Towards an Efficient Flash-Based Mid-Tier Cache

Yi Ou[1], Jianliang Xu[2], and Theo Härder[1]

[1] University of Kaiserslautern
{ou,haerder}@cs.uni-kl.de
[2] Hong Kong Baptist University
xujl@comp.hkbu.edu.hk

Abstract. Due to high access performance and price-per-byte considerations, flash memory has been recommended for use as a mid-tier cache in a multi-tier storage system. However, previous studies related to flash-based mid-tier caching only considered the indirect use of flash memory via a flash translation layer, which causes expensive flash-based cache maintenance. This paper identifies the weaknesses of such indirect methods, with a focus on the *cold-page migration* problem. As improvements, we propose two novel approaches, an indirect approach called LPD (logical page drop) and a native approach called NFA (native flash access). The basic idea is to drop cold pages proactively so that the garbage collection overhead can be minimized. Our experiments demonstrate that both approaches, especially the native one, effectively improve the use of flash memory in the mid-tier cache. NFA reduces the number of garbage collections and block erasures by up to a factor of five and improves the mid-tier throughput by up to 66%.

1 Introduction

Despite fast page-oriented random access, storage devices based on flash memory (flash-based devices), e. g., flash SSDs, are still too expensive to be the prevalent mass storage solution. In fact, flash memory perfectly bridges the gap between RAM and magnetic disks (HDDs) in terms of price per capacity and performance[3]. Therefore, using them in the middle tier of a *three-tier storage hierarchy* is a much more realistic approach. In such a hierarchy, the top tier incorporates fast but expensive RAM-based buffer pools, while the bottom tier is based on slow but cheap storage devices such as HDDs or even low-end flash SSDs. The middle tier acts as a cache larger but slower than the top-tier buffer pool. Such a three-tier storage system can be deployed as, e. g., the storage sub-system of a database (DB) system. Nevertheless, flash memory has some distinguishing characteristics and limitations that make its efficient use technically challenging. This paper studies its efficient use for mid-tier caching.

[3] We focus on NAND flash memory due to its suitability for storage systems.

1.1 Flash memory and FTL

Flash memory supports three basic operations: *read*, *program*, and *erase*. Read and program (also known as write) operations must be performed in units of *flash pages*, while erase operations have to be done at a larger granularity called *flash block (block)*, which contains a multiple of (e. g., 128) flash pages. Read operations have a very small latency (in the sub-millisecond range). However, program operations are much slower than them, typically by an order of magnitude. Erase operations are even slower than program operations, by another order of magnitude. Let C_{fr}, C_{fp}, and C_{fe} be the costs of read, program, and erase operations, respectively, we have: $C_{fr} < C_{fp} < C_{fe}$.

An erase operation turns a block into a *free block*, and, consequently, each of its flash pages into a *free flash page*, which is a flash page that has never been programed after the block erasure. Only free flash pages can be programed, i. e., a non-free flash page can become programmable again – but only after an erase of the entire block. This limitation, known as *erase-before-write*, implies that an *in-place update* of flash pages would be very expensive [1]. Furthermore, after enduring a limited number of program/erase (P/E) cycles[4], a block becomes highly susceptible to bit errors, i. e., it becomes a bad block.

Mainly due to the aforementioned limitations, flash memory is often accessed via an intermediate layer called *flash translation layer (FTL)*, which supports *logical* read and write operations, i. e., reading and writing of *logical pages*. A flash page normally consists of a *main area* of several KBs for storing user data, and a *spare area* of a few bytes to facilitate the FTL implementation. For brevity, we assume that (the size of) a logical page corresponds to a bottom-tier page (e. g., a DB page) and they are of the same size of the main area, and we use the term *page* to refer to a *logical page*, which is to be distinguished from a *physical page*, i. e., a flash page.

To avoid the expensive in-place updates, FTL follows an *out-of-place update* scheme, i. e., each logical page update is served using a free flash page prepared in advance. Consequently, multiple writes to a logical page can result in multiple page versions co-existing in flash memory. The term *valid page* refers to the latest version, while *invalid pages* refer to the older versions. Similarly, a *valid flash page* is the physical page where the latest version resides. When free flash pages are in short supply, some space taken by invalid pages has to be reclaimed. This is done by a procedure called *garbage collection (GC)*, which reduces the number of invalid pages and increases the number of free flash pages.

To keep track of the valid flash page of a *(logical) page*, an address mapping $m_{\mathrm{FTL}} : A_F \mapsto A_f$ is maintained, where A_F represents the set of FTL logical addresses (FLAs), i. e., logical page numbers supported by the FTL, and A_f the set of FTL physical addresses (FPAs), i. e., flash page addresses available on the device. Depending on the map-entry granule of the m_{FTL} implementation, FTL algorithms can be classified into three categories: page-level mapping [2,3], block-level mapping [4,5], and hybrid mapping [6,7]. Among them, page-level mapping

[4] The number of cycles depends on density, vendor, and flash memory type. SLC NAND flash memory is typically rated for ∼100,000 P/E cycles.

