2017

Towards Design and Analysis For High-Performance and Reliable SSDs

Qianbin Xia

TOWARDS DESIGN AND ANALYSIS FOR HIGH-PERFORMANCE AND RELIABLE SSDS

A Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

by

QIANBIN XIA

B.S., Beijing Institute of Technology: Sep. 2007 to Jul. 2011

Director: Dr. Weijun Xiao,

Assistant Professor, Department of Electrical and Computer Engineering

Virginia Commonwealth University

Richmond, Virginia

May, 2017

Acknowledgements

I would like to express my sincere gratitude to my advisor Dr. Weijun Xiao, for his continuous support of my Ph.D study and research, for his patience and immense knowledge. His guidance helped me a lot in all the time of my research and writing of this dissertation. Without his guidance and persistent help, this dissertation would not have been possible. I would also like to thank my committee members, Dr. Wei Cheng, Dr. Preetam Ghosh, Dr. H. Klenke, and Dr. Ruixin Niu for serving on my advisory committee and for their insightful comments and encouragement. I would also like to extend my thanks to other faculty and stuff members of the department of Electrical and Computer Engineering, in particularly to the Dr. Xubin He, Dr. UMIT OZGUR, and administrative assistants Stacy E. Metz and Ellen Gresham.

Meanwhile, I thank my lab mates Dongwei Wang, Liang Xu and other close friends in ECE deparment: Ping Huang, Tao Lu, Yijie HuangFu, Yuhua Guo, and Kun Tang. They helped me a lot in both my study and life. There are also some local friends in Richmond like Bob and Elain, they have invited me several times to their beautiful farm and helped me a lot in my language skills.

I would like to thank my family in China: my parents, my sister, and my brother, for supporting me financially and spiritually. Especially my parents, I appreciate all the support they have provided me over the years. They taught me the value of hardwork and an education.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract

TOWARDS DESIGN AND ANALYSIS FOR HIGH-PERFORMANCE AND RELIABLE SSDS

By Qianbin Xia

A Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

Virginia Commonwealth University, 2017.

Director: Dr. Weijun Xiao,

Assistant Professor, Department of Electrical and Computer Engineering

NAND Flash-based Solid State Disks have many attractive technical merits, such as low power consumption, light weight, shock resistance, sustainability of hotter operation regimes, and extraordinarily high performance for random read access, which makes SSDs immensely popular and be widely employed in different types of environments including portable devices, personal computers, large data centers, and distributed data systems.

However, current SSDs still suffer from several critical inherent limitations, such as the inability of in-place-update, asymmetric read and write performance, slow garbage collection processes, limited endurance, and degraded write performance with the adoption of MLC and TLC techniques. To alleviate these limitations, we propose optimizations from both specific outside applications layer and SSDs' internal layer. Since SSDs are good compromise between the performance and price, so SSDs are widely deployed as second layer caches sitting between DRAMs and hard disks to boost the system performance. Due to the special properties of SSDs such as the in-

ternal garbage collection processes and limited lifetime, traditional cache devices like DRAM and SRAM based optimizations might not work consistently for SSD-based cache. Therefore, for the outside applications layer, our work focus on integrating the special properties of SSDs into the optimizations of SSD caches. First, we propose to leverage the out-of-place update property of SSDs to improve both the performance and lifetime of SSDs. Second, a new zero-migration garbage collection is proposed for SSD read cache to reduce the internal garbage collection activities and prolong the lifetime of SSDs without sacrificing the cache performance. Moreover, when SSDs are deployed as write caches, we come up with a locality-driven dynamic cache allocation scheme to improve both the performance and lifetime of SSD cache by compromising the cache hit ratio and the internal garbage collection overhead. Finally, a workload-aware hybrid ECC design is proposed to alleviate the flash write performance degradation without hurting the read performance and data reliability.

CHAPTER 1

INTRODUCTION

1.1 Background and Problem Statement

Flash memory is organized in units of blocks and pages. A fixed number (32 or 64) of pages compose a block. There are three main operations in flash memory: read, write, and erase. Read and write operations are performed in the unit of pages, while erase operations are on a block basis. Flash memory has several distinctive features, such as out-of-place updates, internal garbage collection, asymmetrical read and write performance, limited erase cycles, and plenty of internal parallelism. Before updating a flash page in SSDs in-place, the whole flash block need to be erased, which is a time-consuming process and introduce unacceptable write latency. Out-of-place update is adopted to mitigate the limitation of time-consuming in-place update. Instead of updating the data in-place, the new data is directed to other free blocks and the old version of data is marked as invalid. An internal garbage collection process will be trigged to reclaim these invalid flash pages due to the out-of-place updates. Before erasing the victim block, the valid data will be migrated to other free space and then the whole victim block will be erased. This internal garbage collection process could significantly affect the performance of SSDs. Besides, each flash block could only be erased limited cycles, after which, the flash block will be unreliable and marked as bad. What's more, flash write operations are much slower than flash read, which could be the performance bottleneck of the SSDs. The emergence of MLC (stores two bit information per cell) and TLC (stores three bit information per cell) technology is to increase the memory density and reduce the price, while at the same time, impairs

the performance and endurance. Table 1 depicts the main parameters for SLC, MLC, and TLC flash memories [1].

Table 1. Parameters for SLC, MLC and TLC

Access Type (unit)	SLC (2KB)	MLC (4KB)	TLC (8KB)
Read (page)	25 us	50 us	∼75 us
Write (page)	0.2∼0.3 ms	0.6∼0.9 ms	0.9∼1.35 ms
Erase (block)	1.5∼2 ms	3 ms	∼4.5 ms
Lifetime (cycle)	100,000	3,000	1,000

Since traditional file systems are designed for the in-place update storage devices, Flash Translation Layer (FTL) has been developed and deployed in SSDs to mimic in-place update like block devices in order to make flash memory compatible with the existing file systems. An FTL includes three main function units: address translation, garbage collector, and wear-leveler. The address translation unit translates the logical page number to the physical page number in the flash memory and hides the erase-before-write feature of flash memory. The mapping methods could be coarsely classified into three categories: page-level mapping, block-level mapping, and hybrid mapping. A page-level mapping [2] can achieve the best performance, it is constrained by the limited resource of expensive SRAM. While a block-level mapping [3] could save huge amount of memory space for the mapping information, it will lead to space wastage and performance degradation. To reach a compromise, several hybrid schemes [4, 5, 6, 7, 8] have been proposed that combine the page-level and block-level mapping together and are mainly based on the following idea: most of the data are mapped at the block level to reduce the overhead, while a small fraction of the data that are frequently accessed are mapped at the page level to guarantee the performance. A garbage collector is used to reclaim the obsolete pages caused by the

out-of-place updates. Whenever the number of free pages drops below a predefined threshold, a garbage collection process will be triggered to create more free space for the incoming requests. A victim block will be selected from the pool, all the valid data in the victim block will be moved to other free space, then the whole block will be erased. There are several different algorithms to select the victim blocks: FIFO GC algorithm [9, 10, 11], which selects the blocks in a cyclic manner; greedy GC algorithm [11, 12], which selects the block with the fewest number of valid pages; the windowed GC algorithm [13], which is a combination of the FIFO and greedy algorithms; the d-choices GC algorithm [14, 15] which selects the victim block containing the fewest number of valid pages from d randomly chosen blocks. The objective of wear-leveler is to get an even erase-count distribution among all flash-memory blocks to improve the overall endurance of flash memory.

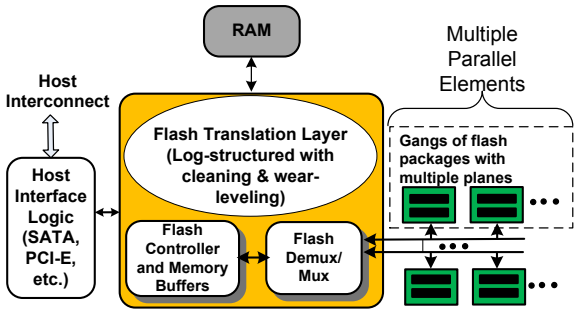


Fig. 1. SSD Architecture [16].

Figure 1 shows the architecture of typical SSDs. A typical SSD is composed of a host interface, an SSD controller, DRAM, a flash controller, ECC engine, and flash chips. The SSD controller contains a processor and an SRAM to store the firmware. The controller is responsible for the data placement, garbage collection, wear leveling, and bad block management. DRAM is used to store both the user data and mapping information. The DRAM allocation between the data cache and mapping table could

is one of the key factors to the performance and lifetime of SSDs. The performance of each individual Flash chip is relatively poor. In order to provide higher bandwidth, modern SSDs are organized into multi-channels. All the channels are independent of each other and can work in a parallel way. Each channel has a flash controller to buffer the pending requests and send the requests to the lower level in a channel. Within a channel, there could be multi-packages and each package contains multiple dies. All these packages and dies could work in an interleaving manner. How to fully utilize the internal parallelism to boost the performance of SSDs is still an opening question. Finally, the ECC engine is deployed to encode and decode the user data and provide stronger reliability.

In summary, SSDs have several inherent properties including out-of-place update, internal garbage collection, asymmetrical read and write performance, limited lifetime, and internal parallelism, which motivates us to design new architectures and algorithms to fully exploit the merits and alleviate the limitations of SSDs for both SSD-based caches and primary storages.

1.2 Proposed Approaches

Our optimizations target both the outside application layers and internal components. For the outside application layers, we focus on the the cache where SSDs are deployed as caches. Previous work have shown the severe interferences between read and write [17, 18], when read and write requests are mixed together and sent to the same SSD. The interference could especially degrade the read performance of SSD cache due to much higher write and erase latencies. Moreover, mixed read and write data in the same Flash block can lead to higher garbage collection overhead. Besides, Xia et al. has shown that read and write requests could be well separated by analyzing the real word IO trace files. Therefore, in our work, we assume separate

SSD-based read and write caches are deployed in the system.

First, we propose to utilize the out-of-place update property of SSDs to improve the performance and lifetime of SSD read cache. Due to the out-of-place update, when a cache eviction occurs, only the metadata will be removed, however the real user data still resides in the flash memory and is accessible before the whole flash block being erased. Therefore, we propose a flash-aware cache design that leverage these evicted but still accessible data to improve both the performance and lifetime of SSD read cache.

Second, a new zero-migration garbage collection scheme is proposed to further extend the lifetime of Flash-based read cache. The valid data migration process inside SSDs during the garbage collection processes will not only introduce extra latency, but also bring additional write operations and hurts the lifetime of SSDs. However, when SSDs are used as read caches, all the data inside SSD cache always have exact copies in the hard disks or write buffers. Therefore, unlike traditional garbage collection processes, our zero-migration garbage collection scheme will aggressively erase the whole flash block without performing valid data migrations to reduce the latency and alleviate the lifetime issue.

Third, we come up with a new locality-driven dynamic Flash cache allocation design to improve both the performance and lifetime of Flash-based cache. SSDs have internal garbage collection activities, which can have significant impact on the cache performance. Moreover, SSDs have limited endurance. Therefore, traditional cache hit ratio oriented optimizations might not obtain consistent performance benefit and may even hurt the endurance of Flash-based cache. Our locality-driven dynamic Flash cache allocation design aims to achieve the optimal cache performance by compromising the cache hit ratio and internal garbage collection overhead. What's more, compared with traditional Flash cache configurations, our proposed design can also

help to prolong the device lifetime.

Finally, we propose a workload-aware differentiated ECC design to improve the Flash write performance. Due to the adoption of ISPP Flash programming scheme, there is an inherent tradeoff between the flash write performance and reliability. Our workload-aware differentiated ECC design applies a fast write scheme with stronger ECC scheme to enhance the flash write speed without compromising the read performance.

CHAPTER 2

FLASH-AWARE HIGH-PERFORMANCE AND ENDURABLE CACHE

2.1 Introduction

An SSD is a good compromise among performance, capacity, and cost. DRAM is too costly and obviously not a persistent storage medium (it loses data when power outage occurs). Conventional HDDs are too slow. Therefore, SSDs are widely used as caches sitting between DRAM and hard disk drives (HDDs) to fill the huge performance gap between DRAM and HDDs [19, 20, 21, 22, 23, 24, 25, 26]. Despite all these attractive merits, SSDs suffer from several inherent limitations, especially the limited erase cycles. Each flash block could only be erased limited cycles, after which the block will be unreliable and marked as bad block. In [20], the authors showed how serious the limited lifetime issue of SSDs could be. When a 60GB Intel 520 SSD is used as a data cache for a deduplication system, where the available capacity of SSD cache is 5% of the deduplicated data. By taking the write speed and the total allowed written amounts before wearing out of Intel 520 SSD into account, the expected SSD lifetime is only several days.

Flash-based SSDs have several distinct properties compared with hard disk drives. Two of the most important aspects are erase-before-write and out-of-place update. A page could only be updated after erasing a whole block which contains multiple pages. The erase operation takes about several milliseconds [27] which will degrade the write performance of SSDs, and out-of-place update is adopted to alleviate the influence of slow erase operations. Instead of updating the data in the original physical loca-

tion, the new data is written to a new free location and the previous data is marked as invalid which will be reclaimed in the future. To support out-of-place updates, a mapping table that associates logical page number with physical page number is maintained by the controller. Whenever the accumulation of invalid pages reaches a threshold, a garbage collection process will be triggered to reclaim the obsolete space. In a typical SSD, the real physical capacity is always larger than the user-addressable physical space, and the surplus space is called over-provision. The over-provisioning part of SSDs is used for two purposes. One is to support the out-of-place update and reduce the frequency of garbage collection. The other is to substitute bad blocks. For enterprise applications where reliability and performance stability are of paramount importance, a large amount of flash memory will be reserved as the over-provisioning space.

When SSDs are used as primary storage devices, previous research work has leveraged the out-of-place update property to improve the performance and alleviate the limitations of flash memory under some special application scenarios like RAID (Redundant Arrays of Inexpensive Disks) [28], CDP (Continuous Data Protection) [29], and Snapshots [30]. While, to the best of our knowledge, when SSDs are used as caches, none of the existing research work has utilized the out-of-place update property to improve the performance. For general cache algorithms, when there is a cache miss and the cache is full, a cache entry will be replaced out by a replacement algorithm, then the missed data will be loaded from the low level storage and inserted into the cache. However, for flash cache, the eviction merely removes the metadata, and the actual user data is still accessible and resides in the physical flash page before being updated or erased.

In this chapter, we propose a flash-aware read cache design through leveraging these evicted but still accessible pages inside SSDs with negligible overhead. Addi-

tionally, a new zero-migration garbage collection scheme is proposed and implemented to further mitigate the lifetime limitation of flash cache.

2.2 Related Work

When flash memory is used as cache, lifetime is one of the main concerns and is becoming more serious due to the continuous decreasing of feature size and adoption of MLC and TLC technologies. A number of solutions have been proposed to alleviate the lifetime problem, typical techniques focus on designing more robust ECC [31, 32], or on improving traditional wear-leveling techniques [33]. Due to the garbage collection and wear leveling processes, the actual number of write operations inside the flash memory is larger than the write requests from the host, which is called write amplification. Several research works tried to extend the lifetime of flash memory through reducing write amplification [34, 35, 36]. CAFTL proposed by Chen et al. [37] integrated the data-deduplication technique into the FTL of SSD to reduce unnecessary duplicate writes and save the lifetime of SSD.

Another way to improve the lifetime of flash memory is retention relaxation. Retention errors are the dominant source of flash memory errors which are caused by charge leakage after the flash cells being programmed [38]. Liu et al. [31] improved the write speed and mitigated the requirement for stronger ECC codes by relaxing the retention time requirement. Cai et al. [39] proposed FCR (Flash Correct-and-Refresh) to extend the limited erase cycles due to retention errors. FCR reads, corrects, and reprograms (in-place) or remaps the stored data before the accumulation of the retention errors exceeding the capability of ECC. Huang et al. [40] aggressively placed the frequently updated data into the worn-out flash blocks which could only sustain shorter retention time to prolong the lifetime of these dead blocks.

Besides these typical techniques to improve the endurance of flash memory, other

research work focuses on specific optimizations for the flash cache. BPLRU proposed by Kim et al. [41] deployed a RAM inside SSD as a write buffer to improve performance and lifetime of flash memory. Kgil et al. [19] put forward a scheme which split the flash cache into separate read and write regions with changeable error correction strength and cell density to improve reliability and lifetime of flash memory. NetApp used flash memory as a second level read cache while used the NVRAM as the second level write cache [42]. Soundararajan et al. [21] used a hard disk drive as a write cache for SSDs. How the partition between the user space and over-provisioning space affects the performance of flash caches was explored in [43]. In that work, the over-provisioning space was dynamically configured based on the properties of the workloads to improve the performance and lifetime at the same time. All of the above proposed schemes are complementary to our solution.

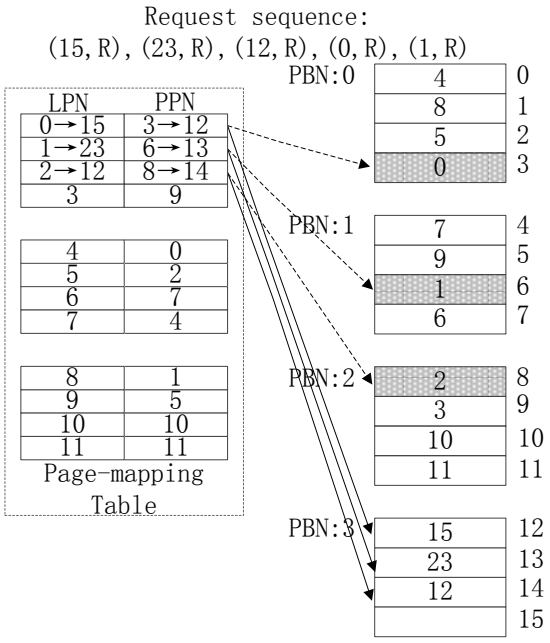


Fig. 2. A simplified example to illustrate the motivation behind our flash-aware cache design.

2.3 Design and Implementation

2.3.1 Motivation Example

In a page-level mapping scheme, a mapping entry consists of an LPN (logical page number) and a corresponding PPN (physical page number). The whole mapping table is constructed and maintained in both RAM and NAND flash memory. When a write request comes , the mapping table will be checked to verify whether the request is a new write or an update for existing data. For a new write, the data will be written to a free location and a new mapping entry will be added to the mapping table. While for an overwrite, first, the data will be written to a new free location, then the old page will be marked as invalid and the mapping table will be updated to reflect this change. But for a flash read cache, the situation is a little bit different. Invalid data will be generated only when a cache miss happens and the cache is full.

Figure 2 shows a simplified example of out-of-place update and the generation of invalid pages under a page-level mapping scheme. For simplicity, we assume that there are four physical blocks and each block consists of four pages. As we have mentioned, the flash memory has some over-provisioning capacity to support the out-of-place update and bad block replacement. In our example, although the real physical capacity is four blocks, user-addressable space is only 3 blocks. Equation (4.1) is the definition of over provision.

$$OP = (C_{total} - C_{user})/C_{total}; \tag{2.1}$$

C_{total} is the real physical capacity, C_{user} is the user-addressable capacity, OP means the percentage of over-provision. We assume the OP is 25% in our example. Initially, the user-addressable space is full, but the over-provisioning part is totally empty. We use the corresponding logical page number to represent the user data in each physical

page. Then a series of requests come from the up level. A request is expressed by its LPN and operation type (read or write). In this figure, the dotted arrows point to the obsolete mapping information. The first request is a read request for LPN 15 which will result in a cache miss. Thus an entry will be evicted out based on a specific cache replacement algorithm for example LRU. Here we assume the LRU entry is LPN 0, so that entry will be evicted out and the corresponding data (PPN 3) will be marked as invalid. Then data for LPN 15 will be fetched from the lower storage device and written to PPN 12. The mapping entry for LPN 15 will be added to the mapping table. The next read requests for LPN 23 and LPN 12 are similar to the previous read request for LPN 23. The LRU cache entries will be evicted out and replaced by the new entries.

Then the fourth and fifth requests are both read requests for the previous evicted data. When treated as a traditional cache device like DRAM, these two requests will lead to two cache misses, we need to evict out two cache entries, fetch the data for LPN 0 and LPN 1 from the lower storage device, write the new data into the flash memory and then update the mapping information. This process will not only degrade the performance, but also reduce the lifetime of flash cache. Fetching data from the lower level storage device like HDDs will introduce a long latency, rewriting the new data into flash memory also brings timing overhead and more erase operations which is also a long-latency process and harmful to the lifetime of flash memory. Even the mapping entries for LPN 0 and LPN 1 have been evicted out from the mapping table, but the user data still reside in the PPN 3 and PPN 6. This gives us the opportunity to design a flash-aware cache architecture. Instead of fetching the data from the lower-level storage device and rewriting it into flash memory, we can merely revive it.

2.3.2 LRU and ARC

Least recently used (LRU) data replacement is one of the most basic and classic cache replacement algorithms. The main idea of LRU is: data recently used is likely to be reused in the near future; data not used in ages is not likely to be used again in the near future. To age the data, a queue will be maintained, recently used at the front and the oldest at the rear. Every time a page is referenced, it is moved to the head of the queue. When a cache miss happens and the cache is full, the LRU based policy evicts the entry which was requested least recently. Then for a read request, the missed data will be fetched from the lower storage device and written into the head of LRU queue.

