report CMU-CS-03-102; Carnegie Melon University, Pittsburgh, PA, USA, 2003.
29. Agrawal, N.; Prabhakaran, V.; Wobber, T.; Davis, J.D.; Manasse, M.; Panigrahy, R. Design Tradeoffs for SSD Performance. In Proceedings of the USENIX Annual Technical Conference (ATC), Oslo, Norway, 23–25 June 2008.
30. Storage Performance Council Website. Available online: http://www.storageperformance.org/home/ (accessed on 22 January 2014).
31. OLTP Traces—UMass Trace Repository. Available online: http://traces.cs.umass.edu/index.php/Storage/Storage/ (accessed on 22 January 2014).
32. Cello99 Traces, HP Labs. Available online: http://tesla.hpl.hp.com/opensource/cello99 (accessed on 22 January 2014).