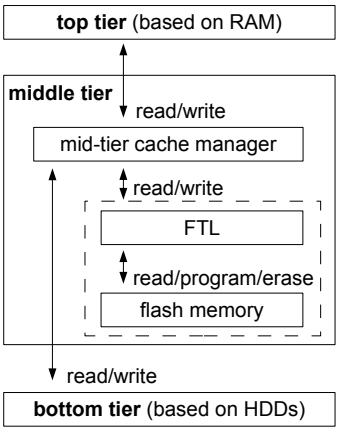


Fig. 1: Three-tier storage system with indirect use of flash memory by the mid-tier cache. FTL makes the native interface of the flash memory transparent to the mid-tier cache manager. The two components of a flash-based device, surrounded by the dashed line, appear as a "black box" and act together as a persistent array of logical pages that can be read and overwritten.

has the greatest performance potential, but it also has the highest resource requirements, mainly due to the mapping table size. However, recent studies have shown that the resource problem of page-level mapping can be effectively dealt with using methods such as demand paging of the mapping table [8] or new hardware such as PCM (phase-change memory) as the mapping table storage media [9]. Therefore, in this paper, we focus on page-level mapping in favor of the performance potential, although the problems studied and the basic ideas leading to our solutions are not specific to any FTL implementation.

1.2 Problem

Previous studies on flash-based mid-tier caching only considered the *indirect* use of flash memory, i. e., the use of flash memory via an FTL, as shown in Fig. 1. Although simplifying the use of flash memory, the indirect approach has some fundamental problems. FTL implementations are usually vendor-specific and proprietary [1,10]. The proprietary FTL logic makes it impossible to accurately model or predict the performance of flash-based devices. This is not acceptable for performance-critical applications, because their optimization is often based on the cost model of the underlying storage devices. Furthermore, without direct control over potentially expensive procedures such as GC, the response time becomes indeterministic for the application. It has been reported that GC can take up to 40 seconds[11], which is not only an issue for applications with real-time requirements, but also intolerable for normal use cases.

For flash-based mid-tier caching, the indirect approach has an even more serious problem related to GC. This problem is explained in the following with the help of a simplified GC procedure, which involves three steps:

1. Select a set of *garbage blocks*, which are blocks containing some invalid pages.
2. Move all valid pages from the garbage blocks to another set of (typically free) blocks, and update the corresponding management information.
3. Erase the garbage blocks, which then become free blocks.

If a block has M pages and Step 1 selects only one garbage block, which has v valid pages, then Step 2 consumes v free flash pages, and the procedure increases the total number of free flash pages by $M - v$, at a total cost of $(C_{fr} + C_{fp}) \times v + C_{fe}$, where $(C_{fr} + C_{fp}) \times v$ is caused by Step 2, and C_{fe} caused by Step 3. The ratio v/M is called *block utilization*. Obviously, GC is more effective and also more efficient for smaller values of v/M, because more free flash pages are gained at a lower cost. Therefore, v/M is an important criterion to be considered for the garbage block selection in Step 1. If the entire flash memory is *highly utilized*, i. e., v/M is statistically close to 1, GC becomes relatively expensive, ineffective, and has to be invoked frequently.

Although for a cache, only hot pages should be kept and cold pages should be evicted, FTL must guarantee each valid page is accessible no matter the page is cold or hot. This means that, during GC processing, cold pages have to be moved along with hot pages (Step 2), while the cold ones, which make v/M unnecessarily high, could actually be discarded from the cache manager perspective. We call this problem the *cold-page migration (CPM)* problem.

More specifically, the CPM problem negatively impacts mid-tier performance in two aspects: 1. The cost of GC, due to the (unnecessary) CPM; 2. The frequency of GC, because, if cold pages are regarded valid, fewer pages can be freed by one invocation of GC, and, as a result, the subsequent GCs have to be invoked earlier. Furthermore, the GC frequency is proportional to the number of block erases, which is inversely proportional to the device life time due to the endurance limitation.

A similar problem exists when flash SSDs are used as the external storage under a file system. File deletion is a frequent operation, but the information about deleted files is normally kept in OS and not available to the SSD. The latter has to keep even the deleted data valid, at a potentially high operational cost. As solution, a *Trim* attribute for the Data Set Management command has been recently proposed and become available in the ATA8-ACS-2 specification [12]. This attribute enables disk drives to be informed about deleted data so that their maintenance can be avoided.

However, no sufficient attention has been paid to the CPM problem, which actually impacts the performance in a more serious way. First, when used in the mid-tier cache, flash-based devices experience a much heavier write traffic than that of file systems, because pages are more frequently loaded into and evicted from the cache. To flash-based devices, heavy write traffic means frequent GCs. Second, the capacity utilization of a mid-tier cache is always full, which makes GC expensive and ineffective (especially for heavy write workloads). In contrast, the GC issue is less critical to file systems, because typically a large portion of their capacity is unused.

1.3 Solution

To solve the CPM problem, we develop two approaches, which share the same basic idea: drop cold pages proactively and ignore them during GCs.

1. The first approach, LPD (logical page drop), accesses flash memory *indirectly* via an *extended FTL*, which can be informed about proactively evicted cold pages, and ignore them during GCs.
2. The second approach, NFA (native flash access), manages flash memory in a *native* way, i. e., it implements the out-of-place update scheme and handles GC by the cache manager, without using an FTL.