Basic LRU merely captures the recency of workloads, Adaptive Replacement Cache (ARC) [44] improves the basic LRU algorithm by capturing both the recency and frequency at the same time and dynamically, adaptively, and continually balancing between the recency and frequency components in an online and self-tuning fashion according to evolving and changing access patterns. In the original ARC architecture, the cache directory is split into two lists, T1 and T2. T1 is used to cache the recently referenced entries, while T2 is used to cache the frequently referenced entries. For any entries in T1, it should be accessed only one time recently, and for any entries in T2, it should be accessed at least twice. Two ghost lists B1 and B2 which only contain the metadata are attached to the bottom of T1 and T2. B1 and B2 are used to record the recently evicted entries from T1 and T2, respectively. The main idea of the learning process is as follows: if there is a hit in B1 then we should increase the size of T1, and if there is a hit in B2 then we should increase the size of T2. To support this learning process, a tunable parameter p is defined as the target size of T1. On a hit in B1, the value of p will be increased, and on a hit in B2, the

value of p will be decreased.

2.3.3 LRU-Based Flash-Aware Cache Design

Algorithm 1 Flash-Aware LRU

Input: The read request stream $x_1, x_2, ..., x_t....$

For every $t \geq 1$ and any x_t, one of the following three cases must occur.
Case I:x_t hits in the LRU queue.
  Move x_t to the head of LRU queue.
  Return the data.
Case II:x_t misses in the LRU queue but hit in the SQ.
  Move the tail entry from the LRU queue to the head of SQ.
  Move x_t from SQ to the head of the LRU queue.
  Return the data.
Case III:x_t misses in both the LRU queue and the SQ.
  if the cache is full
    Move the tail entry from the LRU queue to the head of SQ.
  endif
  Fetch the data from the lower level storage device into the SSD.
  Move x_t to the head of LRU queue.
  Return the data.

ERASE:
  If the victim block contains any entries in SQ
    Delete the corresponding entries in SQ.
  endif

As we described previously, a cache eviction for a flash memory only discards the metadata, while the user data still resides in the physical location. When a read request for the evicted but still available data arrives, instead of fetching the data from the lower level storage device, we can revive the suspected data. Algorithm 1 shows our LRU-based flash-aware cache algorithm FLRU. We add a suspected queue (SQ) to preserve the evicted entries. The size of the LRU queue is determined by the user-addressable physical capacity. The maximal size of the SQ in the number of entries is given by equation (4.2).

$$N_{SQ} = (OP - GC_{th}) * C_{total}; \tag{2.2}$$

GC_{th} is the garbage collection threshold which is defined as the percentage of free physical capacity over the total physical capacity. The reason is that whenever the number of free pages drops to the garbage collection threshold, a garbage collection

process will be triggered to reclaim invalid pages. Therefore, only the subtraction between the over-provision and garbage collection threshold could be utilized by our flash-aware design. On a hit in LRU queue, we will move the requested entry to the head of the LRU queue and return the data like the normal LRU-based cache. On a miss in LRU queue, unlike original LRU-based cache, we will first check with the SQ. If the request hits in the SQ, we can revive it through moving the requested entry from the SQ to the head of the LRU queue. As the entries are maintained in the memory, hence the overhead of moving an entry from the SQ to LRU queue is negligible especially when compared with the long-latency lower-level read and flash write operations. Therefore the access latency of hitting in the SQ is almost the same as hitting in the LRU queue. In this case, a read operation in the lower-level storage device and a write request for the flash cache could be avoided. For a request which misses in both the LRU queue and SQ, we firstly need to move the tail entry from the LRU queue to the head of SQ if the LRU queue is full. Then the requested data will be fetched from the lower-level storage device and written into the flash memory.

Besides adding the additional SQ and changing the original LRU algorithm, we also need to modify the garbage collection part of SSDs. When a garbage collection process is triggered, a victim block will be erased. All the invalid data inside the victim block will never exist any more after the erasure, hence there is no need to preserve the corresponding entries in the SQ. The bottom of algorithm 1 shows our modified garbage collection process. Whenever we need to erase a victim block, if there is any entry for pages in this victim block buffered in the SQ, these entries will be discarded from the SQ.

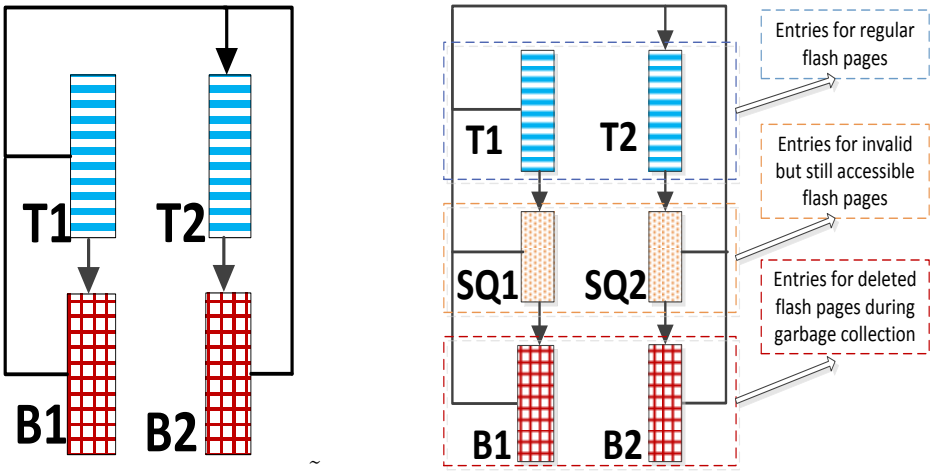
Algorithm 2 Flash-Aware ARC

Input: The read request stream $x_1, x_2, ..., x_t....$

Initialization:Set p=0, T1,T2,B1,B2,SQ1,SQ2 to empty
For any x_t, one of the following six cases must occur.
Case I:x_t is in T1 or T2.
  Move x_t to the MRU position of T2.
  Return the data.
Case II:x_t is in B1.
  If $|B1| \geq |B2|$)
    $k_1 = 1$.
  else
    $k_1 = |B2|/|B1|$.
  endif
  Update p=min{p+k_1, c}
  Replace(x_t, p).
  Move x_t from B1 to the MRU position in T2.
  Also fetch x_t to the cache and return the data.
Case III:x_t is in B2.
  If $|B2| \geq |B1|$
    $k_2 = 1$.
  else
    $k_2 = |B1|/|B2|$.
  endif
  Update p=max{p-k_2, 0}
  Replace(x_t, p).
  Move x_t from B2 to the MRU position in T2.
  Also fetch x_t to the cache and return the data.
CASE IV:x_t is in SQ1 or SQ2.
  Replace(x_t, p).
  Move x_t from SQ1 or SQ2 to the MRU position in T2.
  Return the data.
CASE V:x_t is not in any of the queues.
  CASE A: T1 and B1 have exactly c pages.
    If($|T1| \leq c$)
      Delete LRU page in B1. Replace(x_t, p).
    else
      Here B1 is empty. Move LRU page from T1 to the head of SQ1.
    endif
  CASE B: T1 and B1 have less than c pages.
    If($|T1| + |T2| + |B1| + |B2| \geq c$)
      Delete LRU page in B2, if($|T1| + |T2| + |B1| + |B2|) = 2c$).
      Replace(x_t, p).
    endif
  Finally fetch x_t and move it to MRU position in T1.
  Return the data.

Subroutine Replace(x_t, p)
  If(($|T1| \neq 0$)and(($|T1| > p$)or(x_t is in B2 and $|T1|$=p)))
    Move LRU page from T1 to the head of SQ1.
  else
    Move LRU page from T2 to the head of SQ2.
  endif

ERASE:
  If the victim block contains any entries in SQ1 or SQ2
    Move the entries to the head of B1 or B2.
  endif



(a) Normal ARC. (b) Flash-Aware ARC.

Fig. 3. Comparison between normal ARC and flash-aware ARC.

2.3.4 ARC-Based Flash-Aware Cache

Figure 3(b) describes our flash-aware ARC architecture. Similar to flash-aware LRU architecture, SQ will be added to buffer the evicted but still accessible data in flash memory. But there is a little bit of difference, SQ is split into SQ1 and SQ2 in accordance with T1 and T2. The total size of SQ1 and SQ2 is also defined by equation (4.2). Algorithm 2 shows our flash-aware ARC algorithm FARC. We use c denote the user-addressable physical flash capacity. p is the target size of T1. A *Replace* function is defined to replace an entry out based on the value of p at that time when a cache miss happens or cache hits in SQ1 or SQ2. Unlike the original ARC algorithm, an entry evicted by the *Replace* function will be moved to SQ1 or SQ2 in our FARC algorithm. For any request, one of the six cases listed in algorithm 2 should happen. If a request hits in T1 or T2, the requested entry will be moved to the MRU position of T2 since it has been accessed twice recently. Another two cases: hitting in B1 and hitting in B2 will cause the adjustment of p. In this paper, we

follow the policy used in the original ARC algorithm to adjust the value of p. When a request hits in B1, p will be increased by k_1. If B1 contains more entries than B2, then k_1 is 1, otherwise, k_1 equals the lengths of B2 over the lengths of B1. While a request hits in B2, p will be decreased by k_2. If the length of B2 exceeds the length of B1, then k_2 will be 1, otherwise k_2 will equal the length of B1 over the length of B2. What's more, the value of p will be confined to a range between 0 and c. In fact, due to the adoption of our flash-aware design, the adjustment of p could be performed a little bit differently. The total usable space in flash cache is not c, but c plus the number of entries in SQ1 and SQ2 which we could call $c^{'}$. In the same way, the length of T1 could be extended to include the SQ1 and the length of T2 could be extended to include the SQ2. As the length of SQ1 and SQ2 is unfixed due to the garbage collection process, $c^{'}$, lengths of extended T1 and T2 are also fluctuant. Although we could use these extended variables to make more accurate adjustment for p, we ignored these factors in our paper for the purpose of simplicity. We believe that this does not affect the cache performance too much because the queue lengths of SQ1 and SQ2 are much shorter than those of T1 and T2. After the recalculation of p, an entry will be replaced out and the requested entry will be moved to MRU position of T2 which also has been accessed at least twice recently. Also, the requested data is fetched from the lower-level storage and written to the cache. The forth case is that a request hits in SQ1 or SQ2. An entry will be replaced out by calling the *Replace* function unit. After that, the requested entry is moved from SQ1 or SQ2 to the MRU position of T2. If a request misses in all the queues, an entry from B1 or B2 will be deleted and the *Replace* function will be called or an entry will be moved from T1 to SQ1 as depicted in case V.

The bottom of algorithm 2 is the modified garbage collection process for our new ARC-based flash cache. Whenever a victim block is going to be erased, any page

inside it will be checked, if a page is buffered in SQ1 or SQ2, it should be discarded from SQ1 or SQ2 and moved to B1 or B2.

2.3.5 Discussions of Implementation Issues

One potential problem with our flash-aware design is the communication between the cache management and garbage collection process. Currently, most of the SSDs are designed as black boxes and FTLs including the garbage collection function units are running on embedded processors within SSDs. While the cache management unit that contains the cache queues are maintained by the OS on the host side. Therefore, the cache replacement algorithm running on the host side is unaware of the semantic information about garbage collection. What's more, the address space in the cache queues is the addresses of underneath storage system, rather than the user-addressable space of SSDs. Hence, an additional mapping table should be maintained to translate the address space of the underneath storage system to the SSDs' address space [22].

One possible way to bridge the gap is to merge the FTLs with the cache management unit, either by opening the SSDs and moving FTLs into the host side like Triple-A [45], SDF [46] which has been widely deployed in Baidu's storage system, and Fushion-io's host based FTL [47], or moving the cache management units into the SSDs. By combining the cache management units with FTLs, the mapping table between underneath storage's address space and SSD's address space could be removed, and the mapping table inside the original SSDs can be merged with the cache queues by adding the physical addresses of flash memory to cache entries in the cache queues. Therefore, whenever a cache hit is detected through searching the cache queues, the corresponding physical location in the flash memory could be returned immediately. Moving FTLs to the host side has several benefits like performance enhancement, and cost reduction [45, 46, 47] due to the elimination of redundant re-

sources. The drawback is to consume additional host resources. On the other hand, moving cache management units into the SSDs could deliver good flexibility but result in higher requirement of the computing and memory resources inside SSDs. We adopt and implement the second solution in our simulations to verify the efficiency of our flash-aware design.

Another way is to design a special feedback interface which could expose necessary internal information of SSDs to the host side. A similar interface design has been utilized and proposed in [48] to support the nameless writes scheme which will return the physical address of the data inside SSDs to the host side after each write operation (nameless write interface) or data migration during garbage collection processes (migration or call back interface). Moreover, a real prototype for the nameless writes scheme is implemented and evaluated on the OpenSSD Jasmine board in [49]. To make our flash-aware and zero-migration designs work, we can utilize the migration interface in the nameless writes scheme, whenever the garbage collection happens inside SSDs, the feedback interface will send the LPNs of the invalid and valid pages in the victim block to the host side so that the cache management unit could eliminate the corresponding cache entries from the cache queues.

2.4 Experimental Methodology

We modified the Disksim with SSD extension to evaluate our proposed flash-aware cache schemes [27]. Table 2 lists the main parameters of our experiments. Since LRU and ARC are two of the most widely used cache replacement algorithms to evaluate a cache architecture, we choose the normal LRU and ARC algorithms as our baselines and implement both of them with the Disksim simulator. Our flash-aware LRU, flash-aware ARC, and zero-migration garbage collection algorithms as described in the previous section are implemented to show what benefits could be

Table 2. Configuration of Our Simulator

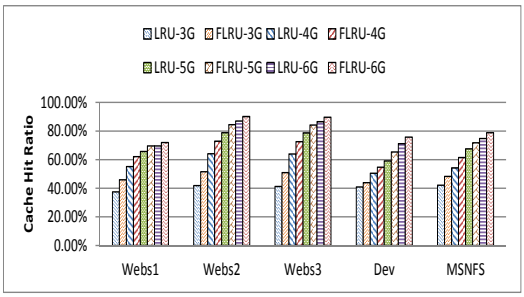
Flash Page Size	4KB
Flash Block Size	256KB
Over-Provision Space	15%, 25%, 35%
GC Threshold	5%
Cache Size	3GB, 4GB, 5GB, 6GB
Page Read Latency	25us
Page Write Latency	200us
Block Erase Latency	1.5ms
Disk Access Latency	5ms

obtained from our proposed schemes.

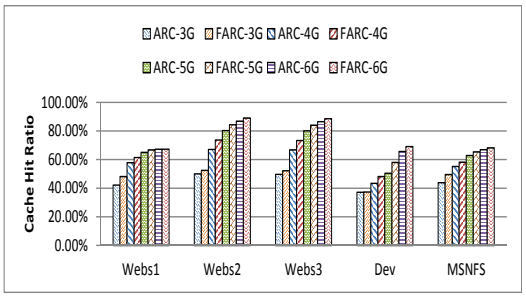
Five realistic workloads: WebSearch1, WebSearch2, WebSearch3, DevDivRelease, and MSNFS are used in our evaluation. WebSearch1, WebSearch2, and WebSearch3 were collected from popular search engines and nearly all the requests are read requests [50]. DevDivRelease and MSNFS are released by Microsoft [51]. DevDivRelease was collected for developers tools release server. MSNFS was collected for MSN storage file server. Since our flash cache is used as a read cache, we only pick out the read requests from DevDivRelease and MSNFS as our test benchmarks. Details of the characteristics of these workloads are depicted in Table 8.

2.5 Experimental Results

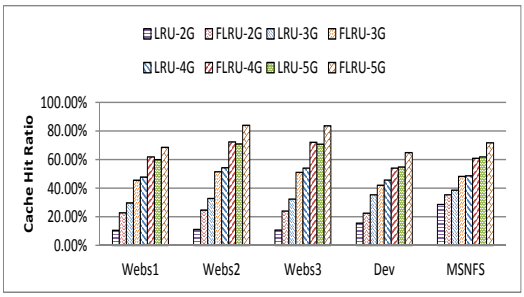
Cache hit ratio, average response time, and erase count are three main metrics used in this paper to evaluate our proposed flash-aware and zero-migration garbage collection cache designs. The first subsection shows the results when only our flash-aware design is applied. Normal LRU, flash-aware LRU, ARC, and flash-aware ARC



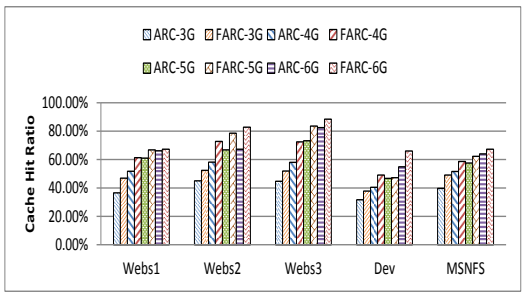
(a) LRU-15



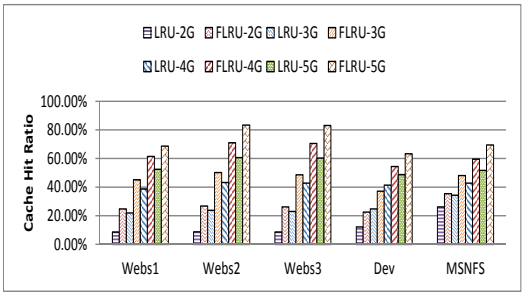
(b) ARC-15



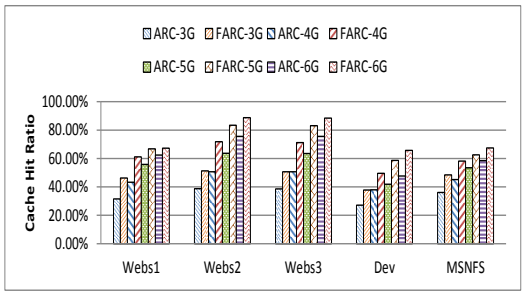
(c) LRU-25



(d) ARC-25



(e) LRU-35



(f) ARC-35

Fig. 4. Cache hit ratios of LRU, FLRU, ARC, and FARC. The cache capacities used here include: 3GB, 4GB, 5GB, and 6GB. Over-provisions configurations are 15%, 25%, and 35%.

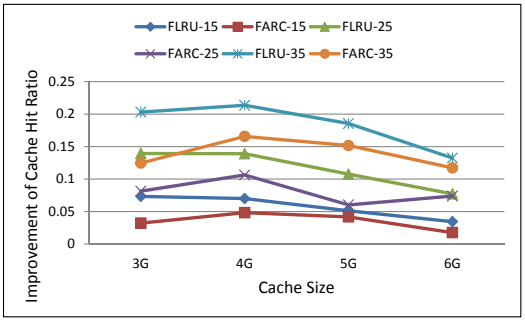
Table 3. Characteristics of I/O workload traces

Workload Name	**Addr. Space**	**Request Amount**	**Read Ratio**	**Avg. Req. Size (KB)**
WebSearch1	16.67GB	1055236	99.98%	15.15
Websearch2	16.67GB	4578819	99.98%	15.07
WebSearch3	16.67GB	4260446	99.97%	15.40
DevDivRelease	9.54GB	1608449	100%	18.45
MSNFS	9.54GB	1340894	100%	14.88

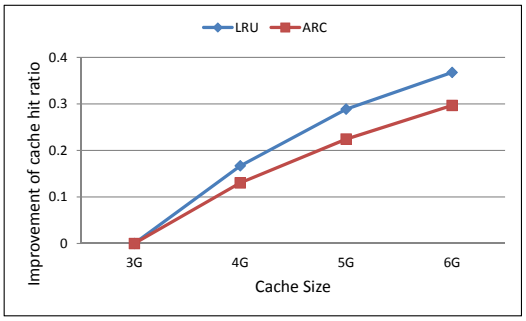
as we have described previously are implemented and simulated to measure the results. The second subsection presents the results when our zero-migration garbage collection scheme is integrated with the normal and flash-aware caches.

2.5.1 Cache Hit Ratio

Cache hit ratio is a common metric to evaluate a cache's performance and efficiency. Figure 4 shows our simulation results. For both LRU and ARC algorithms with multiple flash cache over-provision and capacity configurations, our flash-aware cache designs can achieve promising cache hit ratio improvement. For example, with 15% over-provision, our flash-aware LRU algorithm can increase the cache hit ratio by nearly 10% for WebSearch1 and WebSearch2 when the capacity is 3GB, while our flash-aware ARC design can obtain about 8% improvement in the best case. When over-provision is 25%, the cache hit ratio improvements could reach 19% and 12% for our FLRU and FARC, respectively. With 35% over-provision, FLRU and FARC could achieve about 28% and 21% cache hit ratio improvements in the best scenarios. Moreover, with the increasing of the over-provision, our flash-aware algorithms can gain more advantage over traditional algorithms. For example, for FLRU with



(a) LRU-15

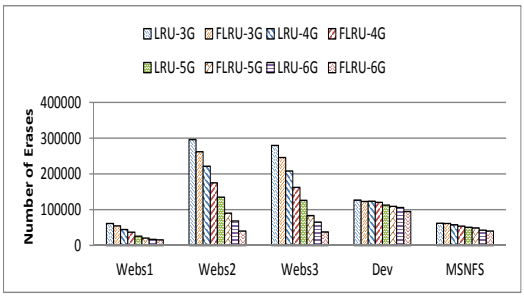


(b) ARC-15

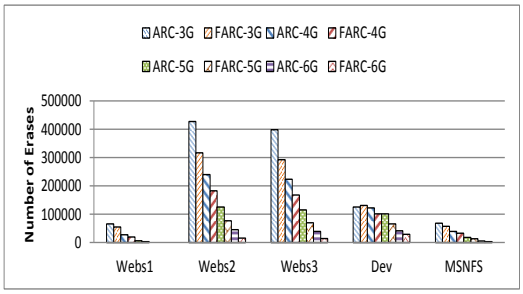
Fig. 5. (a)Improvement of geometric means of cache hit ratios of FLRU and FARC with different over-provisions, baselines are normal LRU and ARC algorithms, and (b)Improvement of geometric means of cache hit ratios with the increasing of cache size. The over-provision configuration here is 15%. The cache hit ratios of 3G for LRU and ARC are used as our baselines for LRU and ARC respectively, all the results in the figure are the differences with the baselines.

over-provision increasing from 15% to 35%, the geometric means of cache hit ratio improvements are around 7.3%, 13.9%, and 20.3%, respectively, when the cache capacity is 3GB. For ARC, the results are similar. There are two reasons that make our flash-aware design obtain more benefits from larger over-provisions. On one hand, larger over-provision means more additional physical flash memory space for our flash-aware algorithm to explore to get more performance promotion. On the other hand, larger over-provision also means the user-addressable physical space is reduced when the total physical capacity is fixed, which will introduce more cache misses for normal LRU and ARC algorithms. Fortunately, our flash-aware algorithm can counteract the negative effects of cache misses. Besides, when the user-addressable physical capacity is large enough, for example when the over-provision is 15% and capacity is 6GB, the benefit from FLRU and FARC is limited. The reason is straightforward, larger capacity means higher cache hit ratio and less improvable space.

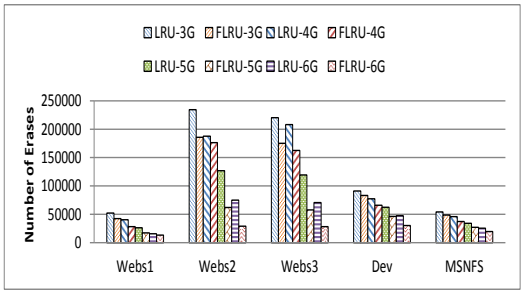
What's more, for WebSearch1-3, original ARC gets much better results than



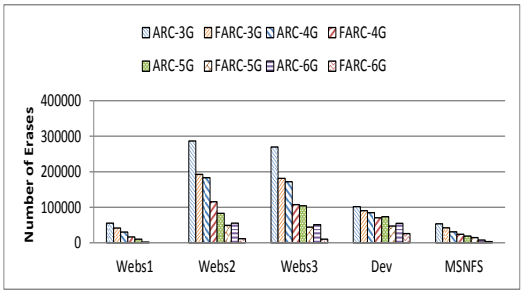
(a) LRU-15



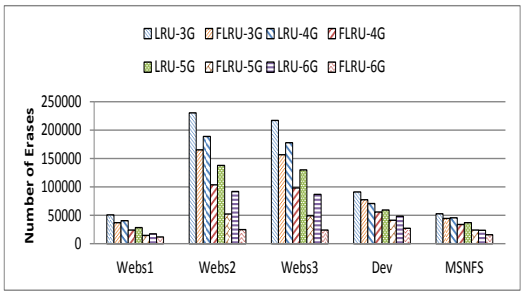
(b) ARC-15



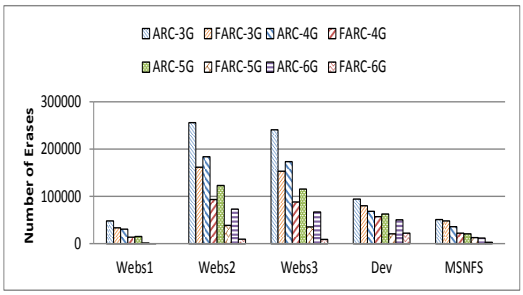
(c) LRU-25



(d) ARC-25



(e) LRU-35



(f) ARC-35

Fig. 6. Erase count collected from simulation of LRU, FLRU, ARC, and FARC. The cache capacities used here include: 3GB, 4GB, 5GB, and 6GB. Over-provisions configurations are 15%, 25%, and 35%.

LRU, but the FLRU can achieve similar or even better results compared with the FARC. Figure 5(a) shows the geometric means of cache hit ratio improvements of our FLRU and FARC algorithms under different flash capacities and over-provisions. It is very clear that LRU can acquire more benefits from our flash-aware design. The reason is that ARC has already done a very good job to cache the locality of workloads which will limit improvable space for our flash-aware design. Figure 5(b) presents the impact of cache size on geometric means of the cache hit ratio. Obviously, LRU can get more benefit from the increase of cache capacity. Since our flash-aware scheme works in a way similar to the expansion of cache capacity, LRU can get more performance promotion from our flash-aware design. Based on this observation, we may use the simple and low overhead LRU algorithm to get similar or even better performance of the ARC algorithm when our flash-aware design is utilized which will further boost the cache performance.

2.5.2 Impacts on Lifetime

In order to investigate how our flash-aware cache algorithms affect the lifetime of flash memory, we collected the erase count for all our experiments. Figure 6 presents our simulation results of erase count. The experimental results clearly show that our flash-aware cache design significantly extends the lifetime of flash memory. For example, our FLRU and FARC can at least reduce the number of erases by about 10% and 17% on average, respectively, when over-provision configuration is 15%. When the over-provision is 35%, the reduction could even reach nearly 72%. Thus the reduction of erase count is more significant than the improvement of cache hit ratios. One of the possible reasons is the write amplification effect of garbage collection processes. Before erasing a block, all the valid pages inside the victim block need to be moved to some new free space and this will introduce more additional writes. Then like the cascade

effect, more additional writes will trigger more garbage collection processes especially when used as caches which means more pressure from the upper-level requests.

What's more, for all these four cache algorithms, higher over-provision will reduce the erase count even with the same total physical cache size. Although, higher over-provision means less user-addressable physical space and lower cache hit ratios for normal cache algorithms. Even for our flash-aware cache algorithms, the cache hit ratios will be a little bit lower with higher over-provision as the over-provisioning part will not always be filled with evicted data. There are two factors that affect the number of erase for a flash cache with a fixed total capacity: cache hit ratio, and over-provision. On one hand, lower cache hit ratio will introduce more writes to bring the missed data into the flash cache. On the other hand, higher over-provision gives the flash memory more space to delay and reduce the number and the overhead of garbage collection processes. From our experimental results, the reduction of erase count from higher over-provision counteracts the penalty from the reduced cache hit ratio.

From Figure 6, we also find that ARC and FARC suffer more erase operations compared with LRU and FLRU even under the cases in which they can achieve higher cache hit ratios. For instance, when the capacity is 3GB and the over-provision is 15%, for WebSearch2, the cache ratios for LRU and ARC are 41.87% and 50.4%, but the LRU suffers less than 300,000 erase operations, while ARC suffers more than 400,000 erase operations. We believe this is due to the multi-queue architecture of the ARC algorithm. Unlike LRU, ARC is divided into four LRU queues: T1, T2, B1, and B2. Cache entries will be moved among these queues, and also cache evictions can happen in both T1 and T2 based on the current cache condition. This multi-queue architecture may generate more data fragmentation which brings the mixture of valid and invalid pages within the same block. And this kind of data fragmentation

will introduce higher overhead for garbage collection processes since more valid pages need to be migrated.

2.6 Summary

In this chapter, we propose a novel flash-aware cache design. One of flash memory's most important properties is out-of-place update. When a flash memory is used as cache, cache evictions will generate superseded but still accessible data due to the out-of-place update property. Our flash-aware cache design takes advantage of these superseded but still accessible data to improve the performance and prolong the lifetime of flash cache. To evaluate the benefits of flash-aware cache design, we implemented the normal LRU, normal ARC, flash-aware LRU (FLRU), and flash-aware ARC (FARC) cache algorithms on the Disksim simulator with SSD extension. Our simulation results demonstrate that our flash-aware cache can improve the cache hit ratio by up to 28%, reduce the average response time by up to 40% with higher performance stability, and alleviate the lifetime limitation of flash cache by reducing the erase count by up to more than 70%.

CHAPTER 3

ZERO-MIGRATION GARBAGE COLLECTION SCHEME FOR FLASH READ CACHE

3.1 Introduction

NAND Flash-based Solid State Disks (SSDs) have been deployed in a wide range of application scenarios including the portable devices, laptops, and high performance computing systems due to many attractive technical merits, such as low power consumption, light weight, shock resistance, sustain hot operation regimes, and extraordinarily high performance for random read access. But currently its still too costly to entirely replace the hard disks with SSDs, therefore SSDs are widely used as disk caches. Despite all these attractive merits, SSDs suffer from several inherent limitations, especially the limited erase cycles. Each flash block could only be erased limited cycles, after which the block will be unreliable and marked as bad. The lifetime issue becomes more serious with the shrinking of process geometries and adoption of MLC and TLC technologies. For TLC-based SSDs, each block could only sustain thousands of erase operations.

Garbage collections are internal activities of SSDs to reclaim the invalid pages generated by the out-of-place updates. A typical garbage collection process involves two steps to reclaim a victim block: data migration and erase. During the data migration process, all the valid pages in the victim block need to be migrated to new free locations before erasing the whole block, which will introduce N_{valid} additional page read and write operations. The data migration process have impacts on both the performance and endurance of SSDs. First, data migration process will introduce

extra latency and block the whole plane or package from servicing the requests from the outside. Moreover, the extra write operations incurred during the data migration processes will introduce more garbage collection and erase operations, therefore hurt the lifetime of SSDs. To reduce the cost of garbage collection process, we propose a new zero-migration garbage collection policy for Flash-based read cache, which eliminates the data page migration process and erases the whole victim block directly. The zero-migration garbage collection design are based on two observations. First, When flash memory is used as read cache, all the data in the flash memory has a backup in the write buffer or lower level storage device. Hence, the removal of valid data migration will never result in loss of data. Second, flash cache receives more pressure from the upper level requests. Therefore, any additional write operations during the garbage collection process will trigger more extra garbage collection processes in the future which will hurt both the performance and lifetime of flash cache.

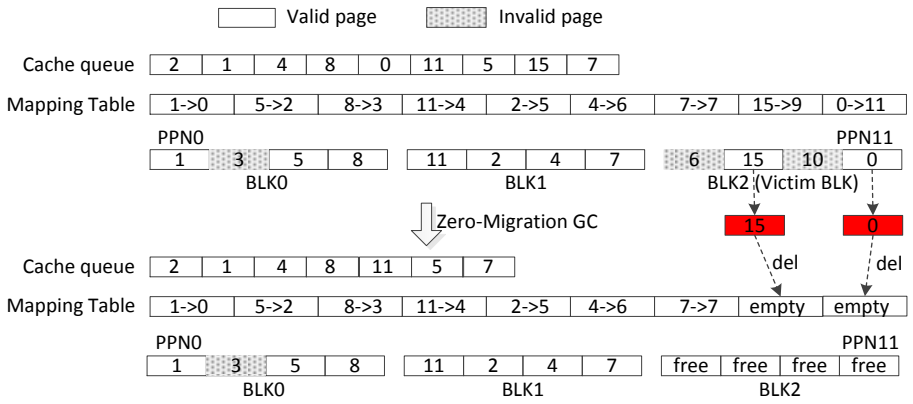


Fig. 7. An example to illustrate the working flow of our zero-migration garbage collection design.

3.2 Related Work

When SSDs are deployed as caches, endurance could be a big concern. Many previous work proposed solutions to prolong the endurance of SSD caches. Liu et al.

proposed DuraCache [52] to improve the endurance of MLC SSD based cache. When SSDs are deployed as write-through caches in data centers, DuraCache deals with the uncorrectable errors in SSD caches as cache misses to prolong the lifetime of SSD caches. Moreover, more ECC parities will be added to the data to provide stronger error tolerability when SSDs reach the wearout threshold. PLC-Cache [20] proposed by Liu et al. targets the deduplication-based primary storage system. When SSDs are deployed as caches atop a deduplication storage system, PLC-Cache achieve the lifetime extension by filtering out the unfrequent and unpopular accesses. Huang et al. proposed LACR [53] to extend the endurance of SSD cache by filtering our seldom accessed blocks and avoiding unnecessary cache replacements. Xia and Xiao proposed an Flash-aware cache design [26, 54] by leveraging the out-of-place update property to enhance the lifetime of SSD-based read cache.

3.3 Design and Implementation

Garbage collection algorithms are one of the key factors that will affect both the performance and lifetime of flash memories. In this section, we describe a new garbage collection method which aims to further improve the lifetime of flash-based read cache. Traditional garbage collection process consists of two parts: valid data migration and flash block erase. The data migration process will introduces extra N_{valid} read and write operations, which hurts both the performance and lifetime of SSDs. However, when SSDs are used as read caches, all the data inside SSDs will have exact backups in the hard disks or write buffers. Therefore, we could aggressively skip the cost valid data migration processes and directly erase the victim block without causing any data loss, which is the main motivation of our zero-migration garbage collection scheme. Figure 7 is a simplified example to show the working flow of our zero-migration scheme. We assume there three victim block candidates, where each block has four

flash pages. To perform zero-migration garbage collection, the FTL selects a victim block using a specific garbage collection policies like the greedy algorithm, which is one of the most popular garbage collection algorithms and will be used in this work. In this example, $BLK2$ has the largest number of invalid pages and will be selected as the victim block. Then our zero-migration garbage collection policy will erase the victim block directly without migrating the valid pages inside the victim block, which are page 15 and page 0 in our example. Since the data residing in the valid pages in the victim block will not exist anymore after the whole victim block being erased, so we need to update the address mapping table and cache queue for synchronization. For all the valid pages inside the victim block, the corresponding entries in the cache queues will be deleted and the mapping information in the mapping table need to be invalidated. Moreover, we also integrate our zero-migration garbage collection scheme with our flash-aware cache design presented in the previous chapter.

Although aggressively remove all valid pages during garbage collection processes may have some negative impacts on the cache hit ratio, the overall performance like the average response time might be unaffected or even improved due to the reduction of garbage collection processes. Besides the basic zero-migration garbage collection design depicted above, we also implement a variant named conditional zero-migration design as a comparison which only deletes the relatively cold pages but keeps the hot pages. In our conditional zero-migration design, we treat the entries in the tail half of the cache queue as cold that will be removed during the garbage collection process, while entries in the head half of the cache queues are regarded as hot and will be migrated to other free locations during the garbage collection process. The result shows the conditional zero-migration garbage collection scheme can only achieve marginal improvement of the cache hit ratio (within 1%) with even worse average response time and limited extension on the lifetime when compared with the

original zero-migration scheme. Therefore, we only present and analyze the result of the original zero-migration garbage collection scheme in result section.

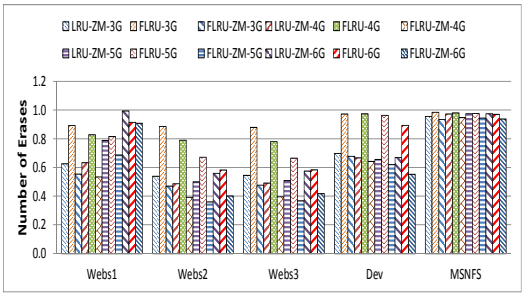
3.4 Experimental Methodology and Results

To evaluate the efficiency of our proposed design we integrated our zero-migration garbage collection scheme with LRU and ARC which are two popular cache replacement algorithms. For the other cache replacement algorithms, they can be easily tailored and integrated with our design. We used Disksim with an ssd extension as our simulator. Five realistic workloads: WebSearch1, WebSearch2, WebSearch3, DevDivRelease, MSNFS are used in our evaluation. WebSearch1, WebSearch2, and WebSearch3 were collected from popular search engine and nearly all the requests are read requests [51]. DevDivRelease and MSNFS are released by Microsoft [50].

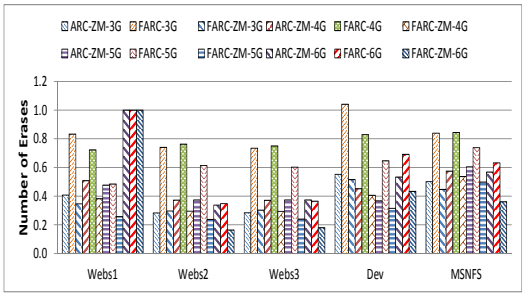
3.4.1 Impacts on Lifetime

Figure 8 shows the results of normalized number of erase operations when our zero-migration garbage collection policy is integrated with the normal and our flash-aware cache algorithms. The results strongly demonstrate that our zero-migration garbage collection scheme can significantly reduce the number of erase operations for both the normal and flash-aware cache algorithms. For normal cache, our zero-migration garbage collection scheme can reduce the erase count by up to about 72%. The reduction of erase count from the combination of flash-aware design and zero-migration garbage collection scheme could even reach nearly 90%.

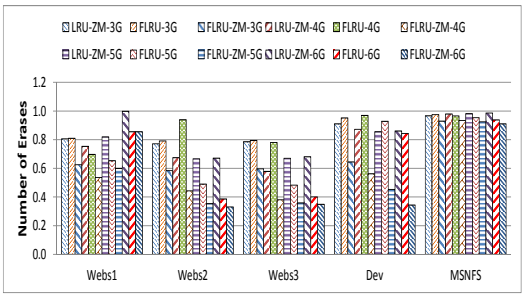
One observation from the result is that the benefit of our zero-migration garbage collection scheme on the lifetime decreases with the increasing of the over-provision for both the normal and flash-aware cache algorithms. The reason is that higher over-provision means more space to delay the garbage collection processes and improve



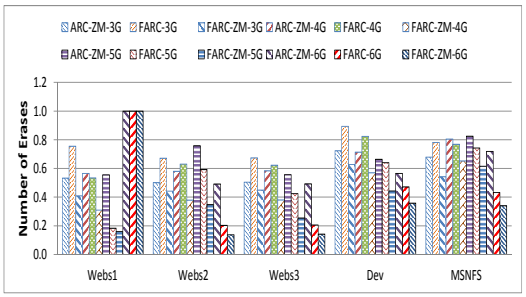
(a) LRU-15



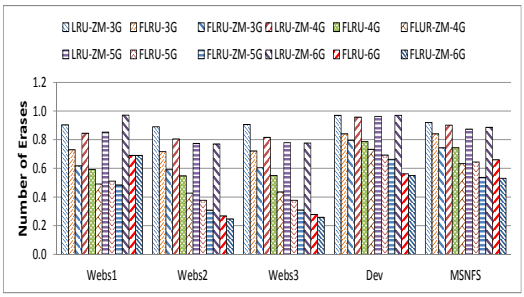
(b) ARC-15



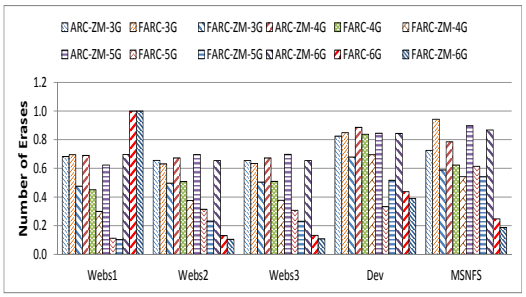
(c) LRU-25



(d) ARC-25



(e) LRU-35



(f) ARC-35

Fig. 8. Normalized erase count collected from simulation of LRU, FLRU, ARC, and FARC with the integration of zero-migration GC scheme. The cache capacities used here include: 3GB, 4GB, 5GB, and 6GB. Over-provisions configurations are 15%, 25%, and 35%.

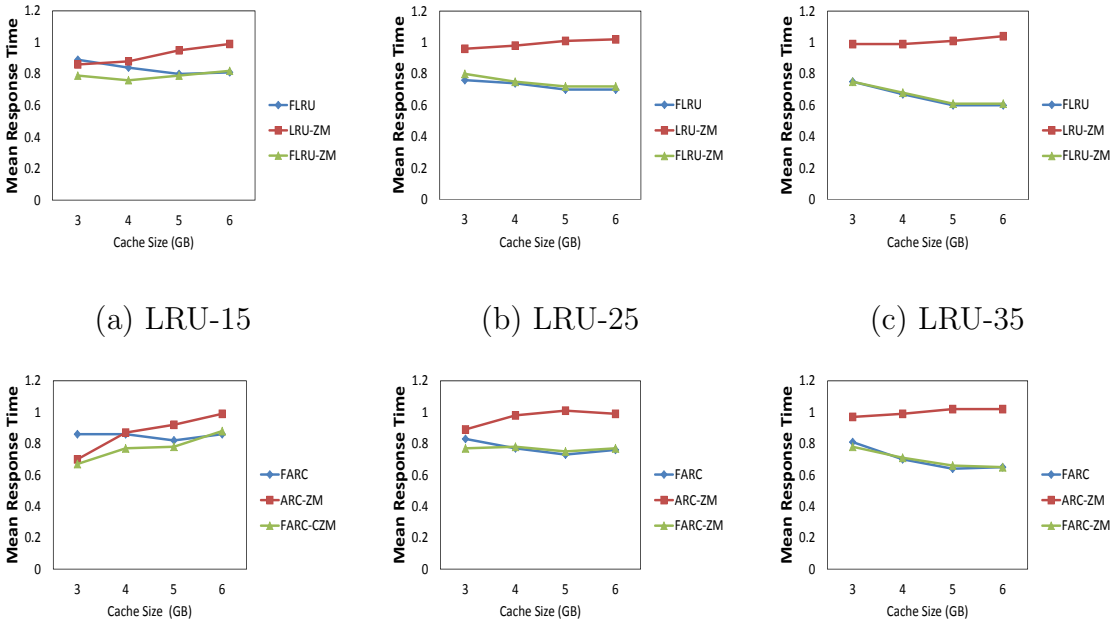
the garbage collection efficiency. On the contrary, the benefit of our flash-aware design is positively related to the over-provision. When the over-provision is 15% and 25%, our zero-migration garbage collection scheme can gain more benefit over the flash-aware design from the perspective of lifetime extension. While, when the over-provision reaches 35%, the flash-aware design can more significantly improve the lifetime of flash cache. Experimental results show that the combination of flash-aware cache and zero-migration garbage collection works well for all these different over-provisioning configurations. Therefore, our flash-aware design and zero-migration garbage collection scheme complement each other in their effect of lifetime extension. What's more, ARC and FARC gain more benefit from our zero-migration garbage collection scheme since they have lower garbage collection efficiency which we have concluded from the previous chapter.