According to our experiments, both approaches significantly outperform the normal indirect approach, by improving the GC effectiveness and reducing its frequency. For example, NFA reduces the GC frequency by a factor of five, which not only contributes to the data access performance, but also implies a greatly extended device life time. In terms of overall performance (IOPS), NFA achieves an improvement ranging from 15% to 66%, depending on the workload.

1.4 Contribution

To the best of our knowledge, our work is the first that identifies the CPM problem. Our work is also the first that considers managing flash memory natively in the mid-tier cache. Our further major contributions are:

- We propose two novel approaches for flash-based mid-tier caching: LPD and NFA, both of them effectively deal with the CPM problem.
- Our study shows that, for a flash-based mid-tier cache, our native approach significantly improves the storage system performance while reducing the resource requirements at the same time.
- More importantly, the results of our study urge the reconsideration of the architectural problem of optimally using flash memory in a DB storage system, i. e., whether it should be managed natively by the DBMS or indirectly via the proprietary FTL implementations.

1.5 Organization

The remainder of this paper is organized as follows: Section 2 discusses related works. Section 3 presents and discusses our approaches. Section 4 reports our experiments for the evaluation of both approaches. The concluding remarks are given in Section 5.

2 Related Work

Before flash memory became a prevalent, disruptive storage technology, many studies, e. g., [13,14,15,16], addressed the problem of multi-level caching in the context of client-server storage systems, where the first-level cache is located at the client side and the second-level (mid-tier) cache is based on RAM in storage server. However, these studies did not consider the specific problems of a *flash-based* mid-tier cache. Our proposals are orthogonal to and can be combined

with their approaches, because their primary goal is to reduce the disk I/O of the storage server, while our approaches primarily focus on the operational costs of the middle tier.

In one of the pioneer works on flash-aware multi-level caching [17], Koltsidas et al. studied the relationships between page sets of the top tier and the mid-tier caches, and proposed flash-specific cost models for three-tier storage systems. In contrast, a detailed three-tier storage system implementation and performance study was presented in [18]. Their empirical study has demonstrated that, for certain spectrum of applications, system performance and energy efficiency can be both improved at the same time, by reducing the amount of energy-hungry RAM-based memory in the top tier and using a much larger amount of flash memory in the middle tier.

Not only academia, but also industry has shown great interest in flash-based mid-tier caching. Canim et al. [19] proposed a temperature-aware replacement policy for managing an SSD-based middle tier, based on access statistics of disk regions. In [20], the authors studied three design alternatives of an SSD-based middle tier, which mainly differ in the way how to deal with dirty pages evicted from the first-tier, e. g., write through or write back.

Although flash-specific cost models and their difference to those of traditional storage devices have been taken into account by previous works on flash-based mid-tier caching [17,18,19,20], they commonly only consider the indirect approach, while hardly any efforts have been made to examine the internals of flash-based devices when used as a mid-tier cache. Such efforts fundamentally distinguish our work from the previous ones.

3 Our Approaches

As introduced in Section 1.3, our basic idea is to drop cold pages proactively and ignore them during GCs. A question critical to the success is to what extent valid but cold pages are dropped. Note, if we drop valid pages too greedily, the benefit will not be covered by the cost of increased accesses to the bottom tier.

Which pages are cold and can be dropped is the decision of the cache manager, while the decision, when and how to do GC, is typically made by the FTL – if we follow the architecture of Fig. 1. Therefore, another important question is how to bring these two pieces of information together.

3.1 LPD

The LPD approach is basically an indirect approach, which follows the architecture shown in Fig. 1. However, to make the basic idea working, we propose, as an extension to the FTL interface, a *delete* operation, in addition to the read and write operations. Similar to the read and write operations, the delete operation is also a logical operation. Upon such a delete request, FTL should mark the corresponding flash page invalid (and update other related management information properly) so that it can be discarded by subsequent GCs.

Algorithm 1: Allocation of a free cache slot by LPD

```
   data: parameter d, set F of free
         slots, set S of occupied slots
1  if F ≠ ∅ then
2  |  remove and return one
   |  element from F ;
3  else
4  |  cache slot v ← select and
   |  remove a victim from S ;
5  |  evict the page cached in v ;
6  |  for 0 to d and S ≠ ∅ do
7  |  |  cache slot s ← select and
   |  |  remove a victim from S ;
8  |  |  evict the page cached in s ;
9  |  |  FTL.delete(s) ;
10 |  |  add s to F ;
11 |  return v ;
```

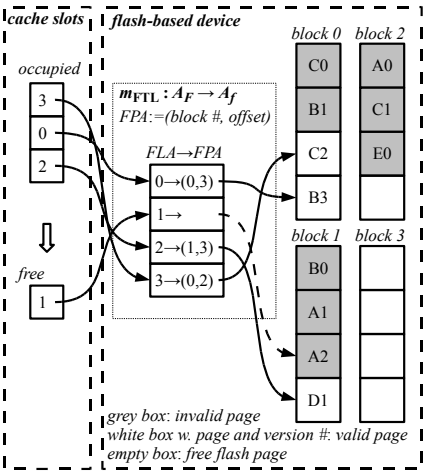


Fig. 2: Example of logical page drop. Note m_{LPD} is not shown in the figure.

LPD has some typical cache manager data structures. To tell whether and where a page is cached, it maintains an address mapping table $m_{\mathrm{LPD}} : A_b \mapsto A_F$, where A_b denotes the set of bottom-tier addresses (BTAs) and A_F the set of FLAs. A *cache slot* is a volatile data structure corresponding to exactly one FLA. In addition to the FLA, the cache slot uses one bit to represent the clean/dirty state of the cached page. A *dirty page* contains updates not yet propagated to the bottom tier. Therefore, evicting such a page involves writing it back to the bottom tier. A *free*[5] cache slot is a cache slot ready to cache a new page. Such a slot is needed when a read or write cache miss occurs, so that the missing page can be stored at the corresponding FLA. Storing the page turns a free cache slot into an *occupied* slot, which becomes free again when the page is evicted.

For the mid-tier cache manager to make use of the extended FTL, the procedure of allocating a free cache slot has to be enhanced by some additional code as shown in Algorithm 1. The piece of code (Line 6 to 10) evicts up to the d coldest pages and instructs FTL to delete them, i. e., *dropping a page* involves evicting it from the cache and deleting it logically via the extended FTL. Page dropping happens after the standard logic of cache replacement (Line 4 to 5), which is only required when there is no free cache slot available.

An example of LPD is shown in Fig. 2, where the cache slot with FLA = 1 was just dropped and became free. The corresponding flash page, although containing the latest version of page A (A2 in the figure), was invalidated (shown in grey). If later block 1 is garbage-collected, A2 can be simply discarded.

The tuning parameter d controls how greedily cold pages are dropped. When $d = 0$, LPD degenerates to the normal indirect approach without using the

[5] There is no connection between free cache slot and free flash page, although both concepts use the word “free” by convention.

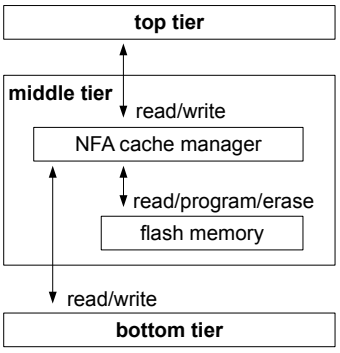


Fig. 3: NFA architecture

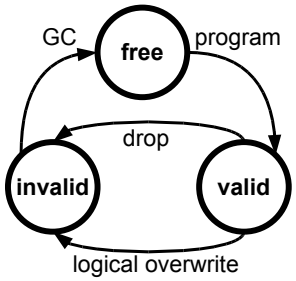


Fig. 4: NFA flash page states

extension. In contrast, when $d > 0$, the d coldest pages are dropped and the same number of cache slots are turned into free slots, ready to be used for the subsequent allocations of free cache slots (Line 1 to 2).

The LPD approach is orthogonal to the cache replacement policy responsible for the victim selection (Line 4 and Line 7), which shall identify the coldest page as per its own definition. In other words, LPD is compatible with other cache management techniques, which can be used to further improve the hit ratio.

3.2 NFA

In contrast to the indirect approaches, NFA does not require an FTL. Instead, it manages flash memory natively. As shown in Fig. 3, the operations available to the NFA cache manager are read and program of *flash pages*, and erase of *blocks*. Besides the common cache management functionality, NFA has to provide the implementation of an out-of-place update scheme and GC.

For the cache management functionality, NFA maintains a mapping table $m_{\text{NFA}} : A_b \mapsto A_f$, where A_b denotes the set of BTAs and A_f the set of FPAs. Note in LPD (and other indirect approaches), two mapping tables are required: m_{LPD} for cache management, and m_{FTL} maintained by FTL (see Section 1.1).

A volatile data structure, *block management structure (BMS)* represents the state of a block. BMS contains two bit vectors, *validity* and *cleanness*, which mark the valid/invalid and clean/dirty states for each flash page in the block. Validity is used by the GC processing, while cleanness is checked when dropping a page. Furthermore, BMS stores, for each of its valid flash pages, the corresponding BTA to speed up reverse lookups, and the corresponding last access time, which is used by the page-dropping logic. The memory consumption of BMS is very low, e. g., using 4 bytes per BTA and another 4 bytes per access time, for 8 KB pages, the memory overhead of BMS is 0.1% at maximum.

Following the out-of-place update scheme, both serving a write request and caching a (not yet cached) page consume a free flash page, which is allocated according to Algorithm 2. The algorithm maintains a write pointer wp, which always points to the next free flash page to be programed. After the program

Algorithm 2: Allocation of a free flash page by NFA

```
   data: pointer wp, set F of free
         blocks, watermarks w_l, w_h
 1 if current block is fully written
   then
 2     wp ← the first flash page of a
       free block ;
 3     if |F| ≤ w_l then
 4         while |F| < w_h do GC;
 5     return wp ;
 6 else
 7     return wp ← wp + 1 ;
```

Algorithm 3: NFA GC

```
   data: page-dropping threshold t
 1 block b ← select a garbage block ;
 2 if all pages in b are valid then
 3     b ← select a victim block ;
 4     t ← the last access time of b ;
 5 foreach page p ∈ b do
 6     if last access time of p ≤ t
       then
 7         drop(p) ;
 8     else
 9         move p to a free flash page ;
10 erase b and mark it a free block ;
```

operation, wp moves to the next free flash page in the same block, until the block is fully written – in that case, wp moves to the begin of a new free block.