3.4.2 Performance

Table 4 lists the difference of the geometric means of cache hit ratios after the adoption of our zero-migration garbage collection policy. A negative value means the decreasing of cache hit ratio, while a positive value implies an promotion of cache hit ratio. From the results, we find that the impacts on the cache hit ratio due to our zero-migration garbage collection scheme is negligible. In most circumstances, the drop of cache hit ratio is less than 2% or even 1%. Even in the worst cases, the loss of the average cache hit ratio is still below 4%. What's more, the average cache hit ratio could be increased by more than 2% under some special cases. We believe that the results could be explained from two aspects. On one hand, our zero-migration garbage collection scheme will reduce the number of available data to serve the coming requests which will result in lower cache hit ratios. On the other hand, the zero-migration garbage collection scheme has the potential effects to remove the

Table 4. Difference of geometric Mean of Cache Hit Ratio with the integration of zero-migration GC scheme

	OP15	OP25	OP35
LRU-ZM-3G	-0.003802	-0.000567	-0.00033
FLRU-ZM-3G	-0.017799	-0.015657	-0.006274
LRU-ZM-4G	-0.018112	-0.011183	-0.004593
FLRU-ZM-4G	-0.026855	-0.023323	-0.01515
LRU-ZM-5G	-0.022802	-0.013943	-0.009183
FRLU-ZM-5G	-0.022093	-0.015644	-0.010069
LRU-ZM-6G	-0.013882	-0.008068	-0.01069
FLRU-ZM-6G	-0.01225	-0.005285	-0.005426
ARC-ZM-3G	-0.011692	-0.010634	-0.013693
FARC-ZM-3G	-0.019931	-0.011394	-0.008574
ARC-ZM-4G	-0.03558	-0.027149	-0.007077
FARC-ZM-4G	-0.033951	-0.024358	-0.016056
ARC-ZM-5G	-0.010991	-0.0111	-0.014651
FARC-ZM-5G	-0.01116	+0.027429	-0.008184
ARC-ZM-6G	-0.010894	+0.025649	-0.010973
FARC-ZM-6G	-0.007603	+0.007748	-0.001445



(a) LRU-15 (b) LRU-25 (c) LRU-35

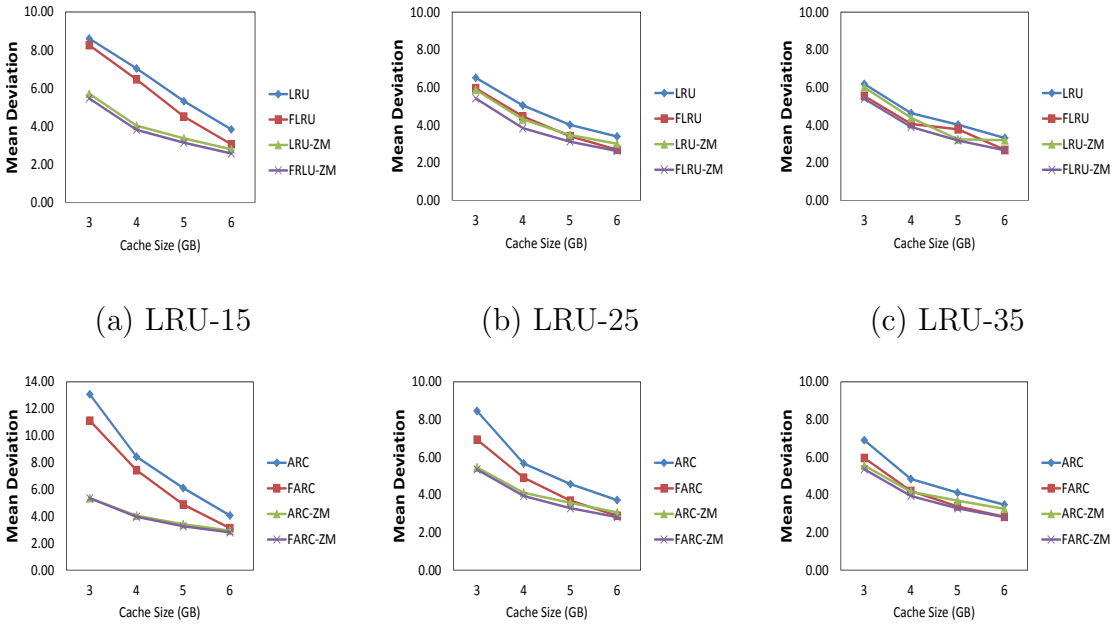
(d) ARC-15 (e) ARC-25 (f) ARC-35

Fig. 9. Normalized geometric means of the average response time of FLRU, LRU-ZM, FLRU-ZM, FARC, ARC-ZM, and FARC-ZM with the normal LRU and ARC as the baseline. The cache capacities used here includes: 3GB, 4GB, 5GB, and 6GB. Over-provisions configurations are 15%, 25%, and 35%.

cold data in advance and make room for the missed data in the future. Hence, the next time when a cache miss occurs, the missed entry can be directly inserted into the cache queues without introducing a cache eviction.

What's more, although our zero-migration garbage collection policy may lead to a little bit sacrifice of cache hit ratio, the real performance of flash cache like the average response time can be unaffected or even improved because of the significant reduction of garbage collection processes which has been verified and utilized in [43]. For flash cache, the cache hit ratio is only one of the important factors that will determine the cache performance. Another vital factor is the garbage collection processes. During a garbage collection process, the whole flash plane or package is unable to serve any requests until the end of the garbage collection process which will increase the response time of the flash cache. For each individual garbage collection process, the overhead consists of the data migration and erase operation. Thanks to our zero-migration garbage collection scheme, the number of garbage collection processes and the cost of each individual garbage collection process could be dramatically reduced due to the removal of data migration. Figure 9 presents the result of the normalized geometric means of average response time when our flash-aware and zero-migration designs are applied to the normal LRU and ARC algorithms. When our flash-aware and zero-migration designs are applied to the normal LRU and ARC algorithms separately, our flash-aware design can gain more benefit from higher over-provision configurations, while the zero-migration design works better if both the over-provision and cache size are small. For instance, when the over-provision is 35% and cache size is larger than 5GB, our flash-aware design can drop the average response time by around 40% for both LRU and ARC algorithms, but the zero-migration design can worsen the average response time by up to 2% (which is still negligible when compared with the significant improvement on the lifetime). However, when the over-provision and

cache size are 15% and 3GB, respectively, our flash-aware design can only reduce the average response time by 11% and 14% for LRU and ARC, while the zero-migration design can improve the performance by 14% and 30% for LRU and ARC. This is consistent with the conclusion in our previous subsection related to the impacts on lifetime. If we combine our flash-aware and zero-migration designs together, we can always obtain considerable reduction of the average response time which is from about 20% to 40%.



(a) LRU-15 (b) LRU-25 (c) LRU-35

(d) ARC-15 (e) ARC-25 (f) ARC-35

Fig. 10. Geometric means of the standard deviation of average reponse time for LRU, FLRU, LRU-ZM, FLRU-ZM, ARC, FARC, ARC-ZM, and FARC-ZM. The cache capacities used here includes: 3GB, 4GB, 5GB, and 6GB. Over-provisions configurations are 15%, 25%, and 35%.

Besides the average response time, another important aspect is the performance stability especially for some high-end or real time applications. In our paper, we use the standard deviation of response time to present the performance stability which is

a popular measure used to quantify the amount of variation of a set of data values. A small standard deviation value of the response time means stable performance, while a large standard deviation value indicates large fluctuations. Figure 10 shows the mean deviations of the response time. In the figure, both our flash-aware and zero-migration designs can provide more stable response times in all the cases, especially the zero-migration design which could reduce the amount of garbage collection and eliminate the data migration process at the same time. What's more, when flash-aware and zero-migration designs are combined together, further enhancement of the performance stability can be achieved.

3.5 Summary

In this chapter, we propose a novel zero-migration garbage collection design to alleviate the lifetime issue of flash-based read cache. When SSDs are deployed as read caches, all the data inside SSDs will have exact backups in the hard disks or write buffers. Based on this observation, our zero-migration design will aggressively erase the whole flash block without performing the cost valid page migration processes. For normal cache such as LRU and ARC, our zero-migration garbage collection scheme could prolong the lifetime of flash-based read cache by up to about 72% with only negligible sacrificing of the cache hit ratio. One of flash memory's most important properties is out-of-place update. Moreover, when our zero-migration garbage collection scheme is combined with our flash-aware cache design presented in the previous chapter, the extension of the lifetime could even reach up to nearly 90% with more stable performance at the same time.

CHAPTER 4

LOCALITY-DRIVEN DYNAMIC FLASH CACHE ALLOCATION

4.1 Introduction

Nowadays, both personal laptops and large data centers are equipped with large main memory to bridge the huge performance gap between the high-performance processors and slow storage systems. This large main memory can be effective for hiding the latency of read-intensive workloads [55], especially for workloads with good locality. However, for write requests, the main memory is much less effective due to its volatile nature, which means that data could be lost during power failure. Therefore, unlike read cache, which follows the popular cache algorithms like LRU and ARC, write buffering are using the write through policy or the high-low water mark to guide the data flushing processes. Whenever the number of dirty data reaches a predefined threshold (high water mark), the dirty data will be flushed back to the underlying hard disks [56]. Therefore, the write operations have been identified as the dominate traffic to the storage system and the performance-critical part of the whole systems equipped with large main memory [57, 58, 59, 60].

Nonvolatile memory techniques such as NAND Flash-based SSDs, phase change memory (PCM), spin transfer torque RAM (STT-RAM), and resistive RAM (R-eRAM) are possible solutions to improve the performance of IO intensive workloads as caches or replacements of main memory, a comprehensive summary could be found in [61]. Currently, NAND Flash-based SSDs are the only mature, widely produced, and deployed technique. STT-RAM and ReRAM are still under the research stage and currently not in volume production. Although PCM has been existed for a long

time with some real-world products, PCM has never been widely deployed due to its high production cost and high write power consumption, which is even higher than DRAM. Nowadays, energy consumption has become a critical design issue for servers and big data centers, where the low power consumption advantage of SSDs has been explored to save the energy [22, 62, 63]. Moreover, Kim et al. [64] observed that although PCM has a better physical write performance, Flash-based SSDs can achieve a better system level write performance. The major reason is that PCM has limited write parallelism due to its high write energy consumption. While, SSD can be easily scaled to obtain high write bandwidth. In this paper, we are interested in the adopting of Flash memory as a write cache for disks because of its huge capacity, low energy consumption, and high achievable write bandwidth. Previous work have presented the benefits of using SSDs as write caches for write-dominated tasks [65, 66, 67]. SSDs have several merits that make it a good candidate for write cache devices between the main memory and hard disk. First, the performance of SSDs are two to three orders of magnitude higher than the hard disk. Second, SSDs are much cheaper than DRAM. Third, SSD is nonvolatile (never losing data with power off and no cold cache misses). Due to the nonvolatile nature of SSDs, it could be used as a write back cache and follow the normal cache algorithms like LRU. Despite all the above merits, SSDs have several limitations, especially the internal garbage collection processes and limited lifetime. In order to update a Flash page in-place, the whole Flash block (usually consists of 64-256 Flash pages) need to be erased, which will introduce remarkable latency for the write operation. To hide this latency, SSD adopts the out-of-place update by relocating the new date to other pre-reserved free space and marking the old data as invalid. To support the out-of-place update, part of the SSD capacity will be reserved as the over provisional space, which is typically 7%-35% of the total SSD capacity. When the accumulation of the invalid data reduces

the available free space to a predefined threshold, a garbage collection process will be trigged to reclaim the invalid Flash pages. The garbage collection is a timing consuming process, which could significantly degrade the performance of SSDs. A higher over-provisional configuration could delay and reduce the internal garbage collection processes.

Another major concern of applying Flash-based SSDs as write caches is the limited endurance. For example, the limited program/erase cycles for SLC Flash memory is about 10,000 from the manufacturers' datasheets. In reality, the block P/E limitation is defined to meet the retention and reliability specifications in industrial standards, e.g., the JEDEC standard JESD47G.01 [68] specifies that NAND Flash blocks cycled to 10% of the P/E limitation must have a retention time of ten years, and fully cycled blocks must have a one-year retention time. However, when SSDs are used as write caches, the cold data will be quickly evicted out, while the hot data will be updated within limited reference intervals. Therefore, the requirement of retention time could be highly relaxed and the endurance of SSD write cache will be notably extended. Previous work have shown the possibility of extending the SSD endurance by the relaxing of retention time [69, 40]. Besides, the advancement of error correction codes like LDPC can also help to prolong the lifetime of SSDs [70, 71] without noticeable sacrificing of SSD write performance.

Most existing advanced cache algorithms like LIRS [72] and ARC [44] all target the traditional cache devices such as DRAM and SRAM, and use the cache hit ratio as the performance metric. Even the latest cache optimizations, which target the SSDs, still try to capture the maximum cache hit ratio [53, 73]. However, due to the internal garbage collection processes, blindly try to maximum the cache hit ratio may lead to suboptimal performance as showed in [54, 43]. For a SSD write cache, there is a compromise between the cache space and over provisional space to obtain the

optimal cache performance. On one hand, more cache space means higher cache hit ratio, but less over provisional space and higher garbage collection overhead. On the other hand, less cache space means more over provisional space and lower garbage collection cost, but at the same time lower cache hit ratio. Hence, how to make the best compromise between the cache space and over provisional space is of crucial importance to obtain the optimal cache performance.

Miss ratio curve (MRC) is a powerful performance metric representing the relationship between the cache miss ratio and cache capacity. Previously, the MRC is relegated as an offline modeling due to the extremely high memory and computing resource demands. However, recent advances like [74, 75] make it possible to generate a lightweight and continuously-updated miss ratio curves online. In this paper we will show how MRCs can be leveraged to guide the SSD capacity allocation between the cache space and over provisional space to achieve the optimal write cache performance.

The rest of this paper is organized as follows. In Section II, we describe the background on SSD and MRCs. Section III presents the details of our proposed dynamically SSD allocation scheme. Section IV presents the evaluation methodology and the experimental results. The related work is included in Section V. Section VI presents concluding remarks.

4.2 Background and Motivation

4.2.1 Miss Ratio Curves

The reuse distance (reuse distance is defined as the number of distinctive data elements accessed between two consecutive uses of the same element) of workloads is of prominent importance for the performance prediction and optimization of stor-

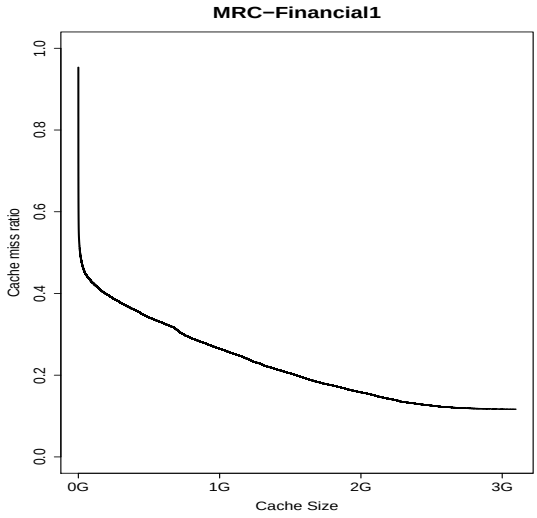


Fig. 11. Miss ratio curve of Financial1.

age and CPU caches. Miss ratio curve (MRC) is the visualized representation of the reuse distances of the workloads. Figure 11 shows an example of the miss ratio curve of Financial1, where x-axis is the cache size and y-axis is the cache miss ratio. The figure shows that MRC is a diminishing curve (the cache miss ratio decreases or stays the same with the increasing of the cache size). MRC could have several important applications. First, MRC can be used as an off-line optimal cache performance analysis like MIN [76]. Second, MRC of workloads could be used to predict cache performance in future. Third, the MRC can help the system administrator determine the size of the cache needed to meet the system performance requirement. Finally, for a system running multi workloads concurrently, an automated cache manager can generate separate MRC for each individual workloads and leverage the MRCs to optimize the cache space allocation among all these different workloads to achieve the optimal system performance [77, 78]. Although MRC is a powerful tool and has many vital application values, precise MRC measurement in the past requires $O(NlogM)$ time and $O(M)$ space for a trace of N accesses to M distinct elements

[79]. The expensive memory and computing resources overhead has restricted MRC to the offline applications. Thanks to the recent advances like SHARDs [74] and AET [75], which reduce the memory overhead to $O(1)$ for the fixed-size MRC construction scheme and $R * O(M))$ for a fixed-rate scheme, where R is the sampling rate. Besides, SHARDs and AET reduce the computing overhead of constructing the whole MRC for a trace of N accesses to $O(N)$, which means the computing overhead of processing each individual request is merely $O(1)$. Moreover, Niu et al. proposed a parallel algorithm to compute the MRC, which could further reduces the computing overhead by 13-50 times. All the above mentioned advancements have removed the offline restriction of MRC and made it possible to be deployed as an powerful online workloads analysis tool. For example, variants of SHARDS [74] have been implemented and deployed as a key component in real prototype implementations to help the data cache management in big data centers [80, 81].

4.3 Performance Modeling

Equation (4.1) shows the definition of the over-provision, where C_{user} and C_{total} is the capacity for caching data and the total capacity of SSD, respectively. Equation (4.2) gives the average cost of garbage collection, where U is the average utilization of each block (valid pages ratio) during garbage collections. Therefore $U * N$ if the total number of Flash pages needed to be migrated, where N is number of pages per block. During valid page migration, a valid page should be first read from its physical location and then rewritten to other free space. Hence, the total cost of valid pages migration is cost of $U * N$ Flash reads and writes. The total GC cost is valid page migration cost plus the following block erase cost as depicted in Equation (4.2). $Cost_{fr}$, $Cost_{fw}$, and $Cost_{erase}$ are the Flash read, write, and block erase overheads,

respectively.

$$OP = \frac{C_{cache}}{C_{total}} \tag{4.1}$$

$$Cost_{gc} = U * N * (Cost_{fr} + Cost_{fw}) + Cost_{erase} \tag{4.2}$$

Each victim block can obtain extra $(1-U)*N$ free pages after garbage collection, so the real average Flash write cost can be depicted by Equation (4.3), which evenly splits the garbage collection overhead to the extra $(1-U)*N$ free pages. If we assume that the valid pages are evenly distributed among the Flash blocks, then U equals OP. However, due to the skewness of the real-world workloads and different garbage collection policies adopted inside SSDs, there could be striking difference between U and OP. Some previous work tried to modeling the relationship between U and OP through a static equation [82], which can not work for all different workloads and SSD configurations. In this paper, we propose to get the U dynamically with training processes according to the changing of the workloads, which will be presented in Section V in detail. For a write hit in the SSD cache, we only need to write the new data to the SSD cache and invalidate the old-version of the data. While, for a write miss, a cache entry will be evicted out and written to the Disk at first, which involves a Flash read and Disk write operations. Then the new data will be inserted into the SSD cache with an additional Flash write operation. Equation (4.4) shows the whole latency of this hybrid storage system. What's more, Equation (4.4) could be further transformed into Equation (4.5), where MR is the miss ratio equals $1 - HR$.

$$Cost'_{fw} = Cost_{fw} + \frac{Cost_{gc}}{(1-U)*N} \tag{4.3}$$

$$Latency = HR * Cost'_{fw} + MR * (Cost_{fr} + Cost'_{fw} + Cost_{hd}) \tag{4.4}$$

$$Latency = Cost'_{fw} + MR * (Cost_{fr} + Cost_{hd}) \tag{4.5}$$

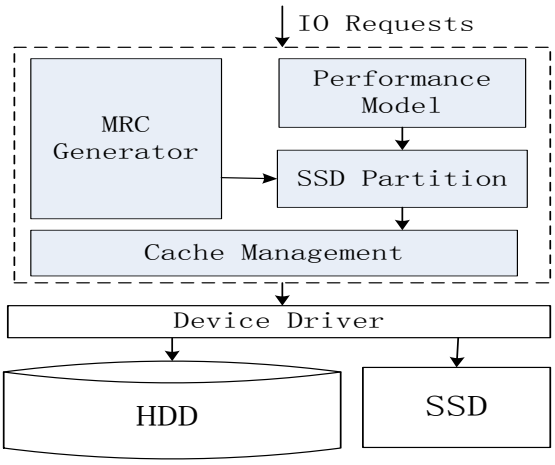


Fig. 12. System Architecture.

4.4 Design and Implementation

In this paper, we propose to utilize the workload behavior in the past to guide the following Flash cache space management by combining our derived performance model with Equation (4.5) and the MRC. Figure 12 shows the system architecture of our hybrid storage system design. Our storage system contains a hard disk as the primary storage and an SSD as the second layer write cache under the device driver. The component above the device driver is our reuse distance aware cache management, which consists of four major components: MRC generator, performance model, SSD partition, and normal cache management.

The MRC generator is deployed to dynamically generate the miss ratio curve for the incoming IO requests periodically. Our proposed scheme tries to leverage the locality of the workloads in the history to guide the Flash cache allocation in the future. Hence a time window T is introduced in our design to define how much

information in the past should be used to guide the behavior in the future and what's the frequency should we dynamically change our configuration. Initially, the MRC generator is empty and receives the incoming requests to record the reuse distances. After time window T, a miss ratio curve will be generated based on all the requests in the previous time window. The generated miss ratio curve will be leveraged to guide the allocation of the cache space. After the allocation of the cache space, a new time window starts and the miss ratio curve for the previous time window will be cleared. How to choose a proper time window T may have significant impacts on the cache performance. A large time window T may introduce too much valueless information long time ago and can't quickly responses to the changing trends of the workloads. While, a small time window T may generate a pessimistic estimation of locality of the workloads and can't fully utilize the SSD space to achieve the optimal performance. Currently, we configure the time window T as the logical time to process a fixed number of Flash page write requests. For 4 GB SSD buffer with 4 KB Flash pages, the time window T is configured as processing 8GB write requests, which equals to 2097152 Flash writes, while for a 16 GB SSD buffer, the time window is set as 40GB write requests that equal to 10485760 Flash write requests based on our observations of sensitive analysis in experimental result section. As discussed in Section II, the memory overhead is $R*O(M)$ for a fixed-rate MRC scheme. If we assume R is 0.0001 and each bucket of MRC requires 12 bytes memory space, then for our small-scale and large-scale cache configurations, the memory overheads are merely 24 KB and 120KB, respectively. Even modern laptops are equipped with 8GB or 16GB memory, let alone the big data centers, so the memory overhead of constructing the MRC is totally affordable. The computing overhead of updating the MRC for each individual request is $O(1)$. For a CPU that works at 2 GHz, the computing cost is only about 0.5ns, which is negligible compared with the latencies of accessing Flash memory and

hard disks. Therefore, the computing overhead of the MRC construction is ignored in our evaluations.

Based on the generated miss ratio curve from the MRC generator, the performance model component calculates the optimal SSD capacity allocation scheme for current workloads based on equation (4.5) that we derived in the previous section, which means how much SSD capacity should be used as the data cache and how much should be reserved for over-provision. To make our performance model work properly, an accurate estimation of the utilization U for the victim blocks during garbage collection processes is of utmost importance. Some previous work proposed to use a static equation [82] to catch the relationship between U and OP, which can not work for all different workloads and SSD configurations. In our implementation, we propose to use a training stage to get U dynamically. At the beginning, we will configure our SSD cache with different OP values and get the corresponding U. Since it is impractical to explore all the possible OP values, our training only performs on several discrete OP values (15%, 25%, 35%, 45%, 55%, 65%, 75%, 85% in our design). Based on the training results, a mapping table between the OP and the U will be builded and stored in the performance model. Since during the training stage, the SSD cache doesn't work at the its optimal configuration in majority of the time. Therefore, we will not perform the training process in ever time window, only when we detect a big shift of the workload properties like the reuse distance distributions, a retraining process will be trigged to update the OP to U mapping table.

The SSD allocator receives the optimal allocation result from the performance model and then adjusts the ratio between the SSD data cache and over-provisional space. In our current implementation, the widely used classic least recently updated policy (LRU) is deployed to evaluate our design. There are three possible cases for the cache space allocation. First, the optimal cache configuration is the same with our

current cache configuration, then nothing need to be changed. Second, the optimal data cache space is larger then the current data cache space, then we increase the maximum size of the LRU queue to the optimal value by inserting more distinctive data in the cache. Third, the optimal data cache capacity is less than the current data cache size, then we need to reduce the maximum size of the LRU queue by evicting the data from the LRU positions of the cache queue. Since the SSD is used as the write cache, all the data inside SSD are dirty. Therefore, during the data eviction process, the data will be written back to the primary hard disk. Besides, the Flash pages inside SSD that contains the data to be evicted also need to be marked as invalid so that it could be reclaimed during the garbage collection processes in the future. Here, the inherent trim command of SSD could be utilized to inform the SSD of the corresponding data evictions.

Initially, we set the preserved space as 35% as the default configuration. Then for every request, we update the miss ratio curve. Whenever the number of requests reaches a pre-defined period T, we leverage the MRC in the past period to find the optimal C_{user} that achieves the best performance. Then the cache space will be configured as this optimal value. Then the MRC will be reset and updated by the coming request in the next period. In this way, optimal C_{user} could be dynamically adjusted according to the changing trends of the workloads.

4.5 Experimental Methodology and Results

To verify the efficiency of our proposed cache design, we modified the Disksim with SSD extension [27] to implement our proposed design. Table 5 lists the main parameters of our simulator. Flash page size is 4KB and each Flash block consists of 64 Flash pages. The Flash page read and write latencies are configured as 25us and 200us, respectively. The Flash block erase cost is 1.5ms and the garbage collection

Table 5. Configuration of Our Simulator

Flash Page Size	4KB
Flash Block Size	256KB
GC Threshold	5%
Cache Size	4GB, 16GB
Page Read Latency	25us
Page Write Latency	200us
Block Erase Latency	1.5ms
Disk Access Latency	5ms

Table 6. Characteristics of I/O workloads traces

Type	Workloads	Working Set Size (KB)	Avg. Req. Size (KB)	Request Amount (GB)
Small Scale	Financial1	3.6	7.2	28.8
	Homes	5	3.9	66.8
Large Scale	Exchange	23.29	12.4	131.69
	MSNFS	23.03	11.12	74.01

threshold is set as 5%. The average hard disk access latency is defined as 5ms. The workloads used as our input are from [50] and [51]. The properties of these workloads are presented in Table 8. Basically, these four workloads could be divided into two categories based on the working set size: small scale (Financial1 and homes) and large scale (Exchange and MSNFS). Accordingly, our simulator is configured with two different SSD physical sizes: 4GB for the small scale workloads, while 16GB for the large scale workloads.

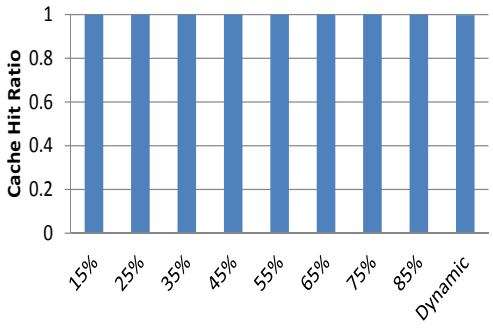
(a) Financial1

(b) Homes

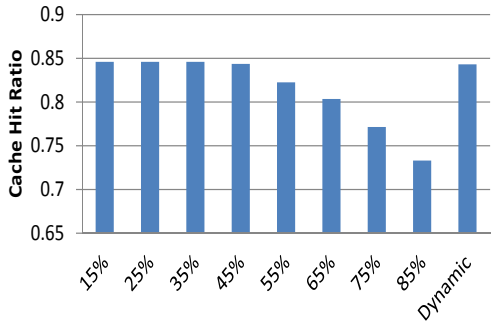
(c) Exchange

(d) MSNFS

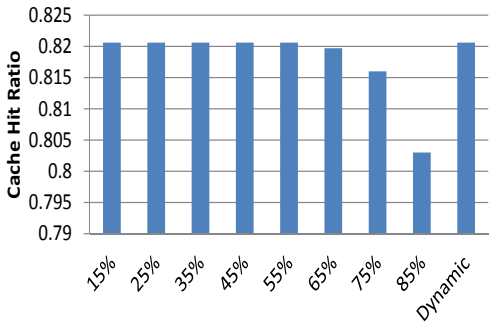
Fig. 13. Average response time of different static over provision configurations and our dynamic allocation scheme.



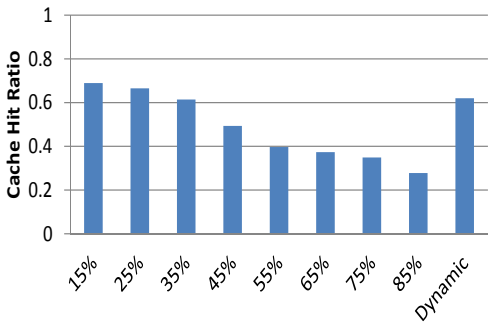
(a) Financial1



(b) Homes



(c) Exchange



(d) MSNFS

Fig. 14. Cache hit ratios of different static over provision configurations and our dynamic allocation scheme.

4.5.1 Performance

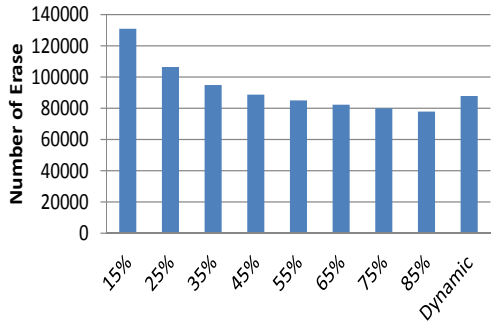
In this section, we evaluate the performance of our dynamical SSD capacity allocation design. For comparison, we also implement the static SSD cache allocation schemes, which include the following over-provisioning configurations: 15%, 25%, 35%, 45%, 55%, 65%, 75%, and 85%. For the static cache allocation schemes, the over-provision of SSD cache is configured as a fixed value like 35% throughout the whole simulation process. While, our dynamic cache allocation scheme will dynamically adjust the ratio between the data cache capacity and over-provisional space to achieve the optimal performance. Figure 13 and Figure 14 show the average response time and cache hit ratio of the static allocation scheme and our dynamic allocation scheme, respectively. From the results, we can observe that the cache hit ratio decrease or keep the same with the increasing of the over-provisioning space. The reason is as follows. Higher over-provision means more SSD space being reserved for out-of-place update and less capacity for caching data, which will lead to lower cache hit ratio. For traditional cache devices like DRAM and SRAM, lower cache hit ratio means lower cache performance. However, due to the internal garbage collection processes, higher cache hit ratio can not always guarantee higher cache performance for SSD-based cache. Figure 13 shows the cache performance variations with the increasing of the over-provision from 15% to 85%. The results show that the cache performance with the increasing of the over-provision is a concave curve, which first increases with the increasing of the over-provision and then decreases with the continuously increasing of the over-provision. When the over-provision is low, the frequent garbage collection processes are the dominant latency contributor. Therefore, increasing the over-provisioning space can reduce the garbage collection activities and improve the system performance. However, when the over-provision reaches the inflection point,

the garbage collection overhead become the secondary contributor to the system latency and the long-latency disk accesses due to the cache misses is dominated, further increasing the over-provisioning space will lead to worse system performance due to the lower cache hit ratio.

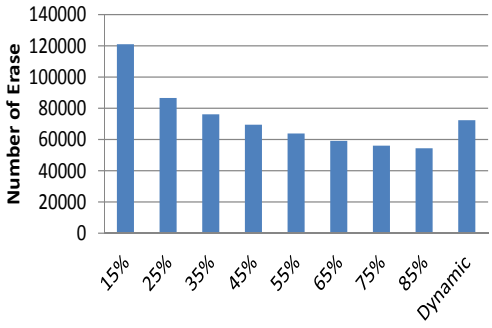
What's more, the results also show that different workloads has different inflection points or optimal static over-provisioning space, for example, the optimal static over-provisions for Financial1, Homes, Exchange, and MSNFS are 85%, 35%, 45%, and 25%, respectively. Therefore, static cache allocation scheme can not always achieve the best system performance. The rightmost bar in the figures shows the performance of our reuse distance aware dynamical cache allocation scheme. The results strongly indicates the effectiveness of our reuse distance aware dynamic cache allocation scheme, which can always obtain the system performance close to the static optimal allocation scheme for Financial1, Homes, and Exchange or even better than the static optimal allocation scheme for MSNFS.

4.5.2 Endurance

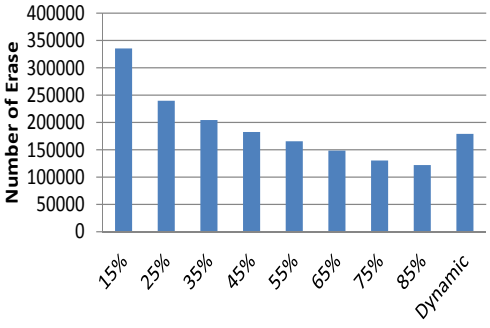
Number of erases is the widely used metric to evaluate the lifetime of SSDs. Figure 15 presents the number of erase for different static cache allocations and our dynamic allocation scheme. For all the four workloads, the number of erase will decrease with the increasing of the over-provisioning space. The reason is straightforward, higher over-provisioning space can delay and reduce the garbage collection activities and improve the garbage collection efficiency. Compared to the typical over-provision configurations that ranges from 7% to 35%, our dynamic cache space allocation scheme not only enhances the system performance, but also reduces the number of erase operations and hence prolongs the lifetime of the cache devices. Besides maximizing the system performance, our dynamic allocation scheme could also



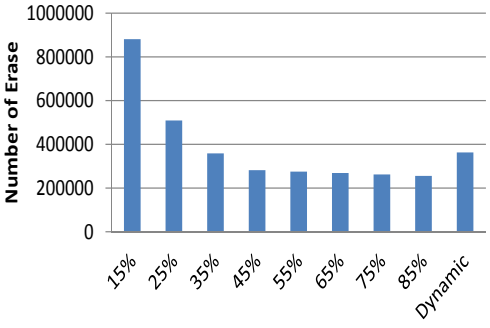
(a) Financial1



(b) Homes



(c) Exchange



(d) MSNFS

Fig. 15. Erase counts of different static over provision configurations and our dynamic allocation scheme.

be used to maximize the device lifetime without violation of the system performance requirement. For example, if the system performance requirement for running Exchange workload is 2ms, we then can relax the over-provisioning space to 65%, which can prolong the SSD lifetime by nearly 20% compared with the 35% configuration for the optimal performance.

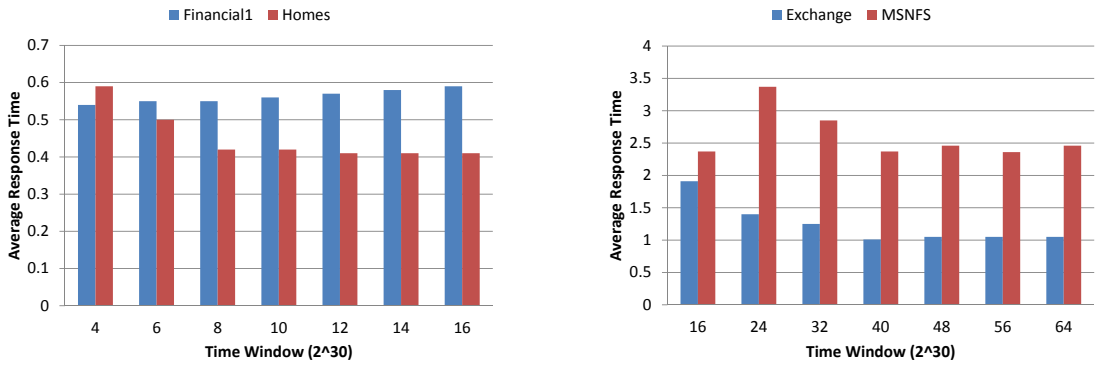


Fig. 16. Effect of different time window sizes on the cache performance.

4.5.3 Sensitive Analysis

In this section, we discuss how the variations of the time window size T will affect the system performance. On one hand, a small time window might pessimistically estimate the locality of the workloads and can't effectively leverage the cache space to get the optimal system performance. On the other hand, a large time window can bring in too much old information with little value and can't take quick action to keep up with the changing locality of the workloads. Figure 16 shows the system performance with different time window sizes. For the small cache with 4G capacity, we change the time window from 4GB to 16GB with the 2GB as the step size. While for the large cache with 16GB capacity, the range of the time window sizes is from 16GB to 64GB and the step size is 8GB. For Homes, when the time window increases

from 4GB to 8 GB, the system performance are noticeably improved due to the full exploration of the workload locality. However,the system performance are relatively stable or only with limited reduction after 8GB, which means the locality of the same workloads are relatively stable and the effectiveness of utilizing the locality of the workloads in the past to guide the cache space allocation in the future. Similar conclusions could be made for the Exchange and MSNFS. The only exception is the Financial1, when the time window size increases from 4GB to 8GB, there is no big differences of the system performance. After 8GB, increasing the time window can lead to some sacrifice of the system performance. The possible reason is the limited active working set size of Financial1, which makes 4GB is big enough to capture the locality of the workloads.

4.6 Related Work

Although cache algorithm and design optimizations are old topics and many advanced cache algorithms have been proposed to improve the cache performance like LIRS [72] and ARC [44]. However, the optimizations of these advanced cache algorithms are based on the traditional cache devices like DRAM and SRAM and use the cache hit ratio as a performance metric, which can not works consistently for SSD-based cache due to the internal garbage collection processes. When SSDs are used as caches in hybrid storage solutions, many optimizations have been proposed to improve the cache performance and lifetime of cache devices. Kgil et al. [19] proposed to take the asymmetrical read and write performance into account by splitting the Flash cache into separate read and write region with dynamically changeable ECC strength and cell density to improve reliability and lifetime of Flash memory. NetApp used Flash memory as a second layer read cache while used the NVRAM as the second layer write cache [42]. Hystor proposed by Chen et al. [65] verified the efficiency to

deploy SSD a write buffer for the performance-critical requests. Based on the highly skewness of real-world IO traces, Pritchett et al. proposed a highly-selective caching scheme for SSD cache [83]. An lazy adaptive replacement (LARC) scheme put forward by Huang et al. [53] tries to delay the replacement of the cache entry to reduce the possible cache pollution and improve the cache hit ratios. Since traditional Belady's MIN [76] only considers the cache hit ratio but not the endurance of the cache device, Cheng et al. proposed a new Flash-aware MIN cache algorithms for SSD cache [73]. However, both LARC and Flash-aware MIN are still using the cache hit ratio as the performance metric, which might not obtain the optimal SSD cache performance. In Flash-aware MIN, data that will never result in cache hit or result less than a pre-defined times cache hit will never be inserted into SSD cache to prolong the SSD endurance. Since small random writes can significantly degrade the performance of SSD cache and bring more serve write amplification problem, RIPQ [84] is proposed to aggregate small writes into a large block buffer to improve the SSD cache performance. Huang et al. [25] proposed FlexECC, which selectively replace ECC with EDC to improve the SSD-based cache performance. The kernel idea of FlexECC is that for clean data in SSD cache with backup in the hard disk, EDC will be applied. While for dirty data without backup in the hard disk, ECC will be applied to guarantee the reliability.

Beside, Oh et al. proposed APS to dynamically split the SSD cache space into to read, write, and over-provisional regions [43]. However, many ghost LRU caches with different cache sizes are needed to get the cache hit ratios under different cache capacities, which is impractical due to high complexity and memory overhead. Besides, a static equation are applied to translate the OP to U, which also makes their design inaccurate. Xia and Xiao proposed to leverage the out-of-place update property and the over-provisional space to improve the SSD read cache performance [26,

54]. In their work, they also shows that traditional advanced cache algorithms like ARC might can obtain higher cache hit ratio, but may result in worse real cache performance like the average response time.

4.7 Summary

Unlike traditional cache devices such as DRAM and SRAM, SSDs have internal garbage collection processes, which could significantly degrade the cache performance, especially when used as write cache. Previous optimizations based on traditional cache devices might not obtain consistent performance for SSD cache and even shorten the device lifetime. Therefore, how to compromise the cache hit ratio with the internal garbage collection overhead is of vital importance to obtain the optimal system performance. In this paper, we propose a locality aware dynamic SSD cache space allocation scheme by utilizing the MRC in the past to guide the SSD capacity allocation to achieve the optimal system performance for SSD write cache. The experimental result clearly demonstrate that our locality aware dynamic SSD cache allocation scheme can always achieve the performance close or even better than the optimal system performance with static allocation schemes. Besides, compared with the typical SSD over-provisioning configurations, our dynamic scheme also has the lifetime advantage.

CHAPTER 5

IMPROVING MLC FLASH PERFORMANCE WITH WORKLOAD-AWARE DIFFERENTIATED ECC

5.1 Introduction

NAND Flash-based Solid State Disk (SSD) has been widely deployed in various environments because of its high performance, nonvolatile property, and low power consumption. To continuously increase the capacity and reduce the bit cost, manufactures are aggressively scaling down the geometries and storing more bits information per flash cell [85]. However, the adoption of these technologies has inevitably resulted in degraded SSD performance especially the write performance. For example, from the previous SLC SSDs to current 2-bit MLC SSDs, the page write latency has increased from 200 us [86] to 1800 us [87], which has been identified as the major performance bottleneck [88].

A flash cell uses floating gate to store electrons and the amount of electrons will affect the cell's threshold voltage V_{th} [89]. Different V_{th} values could represent different data. These operations performed to inject and remove electrons from the floating gate are called program and erase, respectively [90]. Currently, the incremental-step pulse programming (ISPP) scheme [91] is used to perform the flash write operations. ISPP consists of a series of programming-and-verifying steps. In each step, a programming voltage $V_{program}$ is firstly applied to raise a cell's threshold voltage to $V_{current}$, then during the verification stage, the $V_{current}$ will be compared with the expected threshold voltage V_{expect}. If the $V_{current}$ is larger than the V_{expect}, then the program operation is complete. Otherwise, the $V_{program}$ will be increased by a step voltage

ΔV_{pp} and the programming-and-verifying steps will be repeated. Therefore, increasing the step voltage ΔV_{pp} could reduce the programming steps and the overall time to reach the expected threshold voltage and improve the flash write performance. However, a larger ΔV_{pp} will result in worse raw bit error rate (RBER) due to the wider threshold voltage distribution of each programmed state and less noise margin between adjacent programmed states. To compensate the increased error rate due to a larger ΔV_{pp}, a stronger ECC scheme could be applied to provide stronger error correction capability. Currently, BCH code [92] is being widely used in NAND flash memories [93, 94, 95]. Beside BCH, the low-density parity-check (LDPC) codes [96, 97] have attracted a lot of attention in both academic and industrial communities [98, 99, 70] and are the promising substitutes of the BCH code for the future SSDs. Compared with BCH code, LDPC could provide superior error correction capability. However, the complicated and time-consuming decoding process of LDPC will inevitably worsen the flash read performance and limit its rapid adoption in the real SSD product. It has been showed in [70] that the LDPC code can increase the read latency by almost 120%, while the effects on the flash write performance is within 2%.