Because GC is a relatively expensive procedure, it is typically processed by a separate thread. NFA uses a low watermark w_l and a high watermark w_h to control when to start and stop the GC processing. GC is triggered when the number of free blocks is below or equal to w_l, and stops when it reaches w_h, so that multiple garbage blocks can be processed in one batch. The available number of blocks and the high watermark determine the logical capacity of the cache. If we have K blocks with M pages per block, the logical capacity of the cache is: $(K - w_h) \times M$. We say that w_h blocks are *reserved* for GC processing.

Note that the out-of-place update scheme and the use of reserved blocks for GC processing shown in Algorithm 2 are common FTL techniques. They are presented here for comprehension and completeness, because they are now integral to the NFA approach.

The NFA GC procedure (shown in Algorithm 3) is similar to that of a typical FTL in some steps (Line 1, 9, and 10 roughly correspond to Step 1, 2, and 3 of the simplified GC discussed in Section 1.2). The difference is due to the dropping of *victim blocks* and cold pages. Victim blocks are selected by a *victim-selection policy* based on the temporal locality of block accesses. In contrast, garbage blocks are selected by a *garbage-selection policy*, for which block utilization is typically the most important selection criterion. Except for these basic assumptions, the NFA approach is neither dependent on any particular garbage selection policy (Line 1) nor on any particular victim selection policy (Line 3).

Dropping of a victim block happens when the selected garbage block is fully utilized, i. e., all its pages are valid. Garbage-collecting such a block would not gain any free flash page. Furthermore, such a garbage block signals that the overall flash memory utilization is full or close to full (otherwise the garbage-selection policy would return a block with lower block utilization). Therefore, instead of processing the garbage block, a *victim* block is selected by the victim-selection policy (Line 3). The last access time of the block is used to update the page-dropping threshold t. This has the effect that all pages of the victim block

are then dropped immediately (Line 6 to 7). The dynamically updated threshold t is passed on to subsequent GC invocations, where the threshold makes sure that valid pages accessed earlier than t are dropped as well.

A flash page managed by NFA has the same set of possible states (shown in Fig. 4) as those managed by a FTL: free, valid, and invalid. However, NFA has a different set of possible state transitions, e. g., a read or write page miss in an NFA cache can trigger a program operation (for storing the missing page) which changes the state of a free flash page into valid, while for FTL, serving a read request does not require a program operation. Obviously, the drop transition is not present in any FTL, either. The semantic of NFA page dropping is similar to that of LPD: the page is evicted (removing the corresponding entry from m_{NFA}, and, if the page is dirty, it is written back to the bottom tier), and then the corresponding flash page is marked invalid.

From the NFA cache manager perspective, the free flash pages for storing pages newly fetched from the bottom tier (due to page faults) are completely provided by the GC procedure in units of blocks. Therefore, NFA does not require page-level victim selection and eviction, which are common in classical caching.

3.3 Discussion

Although sharing the same basic idea, the presented approaches, LPD and NFA, are quite different from each other. While NFA directly integrates the drop logic into the GC processing, LPD can only select the drop candidates and delete them logically. LPD can not erase a block due to the indirection of FTL – the intermediate layer required by an indirect approach. Therefore, contiguously dropped logical pages may be physically scattered over the flash memory and LPD has no control when these pages will be garbage-collected, which is again the responsibility of FTL. Such dropped pages can neither contribute to the mid-tier cache hit ratio nor contribute to the reduction of GC cost, until the space taken by them is eventually reclaimed by some GC run. In contrast to the LPD approach, the pages dropped by NFA immediately become free flash pages.

To control how greedily pages are dropped, LPD depends on the parameter d, for which an optimal value is difficult to find, while the "greediness" of NFA is limited to M pages (one block at maximum). However, due to the victim selection based on block-level temporal statistics, the NFA hit ratio could be slightly compromised and a few more accesses to the bottom tier would be required.

4 Experiments

To evaluate our approaches, we implemented a three-tier storage system simulator supporting both architectures depicted in Fig. 1 and Fig. 3. The simulated flash memory and HDD modules used in our experiments were identical for both architectures.

The workloads used in our experiments originate from three buffer traces, which contain the logical page requests received by DB buffer managers under

Table 1: Size ratios of the top tier and middle tier relative to the DB size

trace	top tier	middle tier	DB size (max. page number)
TPC-C	2.216%	13.079%	451,166
TPC-H	0.951%	5.611%	1,051,590
TPC-E	0.002%	0.013%	441,138,522

the TPC-C, TPC-H, and TPC-E benchmark workloads. The TPC-C and TPC-H buffer traces were recorded by ourselves, while the TPC-E trace was provided by courtesy of IBM. Therefore, the buffer traces represent typical, strongly varying workloads to the top tier and our results are expected to be indicative for a broad spectrum of applications.

The logical page requests recorded in the buffer traces were sent to the top tier to generate the *mid-tier traces* running the experiments. The top tier, which had a buffer pool of 10,000 pages managed under an LRU replacement policy, served the requests directly from the buffer pool whenever possible. In cases of buffer faults or eviction of dirty pages, it had to read pages from and write pages to the middle tier. The sequences of read and write requests received by the middle tier were recorded and served as the mid-tier traces used in the experiments. We used them to stress the systems containing the middle and bottom tiers. As a result, the access statistics to the flash memory and HDD modules were collected for the performance study.