In this work, we propose a workload-aware differentiated ECC design to improve the write performance of 2-bit MLC flash devices. BCH code is used as our baseline ECC scheme, while LDPC is utilized as the stronger ECC to accelerate the write performance. First, we dynamically separate the read and write logical pages. For write-only pages, LDPC will be applied to improve flash write performance by increasing the step voltage ΔV_p. For write pages in the overlapped section, LDPC with low-cost write scheme will be selectively applied based on the relative write and read hotness of the logical pages. Since LDPC will dramatically degrade the read performance, a rewrite operation with BCH and normal-cost write scheme will be

performed on the logical pages whose read hotness exceed a pre-defined threshold. Our main contributions in this paper include:

- We propose a workload-aware differentiated ECC scheme to improve Flash write performance without compromising Flash read performance.
- We design and implement a lightweight read and write separator by modifying the basic multi-bloom filters, which could dynamically separate the read and write logical pages in both the spatial and temporal spaces.
- we present an efficient implementation of our design. Based on the simulations of realistic disk traces, we demonstrate that our proposed design could reduce the Flash write latency by 48% on average without sacrificing the read performance.

The rest of this chapter is organized as follows. In Section II, we describe the background and related work on flash memory. Section III presents workloads analysis. Section IV depicts the details of our proposed workload-aware differentiated ECC scheme. Section V presents the evaluation methodology and the experimental results. Section VI presents concluding remarks.

5.2 Related Work

With the adoption of small geometrics and MLC technologies, the degraded performance of flash-based SSDs is becoming a critical issue. Many schemes have been proposed to enhance the performance. Some of the previous research work is based on the trade-off between the RBER and program latency. Since the RBER increases with the retention time, Pan et al. [69] and Liu et al. [31] propose to improve the program speed by compromising the retention time requirement. Besides the retention time, RBER also depends on the program/erase cycles and the content hold by the flash cells. It is well known that NAND flash memories gradually wear out with the

increasing of program/erase cycles, and which has been exploited in [100] to improve the flash program speed. As the RBER is also depend on the content of the flash cell, Gao et al. [101] propose to apply the fast program scheme to the flash pages with low-content-aware RBER. DiffECC is proposed in [102] to partially use the high cost write scheme during the idle time to reduce the RBER, which then could be handled by weak ECC scheme, to improve the flash read performance. AGCR [103] proposed by Li et al. tries to separate the accesses into three categories: read-only, write-only, and interleaved accesses. For read-only accesses, high-cost writes will be applied to reduce the overhead of the following read requests. While for write-only accesses, low-cost writes will be used to improve the write performance. For the interleaved accesses, the medium-cost writes will be applied to reach a comprise. Among all the above strategies, only AGCR is similar to our scheme. However, AGCR uses a uniform strong ECC scheme, which is designed for the worst cases, for all the flash pages that could have different error rates due to different write scheme applied on the pages. Therefore, AGCR is far from optimal. To separate the three different kinds of accesses, AGCR need to record the access histories for all logical pages, which will consume additional memory resources, especially with the continuous increasing capacity of SSDs. Moreover, response time of IO requests not only determined by the access latency (encoding and programming for write, sensing and decoding for read) and transfer latency, but also depends on the queueing delay, which has been identified as the key factor of the IO performance in the previous research work [101, 104]. Therefore, unlike AGCR which always conservatively apply the medium-cost write scheme for the interleaved requests, our workload-ware differentiated scheme will selectively apply the low-cost write scheme to the overlapped requests to further alleviate the queueing delay and improve the performance. Besides all the above previous research work, some research work like [26, 105] leverage other special properties of

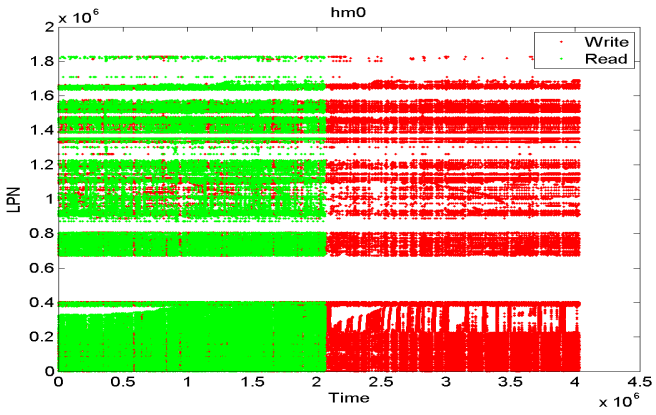
flash memory like the out-of-place updates feature to improve the flash performance.

Fig. 17. Spatial distribution of the read and write requests.

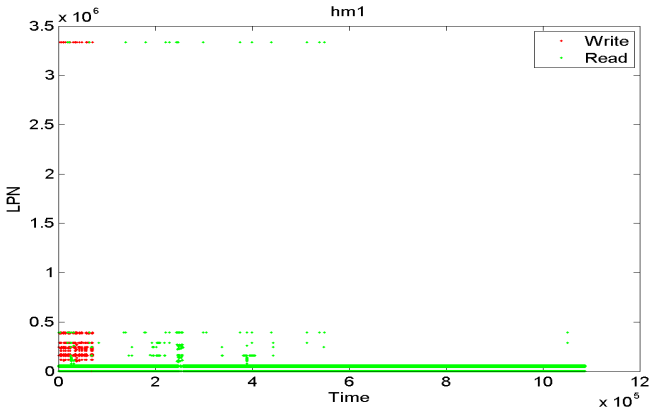
5.3 Design and Implementation

5.3.1 Analysis and Motivation

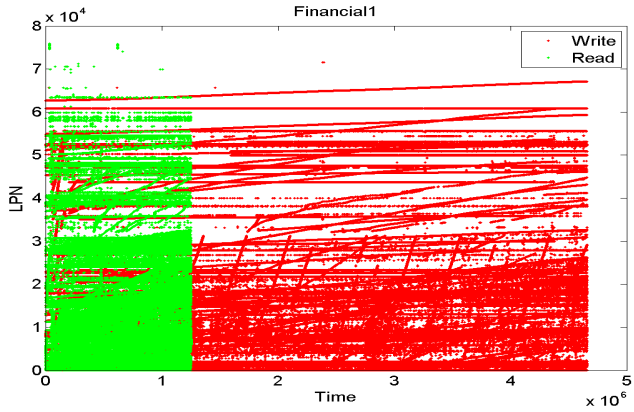
In this section, we present the study on the access characteristics of fifteen representative workloads that are mixed with both read and write requests from the Microsoft Research (MSR) Cambridge [51] and OLTP applications [50]. Figure 22 shows the ratios of write-only, read-only, and interleaved part based on their logical page numbers (here, we assume the flash page size is 8KB). The results clearly indicate that the read and write requests could be well separated based on their logical addresses with the overlapped ratios below 10% or even 5% for most of the workloads. However, hm0, hm1, and Financial1 are three exceptions whose overlapped ratios are around 42%, 28%, and 41%, respectively. Although these three workloads have a very high overlapped ratios, the read and write requests could be easily separated in the time domain. Figure 18 shows the read and write request distributions in a two-dimensional space where x-axis and y-axis are the timing space and logical ad-



(a) hm0



(b) hm1



(c) Financial1

Fig. 18. Read and write request distributions in the two-dimensional space (logical address and timing space).

dress space, respectively. In Figure 18 the red dots are the write requests, while the green dots are the read requests. From the results, we find that, for workloads with high overlapped ratio in the logical address space, the overlapping of read and write logical pages happens merely within a limited timing period, while in the other timing period, they are very well or even totally separated. From the above all results, we make the following two observations.

- Read and write requests are well separated in the logical address space for most workloads with overlapped ratios below 10% or even 5%;
- When read and write requests are highly overlapped in the logical address space, they could be easily separated in the time domain.

These two observations give us the hints to properly design the read and write separating scheme and the whole architecture of our system.

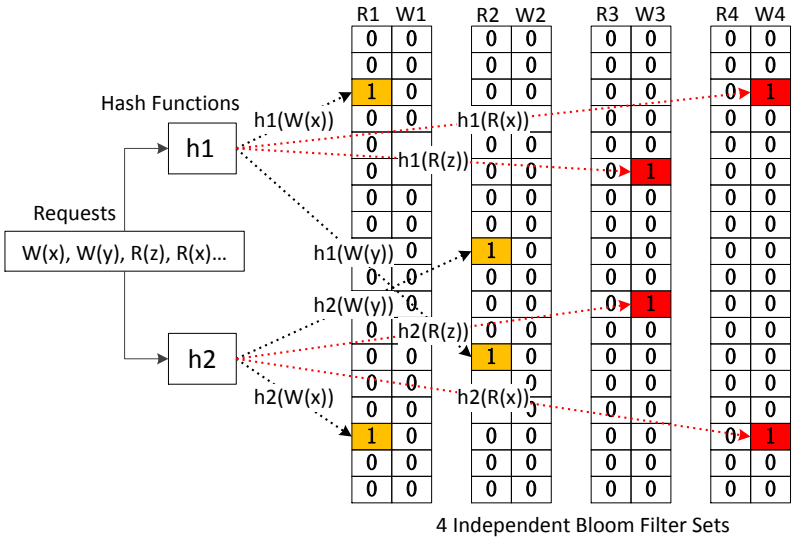


Fig. 19. Read and write separator based on multiple bloom filters.

5.3.2 Read and Write Separator

Multiple bloom filters have been utilized to perform hot data identification for flash-based storage system in [106] that captures both the frequency and recency of the requested data. A multiple bloom filters system consists of V independent bloom filters and K independent hash functions. Each bloom filter (BF) is a M bits array whose initial values are 0. K independent hash functions map the input elements to the corresponding K bit positions of the M bits BFs. To identify the hot data, the logical addresses of the write requests are used as the input for the K hash functions and will be mapped to K corresponding bit positions of the BFs. Each time, one BF will be selected from the V BFs in a round-robin manner, then the values of these K bit positions of the selected BF will be set to 1. To capture both the frequency and recency of the requests, V different weights will be assigned to the V BFs, where the highest and lowest weights are assigned to the latest and oldest updated BFs, respectively. What's more, a decay process is triggered to reset the oldest BF periodically.

Unlike the original multiple bloom filters architecture that only considers write requests, our read and write separator replaces all the independent bloom filters with read and write bloom filter pairs to accommodate both read and write requests. The read and write bloom filters within bloom filter pairs will only be updated by read requests and write requests, respectively. Figure 19 presents an example to show the architecture of our read and write separator and how it works. The read and write separator consists of two hash functions and four independent bloom filter pairs. Initially, all the bloom filter pairs are filled with 0. Then, a write request on logical page x comes. Logical page x will be mapped to two bit positions in the bloom filters and the write bloom filter in the first bloom filter pair will be updated. The second

request is a write request on logical page y. Similarly, the write bloom filter of the second bloom filter pair will be updated. The following request is a read request on logical page z, the read bloom filter of the third bloom filter pair will be updated. The forth is also a read request and the read bloom filter of the forth bloom filter pair is updated. In this way, the read hotness of a logical page could be calculated based on the read bloom filters, while the write hotness is based on the write bloom filters. By comparing the read hotness with the write hotness, the read requests could be separated with write requests as follows:

- Read only: read_hotness > 0 and write_hotness $= 0$
- Write only: read_hotness $= 0$ and write_hotness > 0
- Overlapped: read_hotness > 0 and write_hotness > 0

According to the experimental results in [106], we configure our read and write separator with 2 hash functions, and 4 read and write bloom filter pairs with 2048 bits per bloom filter. The decay period is set as 512 requests. The recency weights are set as 2, 1.5, 1, or 0.5 based on the recency of the bloom filter pairs.

5.3.3 Architecture and Working Flow

Figure 20 gives the overview of our workload-aware differentiated ECC design. In addition to the typical components of SSDs like address translator and garbage collector, a read and write separator, ECC-model selector, ECC mode bit, and a hybrid ECC system that supports both BCH and LDPC are added in our design. A 1-bit ECC mode tag is attached to each page as metadata to help the system choose the corresponding decoder for a read request.

Whenever a host write request arrives, the write bloom filter will be updated. For write requests from both the host side and background operations like garbage

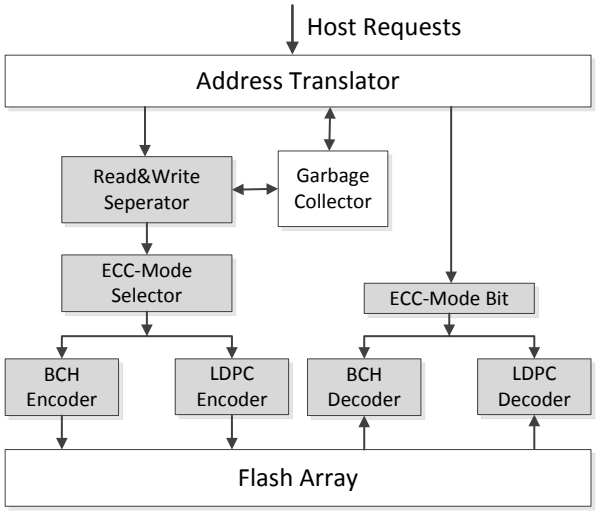


Fig. 20. Architecture of our workload-aware differentiated ECC design.

collection, the ECC-Mode selector chooses the proper ECC encoder and write scheme based on the following regulations:

- For a write-only page, a low-cost write scheme with LDPC encoder will be applied;
- For an overlapped page, if the write_hotness - read_hotness > m, the low-cost write scheme with LDPC encoder will be applied. Otherwise, the normal write scheme with BCH encoder will be applied;

While for a host read request, the read bloom filter will be updated. Then the corresponding decoder will be used to decode the data according to the ECC mode bit. Besides, if the read hotness of a LDPC-encoded logical page is larger than its write hotness (if its write hotness is larger than the read hotness, then the data has a high probability to be updated by the user requests, which makes the rewrite operation unnecessary.) and the difference between the read hotness and write hotness exceeds a predefined threshold n, then a rewrite operation with normal-cost write and BCH encoder will be issued to reduce the overhead of the upcoming read requests. The

selections of the values for m and n will affect the performance gain of our design. A larger value of m will limit the possibility to apply our low-cost write mode. While, a small value of m may hurt the read performance. For threshold n, a larger value may hurt the read performance and a small value may introduce unnecessary rewrite operations and more garbage collection processes. In the next section, we will discuss the selection of the proper m and n.

5.3.4 Overhead Analysis

The overhead of our workload-aware differentiated ECC scheme falls into three categories: memory, firmware, and hardware. The memory overhead comes from two parts: multiple bloom filters based read and write separator, and ECC-Mode bit. The multiple bloom filters used in our implementation consist of 4 read and write bloom filter pairs with 2048 bits per bloom filter. Therefore, the memory overhead of the multiple bloom filters is merely 2KB and it is independent on the SSD capacity. Besides, each flash page needs 1 bit for the ECC-Mode bit. Assuming a 64GB flash memory with 8KB pages, the memory consumption of the ECC-Mode bits is only 1MB. The firmware overhead includes 2 hash functions, and multiple bloom filter check and update. The overhead of these simple processes is negligible. To make our scheme works, the hardware need to support both differentiated program speeds and ECC codes. This differentiated architecture has been utilized and explored in previous studies [31, 102]. What's more, the rewrites for the overlapped read requests may introduce additional write operations inside SSDs and hurt the lifetime, which will be evaluated in Section V.

Table 7. Operation latency configuration

Operation	Latency (us)
BCH Read	45
LDPC Read Fast	75
LDPC Read Medium	110
LDPC Read Slow	145
Write Fast	650
Write Medium	1300
Write Slow	2600
Erase	3800

5.4 Experimental Methodology and Results

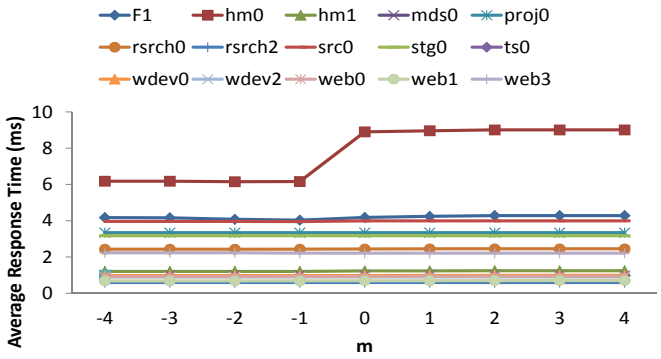
5.4.1 Experimental Methodology

We modified the Disksim with SSD extension [27] to verify our proposed design. We assume only two write schemes are supported by the device for our workload-aware differentiated ECC scheme, the normal-cost write mode and low-cost write mode that doubles the ΔV_p. Therefore, we set the normal-cost flash page write latency as 1.3 ms, low-cost write latency as 650 us, block erase latency as 3.8 ms, BCH-encoded page read latency as 45 us based on [107], and LDPC-encoded page read latency as 145 us (estimated based on [70]). For comparison, we set the latency of high-cost write, medium-cost write, and low-cost write in AGCR as 2.6 ms, 1.3 ms, and 650 us, respectively. Since the LDPC decoding latency depends the error rate of Flash pages, we estimate the average latency for the high-cost read, medium-cost read, and low-cost read for AGCR as 145 us, 110 us, and 75 us (also based on [70]), respectively. The summary of these different-cost operations is listed in Table 7. The

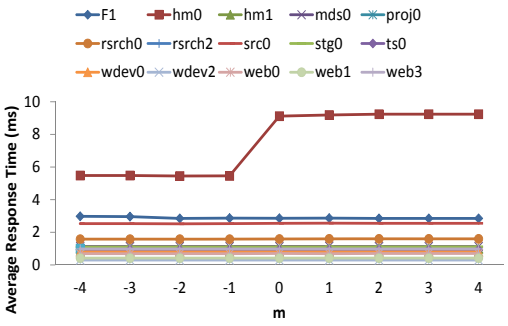
Table 8. Characteristics of I/O workload traces

Workload Name	**Addr. Space**	**Request Amount**	**Read Ratio**	**Avg. Req. Size (KB)**
Financial1	16.67GB	5334857	23.16%	15.15
hm0	13.94GB	3993316	35.5%	7.99
hm1	25.44GB	609311	95.34%	15.16
mds0	33.92GB	1211034	11.89%	9.20
proj0	16.24GB	4224524	12.48%	38.04
rsrch0	16.89GB	1433655	9.32%	8.93
rsrch2	81.65GB	207587	65.69%	4.09
src0	15.63GB	1557814	11.34%	7.21
stg0	10.82GB	2030915	15.19%	11.58
ts0	21.97GB	1801734	17.58%	9.01
wdev0	16.96GB	1143261	20.08%	9.08
wdev2	33.92GB	181266	0.1%	8.15
web0	33.91GB	2029945	29.88%	14.99
web1	67.83GB	160891	54.11%	29.07
web3	169.58GB	31380	32.03%	38.14

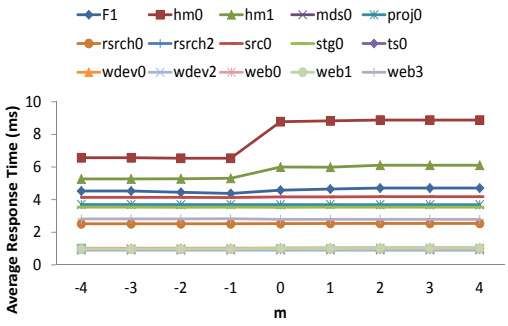
sizes of flash page and block are 8 KB and 1 MB. The SSD is configured with 8 packages. The capacity of the packages are determined by the address space of the specific workloads. The over-provision space and garbage collection threshold are set as 15% and 5%, respectively. Fifteen realistic workloads that have been analyzed in the previous section will be used in our experiments. Details of the characteristics of these workloads are depicted in Table 8.



(a) Overall performance



(b) Read performance



(c) Write performance

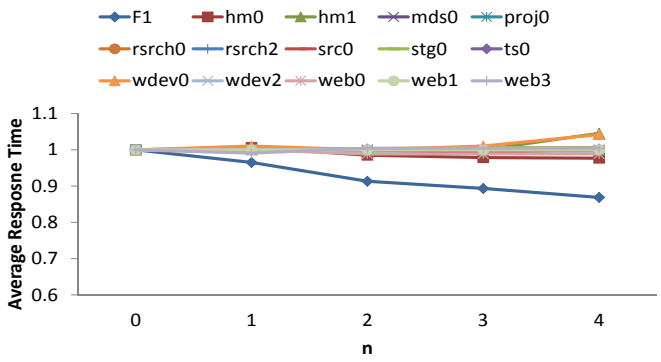
Fig. 21. Performance change with various values of m, here we fix the value of n as 2.

5.4.2 Experimental Results

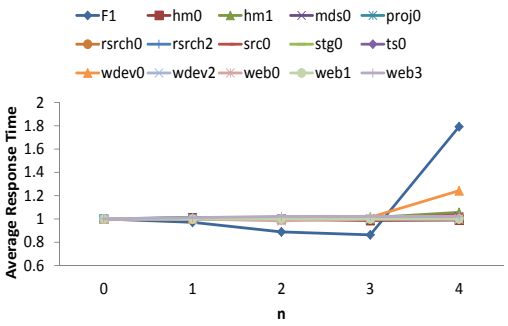
5.4.2.1 Parameters Exploration

In this subsection, we will show how different configurations of m and n will affect the performance of our workload-aware design. First, we fix the value of n with 2, which is the intermediate value of n, and varies the values of m from -4 to 4 based on the configuration of our multiple bloom filters. A positive value of m means that only when the write hotness of the request is larger than its read hotness, the low-cost write pattern will be applied to accelerate the write speed and reduce the queueing delay. While a negative value of m means aggressively apply the low-cost write pattern, even when the request has a higher read hotness. A large value of m may limit the chance to apply the low-cost writes for the overlapped requests, while a small value of m can lead to more high-cost reads. Figure 21(a) shows the overall average response time under different m configurations, where m changes from -4 to 4. The results indicate that the best overall performance can be achieved with m equals -1. Besides, for most of the workloads, changing the values of m only has marginal effects on the overall performance due to the small overlapped ratios of these workloads. One exception is the hm0, the average overall performance could be significantly improved when m changes from positive values to negative values. We believe there are two reasons for this exception. First is the relatively higher overlapped ratio, which means the decreasing of m can generate more low-cost write operations. The second is due to the fact that write operations are much more costly than reads. Therefore, reducing the write latency can effectively mitigate the queueing delay, which has been identified as the key factor of the IO performance in the previous research work [108, 104], and improve the overall performance. Figure 21(b) and Figure 21(c) presents the average read and write performance variations with different values of m, where the

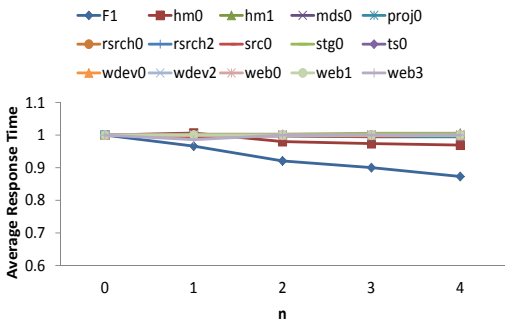
same conclusion can be made. Hence, we set the value of m as -1 as our default configuration in our following evaluations.



(a) Normalized overall performance



(b) Normalized read performance



(c) Normalized write performance

Fig. 22. Performance change with various values of n, here we fix the value of m as -1.

After setting the value of m as -1, we changes the value of n from 0 to 4. Figure 22(a) shows the normalized overall average response time with different values of n. Figure 22(b) and Figure 22(c) presents the normalized average read and write response time. The results indicate that increasing the value of n can always result in better or the same write performance. While for both the read and overall performance, only when n is less than or equal to 3, increasing the value of n can obtain better or similar results. Therefore, we set the value of n as 3 in our following experiments.

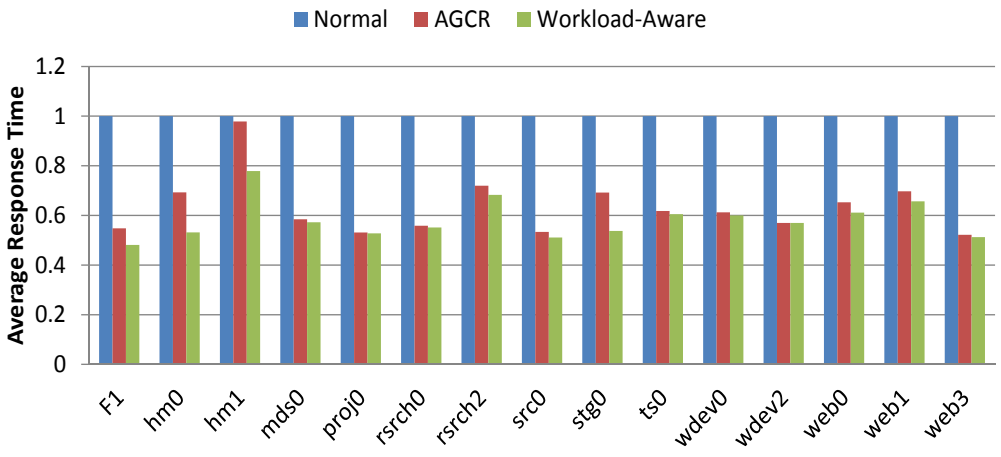
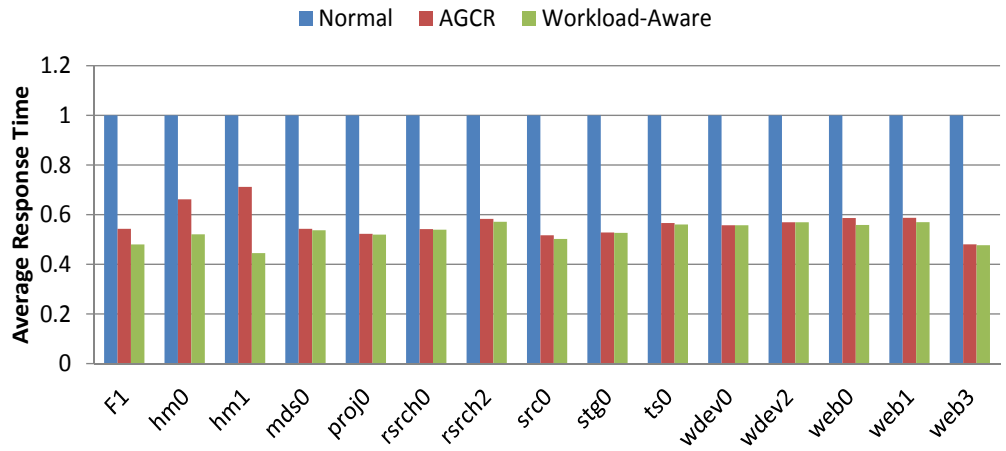


Fig. 23. Normalized overall performance comparison.

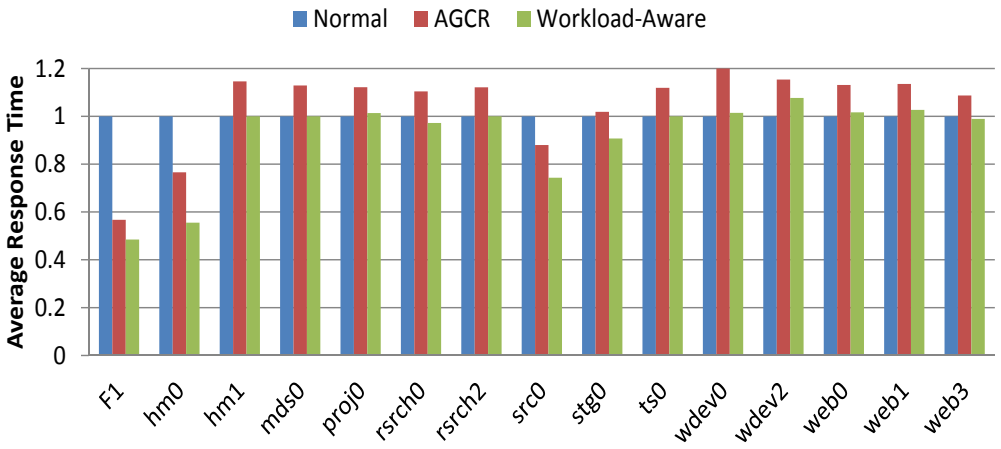
5.4.2.2 Performance

In this section, read, write and overall performance of our workload-aware differentiated ECC design are evaluated and compared with the normal case and the state-of-art work from Li et al. [103].

Figure 23 shows the normalized overall average response time of the normal, AGCR, and our workload-aware design. The result shows that both AGCR and our workload-aware design can significantly improve the overall performance, especially for the workloads with high write ratios like Financail1, mds0, proj0 and so on, where the improvement from AGCR and our workload-aware design could reach up to 45% and 52%, respectively. But for workloads with relatively higher overlapped ratios like Financial1, hm0, and hm1, our workload-aware design can show remarkable advantage over AGCR. For instances, our workload-aware design could improve the overall performance for Financial1, hm0, and hm1 by about 52%, 47% and 32%, respectively. While the overall performance enhancements for Financial1, hm0, and hm1 from AGCR are limited to about 45%, 30%, and 2%. There are three reasons that make



(a) Normalized write average response time

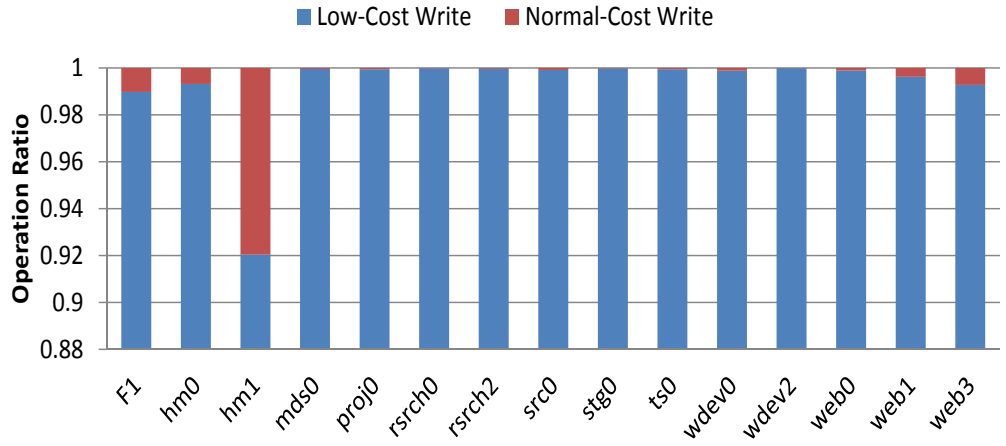


(b) Normalized read average response time

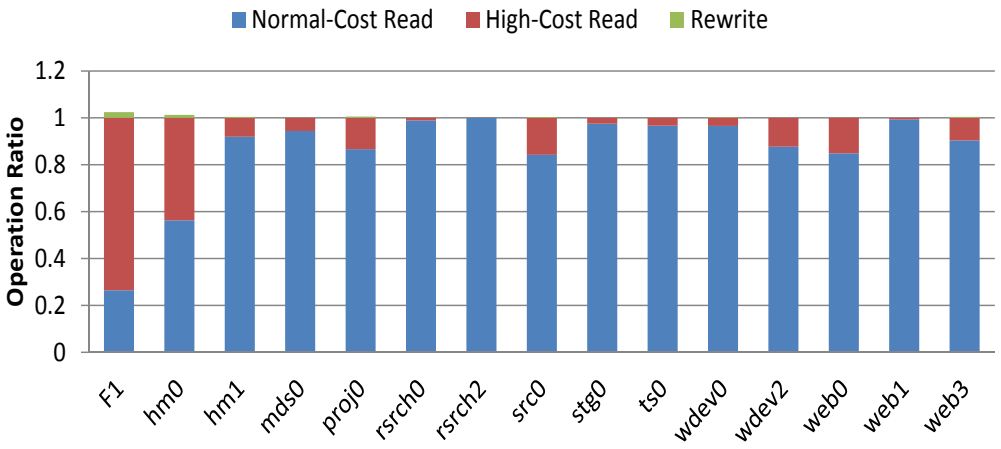
Fig. 24. Normalized write and read performance comparison.

our workload-aware design work better for workloads with higher overlapped ratios. The first is the adoption of the hybrid ECC architecture, which takes the advantage of both the low read latency of BCH encoded data and strong error correcting capability of LDPC code. Second, unlike the AGCR scheme that always apply the medium-cost write for the overlapped requests to make a compromise, our workload-aware scheme could aggressively apply the low-cost writes to these overlapped requests to reduce the queueing delay from these costly write operations and improve the overall performance. Finally, the introduction and proper configuration of parameter n has the potential to remove unnecessary rewrite operations and reduce the time-consuming garbage collection processes.

Besides the overall average performance, the normalized read and write response time are presented in Figure 24 to give more information in detail. For the write performance, both the AGCR and our workload-aware scheme can gain remarkable benefit (more than 40% reduction for most of the workloads). Similar to the overall performance, our workload-aware scheme shows remarkable advantage over AGCR for Financial1, hm0, and hm1. For the read requests, our workload-aware scheme can still maintain the same performance for majority of the workloads and could even prominently improve the performance by more than 40% for some highly overlapped workloads like Financail1 and hm0, which comes from the noticeable reduction of the queueing delay. For src0, although the overlapped ratio is about 10%, the total read ratio is also about 10%, which means that the read requests of src0 highly overlapped with the write requests. Therefore, the read performance can be boosted with the reduction of queueing delay from the application of low-cost write pattern. While for the read performance of AGCR, except Financial1, hm0, and src0 which can benefit from the prominent reduction of queueing delay, most of the workloads shows worse read performance even compared with the normal scheme due to the costly



(a) Different cost write ratios



(b) Different cost read ratios

Fig. 25. Distributions of different cost operations.

LDPC-based read operations.

To help us better understand the performance variations of our workload-aware design, Figure 25 shows the distributions of operations of different costs, which includes normal-cost read, high-cost read, normal-cost write, low-cost write, and rewrite. For most of the workloads, more than 80% of the operations are low-cost writes and normal-cost read, which means our workload-aware scheme can properly distinguish between read-dominated and write-dominated requests and apply the differentiated read and write patterns accordingly. While for Financial1 and hm0, our workload-aware design can aggressively apply more of the low-cost write to reduce the queueing delay and achieve the best performance. For hm1, since more than 95% are read requests, therefore more normal-cost reads is preferred by our workload-aware design. Moreover, our workload-aware design merely introduce negligible rewrite operations.

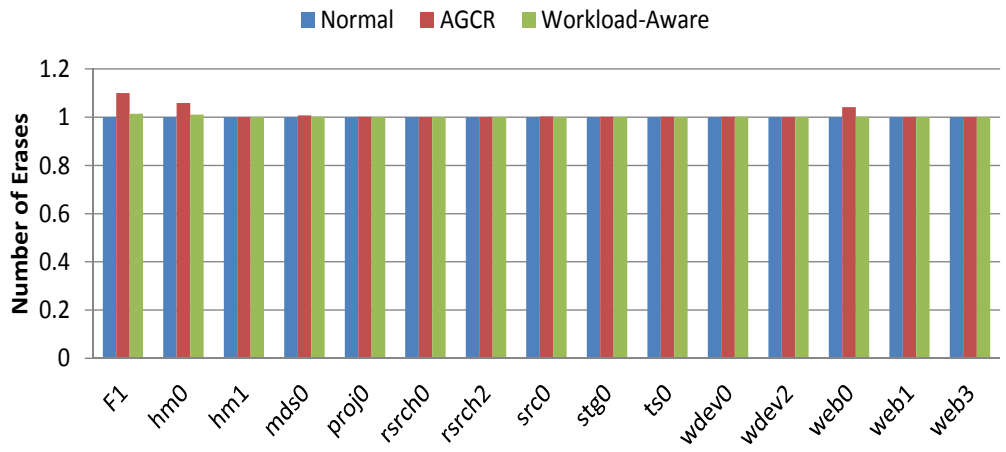


Fig. 26. Normalized number of erases.

5.4.2.3 Impacts on Lifetime

In our workload-aware scheme, rewrite operations are issued to transfer the high-cost read operations to the normal-cost read operations, which may have some negative effects on the lifetime of flash memory. Figure 26 presents the normalized number of erase operations. Compared with AGCR, our workload-aware scheme introduces less and negligible extra erase operations. For all of the workloads, the increased erase ratios from our workload-aware design are within 2%.

5.5 Summary

In this chapter, we propose a workload-aware differentiated ECC scheme to improve the SSD write performance without sacrificing the read performance. The main idea is to dynamically classify the logical pages into three categories: write-only, read-only, and overlapped part. For write-only logical pages, low-cost write with strong ECC scheme will be applied to increase the write performance. For write logical pages in the overlapped part, the low-cost writes with strong ECC will be selectively used based on their relative write and read hotness. While for any read logical pages encoded with a stronger ECC, we will rewrite them with the normal-cost write and ECC scheme if their hotness exceed a pre-defined threshold. The evaluation results show that our workload-aware differentiated ECC scheme could reduce the write and read response times by 48% and 11% on average, respectively. Even compared with the latest previous work, our workload-aware design can still gain about 4% write performance and 11% read performance improvement.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

In this dissertation, we make the following contributions to improve the performance and endurance of NAND-based SSDs:

1. We propose a novel flash-aware high-performance and endurable cache by leveraging the special out-of-place update property of SSDs. Due to the out-of-place updates, when SSDs are used as caches, the cache eviction only removes the metadata of the cache entry from the cache queue, however the real user data still exists inside SSDs until the whole flash block being erased. Our flash-aware cache design tries to utilize the evicted but still available data to improve the cache performance and prolong the lifetime of Flash-based cache. The experimental results demonstrate that our flash-aware cache design improve the performance by up to 40% and prolong the endurance of SSD cache by up to more than 70%.

2. We propose a new zero-migration garbage collection scheme to reduce the overhead and frequency of internal garbage collection processes for flash-based read cache. Typical garbage collection processes requires the valid data migration processes to move the valid pages to other free location before erasing the whole victim block, which can introduce extra write operations and hurt the endurance of SSDs. Our zero-migration tries to aggressively erase the whole victim block without performing the cost valid data migration processes based

on the following observation: when SSDs are used as read cache, all the data inside SSD cache will have exact backup in the hard disks or write buffering, therefore aggressively erase the victim block by skipping the data migration processes will never result in data loss. The experimental results show that our zero-migration garbage collection scheme could prolong the lifetime of SSD cache by up to 72% without sacrificing the cache performance.

3. We propose a locality-driven dynamic flash cache allocation scheme. When SSDs are used as write caches, the traditional cache hit ratio oriented cache optimizations could not obtain consistent performance benefit and may even hurt the lifetime of SSDs due to the internal garbage collection activities. Therefore, our locality-driven dynamic flash cache allocation scheme tries to leverage the miss ratio curve to dynamically configure the SSD cache to achieve the optimal cache performance by compromising the cache hit ratio and internal garbage collection overhead. The experimental results indicate that our design could also achieve similar or even better performance when compared with the static optimal cache configurations.

4. we propose a workload-aware differentiated ECC scheme to improve the SSD write performance without sacrificing the read performance. The main idea is to dynamically classify the logical pages into three categories: write-only, read-only, and overlapped part. For write-only logical pages, low-cost write with strong ECC scheme will be applied to increase the write performance. For write logical pages in the overlapped part, the low-cost writes with strong ECC will be selectively used based on their relative write and read hotness. While for any read logical pages encoded with a stronger ECC, we will rewrite them with the normal-cost write and ECC scheme if their hotness exceed a pre-defined

threshold. The evaluation results show that our workload-aware differentiated ECC scheme could reduce the write and read response times by 48% and 11% on average, respectively. Even compared with the latest previous work, our workload-aware design can still gain about 4% write performance and 11% read performance improvement.

6.2 Future Work

Currently, for the cache optimizations, our work focus on the case where SSD-based read cache and write cache are separated due to severe interference between Flash read and Flash write. However, the architecture with separated read and write caches might introduce additional overhead to guarantee data consistency between the read cache and write cache. Moreover, the separation of read cache and write cache might lead to inefficiency of the cache capacities if the workloads are read dominated or write dominated. Therefore, as part of our future work, we are going to integrate our proposed cache optimizations in this work with the unified cache architecture and come up with solutions to alleviate the internal Flash read and write interferences.

Each individual Flash chip could be very slow. SSDs obtain high read and write bandwidths by providing rich internal channel level, package level, die level, and plane level parallelism. Therefore, how to effectively utilize the internal parallelism is of key importance to the system performance. As another part of our future work, we target improving the SSD cache performance by fully leveraging the rich internal parallelism of SSDs. For example, instead of reading and evicting one single Flash page at each time, we could read and evict multiple Flash pages belonging to different parallel units at the same time for the cache read requests and cache eviction operations to explore the internal parallelism.

PUBLICATIONS

1. Qianbin Xia, and Weijun Xiao. "Locality-Driven Dynamic Flash Cache Allocation", (Submitted to MASCOTS, 2017).

2. Qianbin Xia, and Weijun Xiao. "Improving MLC Flash Performance with Workload-Aware Differentiated ECC." Parallel and Distributed Systems (ICPADS), 2016 IEEE 22nd International Conference on. IEEE, 2016.

3. Qianbin Xia, and Weijun Xiao, "High-performance and Endurable Cache Management For Flash-based Read Caching", IEEE Transactions on Parallel and Distributed Systems, no. 1, pp. 1, 2016.

4. Qianbin Xia, and Weijun Xiao. Zero-Migration Garbage Collection Scheme for Flash Read Cache, PACT, 2015. (Poster)

5. Qianbin Xia, and Weijun Xiao. "Flash-Aware High-Performance and Endurable Cache." Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2015 IEEE 23rd International Symposium on. IEEE, 2015.

REFERENCES

[1] *Technical white paper: Solid State Drive Technology*. 2013. URL: http://h10032.www1.hp.com/ctg/Manual/c03757461.pdf.

[2] Mei-Ling Chiang, Paul CH Lee, and Ruei-Chuan Chang. "Using data clustering to improve cleaning performance for flash memory". In: *SOFTWARE-PRACTICE & EXPERIENCE* 29.3 (1999), pp. 267–290.