Three approaches were under comparison: NFA, LPD (with $d = 1024$ unless otherwise specified), and a baseline (BL), which is a mid-tier cache with indirect flash access (but without the delete extension). Our FTL implementation uses page-level mapping, which is the ideal case for the indirect approaches LPD and BL. For all three approaches, the LRU replacement policy was used for selecting victim cache pages (LPD and BL) or victim blocks (NFA), and the *greedy policy* [21,22] is used for selecting garbage blocks, which always selects the block having the least number of valid pages.

For each approach, the flash memory module was configured to have 512 blocks of 128 pages. Similar to [23] and [24], the low and high watermarks for GC were set to 5% and 10%, respectively. Due to this setting, the logical size of the mid-tier cache is 59,008 pages ($(512 - 51) \times 128$) for all approaches. In Table 1, we list the ratios of the top-tier buffer pool size and the logical size of the mid-tier cache relative to the DB size (using the maximum page number as an estimate[6]).

4.1 Overall performance

We use the throughput of the middle tier, i. e., the throughput seen by the top tier, as the overall performance metric, which is defined as: throughput $= N/t_v$,

[6] For the TPC-E trace, the DB size estimation is coarse because the trace was converted from a proprietary format addressing more than 20 DB files whose sizes and utilization were unavailable to us.

Table 2: Operation costs

operation	cost (ms)
C_{fr}	0.035
C_{fp}	0.350
C_{fe}	1.500
C_H	5.500

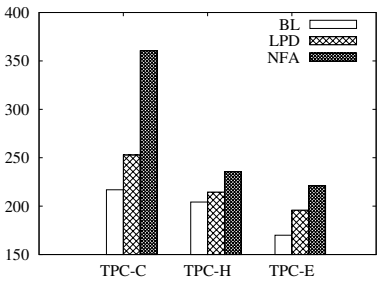


Fig. 5: Throughput (IOPS)

where N is the number of page requests in a trace and t_v its execution time, which is further defined as:

$$t_v = t_m + t_b \tag{1}$$

t_m represents the total operational cost in the flash memory, and t_b the total disk I/O cost. t_m is defined as $t_m = n_{fr} \times C_{fr} + n_{fp} \times C_{fp} + n_{fe} \times C_{fe}$, where n_{fr}, n_{fp}, and n_{fe} are the numbers of flash read, program, and erase operations. t_b is similarly defined as $t_b = n_H \times C_H$, with C_H being the cost of a disk access and n_H the number of disk accesses. Therefore, the trace execution time t_v is the weighted sum of all media access operations performed in the middle tier and bottom tier while running the trace.

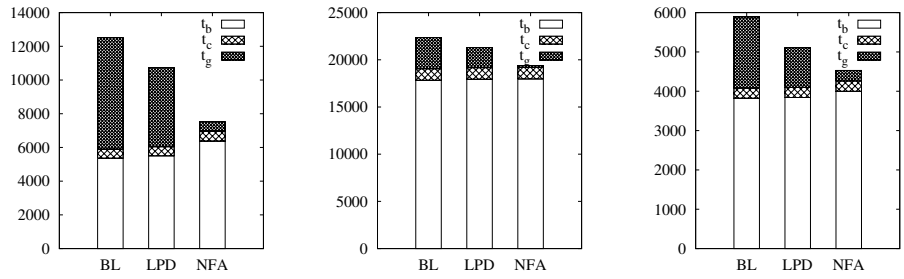
For the costs of flash operations, C_{fr}, C_{fp}, and C_{fe}, we used the corresponding performance metrics of a typical SLC NAND flash memory of a leading manufacturer, while the disk access cost corresponds to the average latency of a WD1500HLFS HDD [25]. These costs are listed in Table 2.

Fig. 5 compares the overall performance of the three approaches under the TPC-C, TPC-H, and TPC-E workloads. Our two approaches, NFA and LPD, significantly outperformed BL, and NFA had a clear performance advantage over LPD. For the TPC-C workload, NFA achieved an improvement of 43% and 66% compared with LPD and BL respectively.

The performance improvement of our approaches can be entirely credited to the cost reduction in the middle tier, because both of our approaches do not focus on minimizing disk accesses. In fact, they even had a slightly higher number of disk accesses due to proactive page dropping. It is expected that a small fraction of the dropped pages are re-requested shortly after the dropping, which increases disk accesses. However, this is the small price we have to pay in order to achieve the overall performance gain.

Fig. 6 confirms our expectation, where we provide a breakdown of the execution times according to (1). The mid-tier cost t_m is further broken down into two fractions: the fraction caused by GCs, denoted as t_g, and the fraction caused by normal caching operations (e. g., read operations due to cache hits and program operations due to cache replacements), denoted as t_c, such that

$$t_v = t_m + t_b = (t_g + t_c) + t_b$$



(a) TPC-C **(b)** TPC-H **(c)** TPC-E

Fig. 6: Breakdown of the trace execution time (seconds) into the fractions of GC t_g, cache overhead t_c, and disk accesses t_b

As clearly shown in Fig. 6, both our approaches effectively improved the GC fraction, without significantly increasing the cost of other two fractions.

The remainder of this section is a detailed analysis of the experimental results. Due to space constraints, we only focus on the performance metrics collected under the TPC-C workload and omit those of the TPC-H and TPC-E workloads, from which similar observations were made.

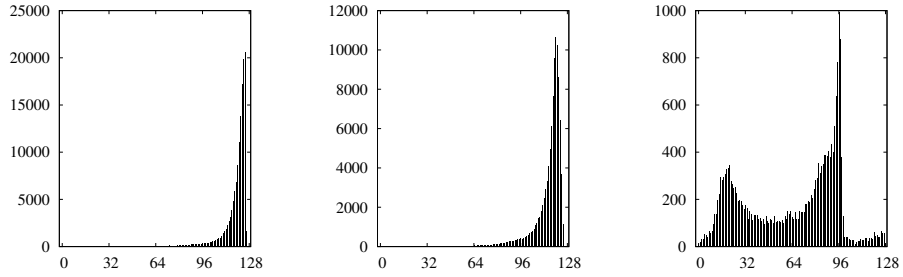
4.2 Detailed analysis

To further understand why our approaches improved the GC efficiency and reduced the number of its invocations, we plotted, in Fig. 7, the distribution of the number of valid pages in garbage-collected blocks. The majority of blocks garbage-collected in the BL configuration had a number of valid pages very close to 128, which resulted in a poor efficiency of GC. Compared with Fig. 7a, the dense region in Fig. 7b is located slightly farther to the left, meaning fewer valid pages in the garbage blocks. For NFA, the majority of garbage-collected blocks had less than 96 valid pages per block, i. e., more than 32 pages could be freed for each garbage block.

Interestingly, in Fig. 7c, the region between 96 and 127 is very sparse. This is the filtering effect (Line 6 to 7 of Algorithm 3). The valid pages in a block either become invalidated due to logical overwrites or are filtered out when they become cold. Therefore, the probability that a block has full or close-to-full utilization is artificially reduced.

For LPD, we ran the trace multiple times scaling d from 0 up to 65,536, which controls how greedily pages are dropped from the cache. For $d = 0$, LPD is equivalent to BL, which does not use the extended FTL and does not drop any pages. For $d = 65536$, it drops all pages from the cache whenever a cache replacement occurs (Line 5 to 11 of Algorithm 1).

Under the same workload (independent of d), NFA processed 22,106 GCs and achieved a hit ratio of 0.7438. Relative to these values, Fig. 8 plots the number of GCs and the hit ratio of LPD, with d scaled from 0 to 65,536. For $d = 0$,



(a) BL **(b)** LPD **(c)** NFA

Fig. 7: Distribution of the number of valid pages in garbage-collected blocks. A bar of height y at position x on the x-axis means that it happened y times that a block being garbage-collected contains x valid pages. Note the different scales of the y-axis.

LPD (and BL, due to equivalence) obtained a slightly higher hit ratio than NFA (by 5.84%), however, its number of GCs was much higher than that of NFA (by a factor of five). For $d = 65536$, although LPD's number of GCs was greatly reduced (still higher than that of NFA by 21%), its hit ratio drastically dropped and became only 63.1% of the NFA hit ratio. Note, we could not find a value for $d \in [0, 65536]$ for LPD, such that the number of GCs is lower and the hit ratio is higher than those of NFA at the same time.

4.3 Wear leveling

So far, we have not discussed other aspects of flash memory management such as wear leveling and bad block management, which are not the focus of our current work, because they can be dealt with using standard techniques proposed in previous works related to FTL. However, fortunately, our approaches seem to

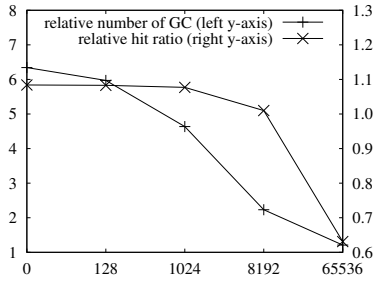


Fig. 8: Number of GCs and hit ratio of LPD relative to NFA, when d is scaled from 0 to 65,536

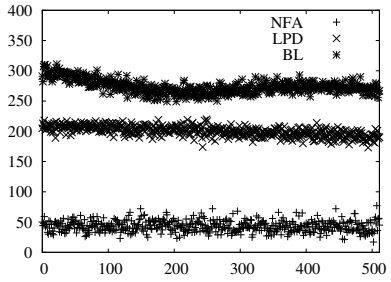


Fig. 9: Number of erases for each block. Each position on the x-axis refers to a block.

have automatically distributed the erases uniformly to the blocks, as shown in Fig. 9, where the number of erases for each of the 512 blocks is plotted for all three approaches under comparison.

5 Conclusion

In this paper, we studied the problem of efficiently using flash memory for a mid-tier cache in a three-tier storage system. We identified the problems of using flash memory indirectly, which is the common approach taken by previous works. Among these problems, the most important one is the CPM problem, which not only greatly impacts performance, but also shortens the life time of flash-based devices used in the cache. Our basic idea to solve this problem is to drop cold pages proactively and ignore them during GCs. Based on this basic idea, we proposed two approaches, an indirect one and a native one, that effectively handle the problem, as shown by our experiments. The experiments also demonstrated the gravity of the CPM problem, which is ignored so far by typical indirect approaches represented by the baseline. The cache-specific knowledge (e. g., which pages can be dropped) and the direct control over the flash memory (e. g., when is the GC to be started) is the key to the significant performance gain achieved by NFA, the native approach.

We believe that the optimal use of flash memory in a mid-tier cache can only be achieved when the flash memory is managed natively by the cache management software. For similar reasons, system designers should seriously consider how to natively support flash memory in the database software.

6 Acknowledgement

Yi Ou's work is partially supported by the German Research Foundation and the Carl Zeiss Foundation. Jianliang Xu's work is partially supported by Research Grants Council (RGC) of Hong Kong under grant nos. HKBU211510 and G_HK018/11. The authors are grateful to German Academic Exchange Service (DAAD) for supporting their cooperation. They are also grateful to anonymous referees for valuable comments.

References

1. Gal, E., Toledo, S.: Algorithms and data structures for flash memories. ACM Computing Surveys **37**(2) (2005) 138–163
2. Ban, A.: Flash file system (April 1995) US Patent 5,404,485.
3. Birrell, A., Isard, M., Thacker, C., Wobber, T.: A design for high-performance flash disks. SIGOPS Oper. Syst. Rev. **41**(2) (2007) 88–93
4. Ban, A.: Flash file system optimized for page-mode flash technologies (October 1999) US Patent 5,937,425.
5. Estakhri, P., Iman, B.: Moving sequential sectors within a block of information in a flash memory mass storage architecture (July 1999) US Patent 5,930,815.

6. Kim, J., Kim, J.M., et al.: A space-efficient flash translation layer for CompactFlash systems. IEEE Trans. on Consumer Electronics **48**(2) (May 2002) 366–375
7. Lee, S.W., Park, D.J., et al.: A log buffer-based flash translation layer using fully-associative sector translation. ACM Trans. Embed. Comput. Syst. **6**(3) (July 2007)
8. Gupta, A., Kim, Y., Urgaonkar, B.: DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In: Proc. of ASPLOS'09, New York, NY, USA, ACM (2009) 229–240
9. Kim, J.K., Lee, H.G., et al.: A PRAM and NAND flash hybrid architecture for high-performance embedded storage subsystems. In: Proc. of EMSOFT'08, New York, NY, USA, ACM (2008) 31–40
10. Chung, T., Park, D., Park, S., Lee, D., Lee, S., Song, H.: A survey of flash translation layer. Journal of Systems Architecture **55**(5) (2009) 332–343
11. Chang, L.P., Kuo, T.W., Lo, S.W.: Real-time garbage collection for flash-memory storage systems of real-time embedded systems. ACM Trans. Embed. Comput. Syst. **3**(4) (November 2004) 837–863
12. INCITS T13: Data Set Management commands proposal for ATA8-ACS2 (revision 6). `http://t13.org/Documents/UploadedDocuments/docs2008/e07154r6-Data_Set_Management_Proposal_for_ATA-ACS2.doc` (2007)
13. Zhou, Y., Chen, Z., et al.: Second-level buffer cache management. IEEE Trans. on Parallel and Distributed Systems **15**(6) (2004) 505–519
14. Chen, Z., Zhang, Y., et al.: Empirical evaluation of multi-level buffer cache collaboration for storage systems. In: Proc. of SIGMETRICS'05, ACM (2005) 145–156
15. Jiang, S., Davis, K., et al.: Coordinated multilevel buffer cache management with consistent access locality quantification. IEEE Trans. on Computers (2007) 95–108
16. Gill, B.: On multi-level exclusive caching: offline optimality and why promotions are better than demotions. In: Proc. of FAST'08, USENIX Association (2008) 1–17
17. Koltsidas, I., Viglas, S.D.: The case for flash-aware multi-level caching. Technical report, University of Edinburgh (2009)
18. Ou, Y., Härder, T.: Trading memory for performance and energy. In: Proc. of DASFAA'11, Springer-Verlag (2011) 241–253
19. Canim, M., Mihaila, G., et al.: SSD bufferpool extensions for database systems. In: Proc. of VLDB'10. (2010) 1435–1446
20. Do, J., DeWitt, D., Zhang, D., Naughton, J., et al.: Turbocharging DBMS buffer pool using SSDs. In: Proc. of SIGMOD'11, ACM (2011) 1113–1124
21. Rosenblum, M., Ousterhout, J.K.: The design and implementation of a log-structured file system. ACM Trans. Comput. Syst. **10** (February 1992) 26–52
22. Kawaguchi, A., Nishioka, S., Motoda, H.: A flash-memory based file system. In: Proc. of TCON'95, Berkeley, CA, USA, USENIX Association (1995)
23. On, S.T., Xu, J., et al.: Flag Commit: Supporting efficient transaction recovery on flash-based DBMSs. IEEE Trans. on Knowledge and Data Engineering **99** (2011)
24. Prabhakaran, V., Rodeheffer, T.L., Zhou, L.: Transactional flash. In: Proc. of OSDI'08, Berkeley, CA, USA, USENIX Association (2008) 147–160
25. Western Digital Corp.: Specifications for the 150 GB SATA 3.0 Gb/s VelociRaptor drive (model WD1500HLFS). `http://wdc.custhelp.com/app/answers/detail/search/1/a_id/2716` (2011)