[3] Amir Ban. *Flash file system*. US Patent 5,404,485. 1995.

[4] Sang-Won Lee et al. "A log buffer-based flash translation layer using fully-associative sector translation". In: *ACM Transactions on Embedded Computing Systems (TECS)* 6.3 (2007), p. 18.

[5] Jeong-Uk Kang et al. "A superblock-based flash translation layer for NAND flash memory". In: *Proceedings of the 6th ACM & IEEE International conference on Embedded software*. ACM. 2006, pp. 161–170.

[6] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. *DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings*. Vol. 44. 3. ACM, 2009.

[7] Sungjin Lee et al. "LAST: locality-aware sector translation for NAND flash memory-based storage systems". In: *ACM SIGOPS Operating Systems Review* 42.6 (2008), pp. 36–42.

[8] You Zhou et al. "An efficient page-level FTL to optimize address translation in flash memory". In: *Proceedings of the Tenth European Conference on Computer Systems*. ACM. 2015, p. 12.

[9] John T Robinson. "Analysis of steady-state segment storage utilizations in a log-structured file system with least-utilized segment cleaning". In: *ACM SIGOPS Operating Systems Review* 30.4 (1996), pp. 29–32.

[10] Luojie Xiang and Brian Kurkoski. "An improved analytical expression for write amplification in NAND flash". In: *arXiv preprint arXiv:1110.4245* (2011).

[11] Peter Desnoyers. "Analytic modeling of SSD write performance". In: *Proceedings of the 5th Annual International Systems and Storage Conference*. ACM. 2012, p. 12.

[12] Werner Bux and Ilias Iliadis. "Performance of greedy garbage collection in flash-based solid-state drives". In: *Performance Evaluation* 67.11 (2010), pp. 1172–1186.

[13] Xiao-Yu Hu et al. "Write amplification analysis in flash-based solid state drives". In: *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. ACM. 2009, p. 10.

[14] Benny Van Houdt. "A mean field model for a class of garbage collection algorithms in flash-based solid state drives". In: *ACM SIGMETRICS Performance Evaluation Review*. Vol. 41. 1. ACM. 2013, pp. 191–202.

[15] Yongkun Li, Patrick PC Lee, and John Lui. "Stochastic modeling of large-scale solid-state storage systems: analysis, design tradeoffs and optimization". In: *Proceedings of the ACM SIGMETRICS/international conference on Measurement and modeling of computer systems*. ACM. 2013, pp. 179–190.

[16] Abhishek Rajimwale, Vijayan Prabhakaran, and John D Davis. "Block Management in Solid-State Devices." In: *USENIX Annual Technical Conference*. 2009.

[17] Mohit Saxena and Michael M Swift. "FlashVM: Virtual Memory Management on Flash." In: *USENIX Annual Technical Conference*. 2010.

[18] Stan Park and Kai Shen. "FIOS: a fair, efficient flash I/O scheduler." In: *FAST*. 2012, p. 13.

[19] Taeho Kgil, David Roberts, and Trevor Mudge. "Improving NAND flash based disk caches". In: *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*. IEEE. 2008, pp. 327–338.

[20] Jian Liu et al. "PLC-cache: Endurable SSD cache for deduplication-based primary storage". In: *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on*. IEEE. 2014, pp. 1–12.

[21] Gokul Soundararajan et al. "Extending SSD Lifetimes with Disk-Based Write Caches." In: *FAST*. Vol. 10. 2010, pp. 101–114.

[22] Taeho Kgil and Trevor Mudge. "FlashCache: a NAND flash memory file cache for low power web servers". In: *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. ACM. 2006, pp. 103–112.

[23] Fei Meng et al. "vCacheShare: automated server flash cache space management in a virtualization environment". In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 2014, pp. 133–144.

[24] Dai Qin, Angela Demke Brown, and Ashvin Goel. "Reliable writeback for client-side flash caches". In: *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference*. USENIX Association. 2014, pp. 451–462.

[25] Ping Huang et al. “FlexECC: partially relaxing ECC of MLC SSD for better cache performance”. In: *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference*. USENIX Association. 2014, pp. 489–500.

[26] Qianbin Xia and Weijun Xiao. “Flash-Aware High-Performance and Endurable Cache”. In: *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2015 IEEE 23rd International Symposium on*. IEEE. 2015, pp. 47–50.

[27] Nitin Agrawal et al. “Design Tradeoffs for SSD Performance.” In: *USENIX Annual Technical Conference*. 2008, pp. 57–70.

[28] Soojun Im and Dongkun Shin. “Flash-aware RAID techniques for dependable and high-performance flash memory SSD”. In: *Computers, IEEE Transactions on* 60.1 (2011), pp. 80–92.

[29] Ping Huang, Ke Zhou, and Chunling Wu. “ShiftFlash: Make flash-based storage more resilient and robust”. In: *Performance Evaluation* 68.11 (2011), pp. 1193–1206.

[30] Sriram Subramanian et al. “Snapshots in a Flash with ioSnap”. In: *Proceedings of the Ninth European Conference on Computer Systems*. ACM. 2014, p. 23.

[31] Ren-Shuo Liu, Chia-Lin Yang, and Wei Wu. “Optimizing NAND flash-based SSDs via retention relaxation”. In: *Proceedings of the 10th USENIX conference on File and Storage Technologies*. 2012.

[32] Cristian Zambelli et al. “A cross-layer approach for new reliability-performance trade-offs in MLC NAND flash memories”. In: *Proceedings of the Conference*

on Design, Automation and Test in Europe. EDA Consortium. 2012, pp. 881–886.

[33] Yangyang Pan, Guiqiang Dong, and Tong Zhang. "Error rate-based wear-leveling for NAND flash memory at highly scaled technology nodes". In: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 21.7 (2013), pp. 1350–1354.

[34] Xiao-Yu Hu, Robert Haas, and Eleftheriou Evangelos. "Container marking: Combining data placement, garbage collection and wear levelling for flash". In: *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*. IEEE. 2011, pp. 237–247.

[35] Radu Stoica and Anastasia Ailamaki. "Improving flash write performance by using update frequency". In: *Proceedings of the VLDB Endowment* 6.9 (2013), pp. 733–744.

[36] Youyou Lu, Jiwu Shu, Weimin Zheng, et al. "Extending the lifetime of flash-based storage through reducing write amplification from file systems." In: *FAST*. 2013, pp. 257–270.

[37] Feng Chen, Tian Luo, and Xiaodong Zhang. "CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives." In: *FAST*. Vol. 11. 2011.

[38] Yu Cai et al. "Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*. IEEE. 2012, pp. 521–526.

[39] Yu Cai et al. “Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime”. In: *Computer Design (ICCD), 2012 IEEE 30th International Conference on*. IEEE. 2012, pp. 94–101.

[40] Ping Huang et al. “An aggressive worn-out flash block management scheme to alleviate SSD performance degradation”. In: *Proceedings of the Ninth European Conference on Computer Systems*. ACM. 2014, p. 22.

[41] Hyojun Kim and Seongjun Ahn. “BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage.” In: *FAST*. Vol. 8. 2008, pp. 1–14.

[42] Mark Woods. *White paper: Optimizing Storage Performance and Cost with Intelligent Caching*. Tech. rep. WP-7107. NetApp, 2010.

[43] Yongseok Oh et al. “Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems.” In: *FAST*. Vol. 12. 2012.

[44] Nimrod Megiddo and Dharmendra S Modha. “ARC: A Self-Tuning, Low Overhead Replacement Cache.” In: *FAST*. Vol. 3. 2003, pp. 115–130.

[45] Myoungsoo Jung et al. “Triple-A: a Non-SSD based autonomic all-flash array for high performance storage systems”. In: *ACM SIGARCH Computer Architecture News*. Vol. 42. 1. ACM. 2014, pp. 441–454.

[46] Jian Ouyang et al. “SDF: Software-defined flash for web-scale internet storage systems”. In: *ACM SIGPLAN Notices*. Vol. 49. 4. ACM. 2014, pp. 471–484.

[47] Ashish Batwara. “Leveraging Host Based Flash Translation Layer for Application Acceleration”. In: *Flash Memory Summit*. 2012.

[48] Yiying Zhang et al. “De-indirection for flash-based SSDs with nameless writes.” In: *FAST*. 2012, p. 1.

[49] Mohit Saxena et al. “Getting real: Lessons in transitioning research simulations into hardware systems”. In: *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*. 2013, pp. 215–228.

[50] *UMass Trace Repository*. `http://traces.cs.umass.edu`.

[51] *SNIA-Block I/O Traces*. `http://iotta.snia.org/tracetypes/3`.

[52] Ren-Shuo Liu et al. “DuraCache: A durable SSD cache using MLC NAND flash”. In: *Proceedings of the 50th Annual Design Automation Conference*. ACM. 2013, p. 166.

[53] Sai Huang et al. “Improving flash-based disk cache with lazy adaptive replacement”. In: *ACM Transactions on Storage (TOS)* 12.2 (2016), p. 8.

[54] Qianbin Xia and Weijun Xiao. “High-Performance and Endurable Cache Management for Flash-Based Read Caching”. In: *IEEE Transactions on Parallel and Distributed Systems* 27.12 (2016), pp. 3518–3531.

[55] John Ousterhout and Fred Douglis. “Beating the I/O bottleneck: A case for log-structured file systems”. In: *ACM SIGOPS Operating Systems Review* 23.1 (1989), pp. 11–28.

[56] Alexandros Batsakis et al. “AWOL: An Adaptive Write Optimizations Layer.” In: *FAST*. Vol. 8. 2008, pp. 1–14.

[57] Medha Bhadkamkar et al. “BORG: Block-reORGanization for Self-optimizing Storage Systems.” In: *FAST*. Vol. 9. Citeseer. 2009, pp. 183–196.

[58] Ricardo Koller and Raju Rangaswami. "I/O deduplication: Utilizing content similarity to improve I/O performance". In: *ACM Transactions on Storage (TOS)* 6.3 (2010), p. 13.

[59] Drew S Roselli, Jacob R Lorch, Thomas E Anderson, et al. "A Comparison of File System Workloads." In: *USENIX annual technical conference, general track*. 2000, pp. 41–54.

[60] Akshat Verma et al. "SRCMap: Energy Proportional Storage Using Dynamic Consolidation." In: *FAST*. Vol. 10. 2010, pp. 267–280.

[61] Sparsh Mittal and Jeffrey S Vetter. "A survey of software techniques for using non-volatile memories for storage and main memory systems". In: *IEEE Transactions on Parallel and Distributed Systems* 27.5 (2016), pp. 1537–1550.

[62] Hyo J Lee, Kyu H Lee, and Sam H Noh. "Augmenting RAID with an SSD for energy relief". In: *Proceedings of the 2008 conference on Power aware computing and systems*. USENIX Association. 2008, pp. 12–12.

[63] Minseok Song and Manjong Kim. "Solid state disk (SSD) management for reducing disk energy consumption in video servers". In: *Proc. of the FAST*. 2011.

[64] Hyojun Kim et al. "Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches". In: *ACM Transactions on Storage (TOS)* 10.4 (2014), p. 15.

[65] Feng Chen, David A Koufaty, and Xiaodong Zhang. "Hystor: making the best use of solid state drives in high performance storage systems". In: *Proceedings of the international conference on Supercomputing*. ACM. 2011, pp. 22–32.

[66] Ricardo Koller et al. “Write policies for host-side flash caches.” In: *FAST*. 2013, pp. 45–58.

[67] Sangwook Kim et al. “Request-Oriented Durable Write Caching for Application Performance.” In: *USENIX Annual Technical Conference*. 2015, pp. 193–206.

[68] DDR JEDEC. “SDRAM standard”. In: *JESD79-3F, JEDEC SOLID STATE TECHNOLOGY ASSOCIATION* (2010).

[69] Yangyang Pan et al. “Quasi-nonvolatile SSD: Trading flash memory nonvolatility to improve storage system performance for enterprise applications”. In: *High Performance Computer Architecture (CA), 2012 IEEE 18th International Symposium on*. IEEE. 2012, pp. 1–10.

[70] Kai Zhao et al. “LDPC-in-SSD: making advanced error correction codes work effectively in solid state drives”. In: *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*. 2013, pp. 243–256.

[71] Qianbin Xia and Weijun Xiao. “Improving MLC Flash Performance with Workload-Aware Differentiated ECC”. In: *Parallel and Distributed Systems (ICPADS), 2016 IEEE 22nd International Conference on*. IEEE. 2016, pp. 545–552.

[72] Song Jiang and Xiaodong Zhang. “LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance”. In: *ACM SIGMETRICS Performance Evaluation Review* 30.1 (2002), pp. 31–42.

[73] Yue Cheng et al. “Erasing Beladys Limitations: In Search of Flash Cache Offline Optimality”. In: *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association. 2016, pp. 379–392.

[74] Carl A Waldspurger et al. “Efficient MRC Construction with SHARDS.” In: *FAST*. 2015, pp. 95–110.

[75] Xiameng Hu et al. “Kinetic modeling of data eviction in cache”. In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association. 2016, pp. 351–364.

[76] Thomas E. Anderson. “The performance of spin lock alternatives for shared-money multiprocessors”. In: *IEEE Transactions on Parallel and Distributed Systems* 1.1 (1990), pp. 6–16.

[77] Ricardo Koller, Ali José Mashtizadeh, and Raju Rangaswami. “Centaur: Host-side SSD caching for storage performance control”. In: *Autonomic Computing (ICAC), 2015 IEEE International Conference on*. IEEE. 2015, pp. 51–60.

[78] Xiameng Hu et al. “LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache.” In: *USENIX Annual Technical Conference*. 2015, pp. 57–69.

[79] Frank Olken. *Efficient methods for calculating the success function of fixed-space replacement policies*. Tech. rep. Lawrence Berkeley Lab., CA (USA), 1981.

[80] Ioan Stefanovici et al. “Software-defined caching: Managing caches in multi-tenant data centers”. In: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM. 2015, pp. 174–181.

[81] Ioan A Stefanovici et al. “sRoute: Treating the Storage Stack Like a Network.” In: *FAST*. 2016, pp. 197–212.

[82] Hunki Kwon et al. “Janus-FTL: finding the optimal point on the spectrum between page and block mapping schemes”. In: *Proceedings of the tenth ACM international conference on Embedded software*. ACM. 2010, pp. 169–178.

[83] Timothy Pritchett and Mithuna Thottethodi. “SieveStore: a highly-selective, ensemble-level disk cache for cost-performance”. In: *ACM SIGARCH Computer Architecture News*. Vol. 38. 3. ACM. 2010, pp. 163–174.

[84] Linpeng Tang et al. “RIPQ: advanced photo caching on flash for facebook”. In: *Proceedings of the 13th USENIX Conference on File and Storage Technologies*. USENIX Association. 2015, pp. 373–386.

[85] Eric Deal. “Trends in NAND flash memory error correction”. In: *Cyclic Design, June* (2009).

[86] Samsung Electronics. *K9F8G08UXM Flash memory datasheet*.

[87] Steven Swanson. *Flash Memory Overview*.

[88] Deepak Ajwani et al. *Characterizing the performance of flash memory storage devices and its impact on algorithm design*. Springer, 2008.

[89] Roberto Bez et al. “Introduction to flash memory”. In: *Proceedings of the IEEE* 91.4 (2003), pp. 489–502.

[90] K Eshghi and R Micheloni. “SSD architecture and PCI Express interface”. In: *Inside Solid State Drives (SSDs)*. Springer, 2013, pp. 19–45.

[91] Kang-Deog Suh et al. “A 3.3 V 32 Mb NAND flash memory with incremental step pulse programming scheme”. In: *Solid-State Circuits, IEEE Journal of* 30.11 (1995), pp. 1149–1156.

[92] Raj Chandra Bose and Dwijendra K Ray-Chaudhuri. "On a class of error correcting binary group codes". In: *Information and control* 3.1 (1960), pp. 68–79.

[93] Stefano Gregori et al. "On-chip error correcting techniques for new-generation flash memories". In: *Proceedings of the IEEE* 91.4 (2003), pp. 602–616.

[94] Nelson Duann. "Error correcting techniques for future NAND flash memory in SSD applications". In: *Technology* (2009), p. 1.

[95] Fei Sun, Ken Rose, and Tong Zhang. "On the use of strong BCH codes for improving multilevel NAND flash memory storage capacity". In: *IEEE Workshop on Signal Processing Systems (SiPS): Design and Implementation*. 2006.

[96] Robert G Gallager. "Low-density parity-check codes". In: *Information Theory, IRE Transactions on* 8.1 (1962), pp. 21–28.

[97] David JC MacKay. "Good error-correcting codes based on very sparse matrices". In: *Information Theory, IEEE Transactions on* 45.2 (1999), pp. 399–431.

[98] J Yang. "Novel ECC architecture enhances storage system reliability". In: *Proc. Flash Memory Summit* (2012).

[99] Shuhei Tanakamaru, Yuki Yanagihara, and Ken Takeuchi. "Over-10×-extended-lifetime 76%-reduced-error solid-state drives (SSDs) with error-prediction LDPC architecture and error-recovery scheme". In: *2012 IEEE International Solid-State Circuits Conference*. 2012.

[100] Yangyang Pan, Guiqiang Dong, and Tong Zhang. "Exploiting Memory Device Wear-Out Dynamics to Improve NAND Flash Memory System Performance." In: *FAST*. Vol. 11. 2011, pp. 18–18.

[101] Congming Gao et al. "Exploit asymmetric error rates of cell states to improve the performance of flash memory storage systems". In: *Computer Design (ICCD), 2014 32nd IEEE International Conference on*. IEEE. 2014, pp. 202–207.

[102] Guanying Wu et al. "DiffECC: improving SSD read performance using differentiated error correction coding schemes". In: *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*. IEEE. 2010, pp. 57–66.

[103] Qiao Li et al. "Access Characteristic Guided Read and Write Cost Regulation for Performance Improvement on Flash Memory". In: *14th USENIX Conference on File and Storage Technologies (FAST16)*. 2016, p. 125.

[104] Guanying Wu and Xubin He. "Reducing SSD read latency via NAND flash program and erase suspension." In: *FAST*. Vol. 12. 2012, pp. 10–10.

[105] Q. Xia and W. Xiao. "High-performance and Endurable Cache Management For Flash-based Read Caching". In: *IEEE Transactions on Parallel and Distributed Systems* PP.99 (2016), pp. 1–1. ISSN: 1045-9219. DOI: 10.1109/TPDS.2016.2537822.

[106] Dongchul Park and David HC Du. "Hot data identification for flash-based storage systems using multiple bloom filters". In: *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*. IEEE. 2011, pp. 1–11.

[107] Michele Fabiano et al. "Design and optimization of adaptable BCH codecs for NAND flash memories". In: *Microprocessors and Microsystems* 37.4 (2013), pp. 407–419.

[108] Congming Gao et al. “Exploiting parallelism in I/O scheduling for access conflict minimization in flash-based solid state drives”. In: *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE. 2014, pp. 1–11.

[109] Yuan-Hao Chang, Jen-Wei Hsieh, and Tei-Wei Kuo. “Endurance enhancement of flash-memory storage systems: an efficient static wear leveling design”. In: *Proceedings of the 44th annual Design Automation Conference*. ACM. 2007, pp. 212–217.

[110] Arie Tal. “Two Technologies Compared: NOR vs. NAND”. In: *M-Systems White Paper* (2003).

[111] Mark Woods. *Optimizing storage performance and cost with intelligent caching*. 2010.

[112] Xinde Hu. “LDPC codes for flash channel”. In: *Proc. Flash Memory Summit* (2012).

[113] E Yeo. “An LDPC-enabled flash controller in 40 nm CMOS”. In: *Proc. Flash Memory Summit* (2012).

[114] Ravi Motwani and Chong Ong. “Robust decoder architecture for multi-level flash memory storage channels”. In: *Computing, Networking and Communications (ICNC), 2012 International Conference on*. IEEE. 2012, pp. 492–496.

[115] Seon-yeong Park et al. “CFLRU: a replacement algorithm for flash memory”. In: *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. ACM. 2006, pp. 234–241.

[116] Heeseung Jo et al. “FAB: flash-aware buffer management policy for portable media players”. In: *IEEE Transactions on Consumer Electronics* 52.2 (2006), pp. 485–493.

[117] Jinho Seol et al. “A buffer replacement algorithm exploiting multi-chip parallelism in solid state disks”. In: *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM. 2009, pp. 137–146.

[118] Guanying Wu, Ben Eckart, and Xubin He. “BPAC: An adaptive write buffer management scheme for flash-based Solid State Drives”. In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE. 2010, pp. 1–6.

[119] Jian Hu et al. “PUD-LRU: An erase-efficient write buffer management algorithm for flash memory SSD”. In: *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE. 2010, pp. 69–78.

[120] Hyotaek Shim et al. “An adaptive partitioning scheme for DRAM-based cache in solid state drives”. In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE. 2010, pp. 1–12.

[121] Tao Xie and Janak Koshia. “Boosting random write performance for enterprise flash storage systems”. In: *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE. 2011, pp. 1–10.

[122] Jian Hu et al. “GC-ARM: garbage collection-aware RAM management for flash based solid state drives”. In: *Networking, Architecture and Storage (NAS), 2012 IEEE 7th International Conference on*. IEEE. 2012, pp. 134–143.

[123] Qingsong Wei, Cheng Chen, and Jun Yang. “CBM: A cooperative buffer management for SSD”. In: *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE. 2014, pp. 1–12.

[124] Suzhen Wu et al. “GCaR: Garbage Collection aware Cache Management with Improved Performance for Flash-based SSDs”. In: *Proceedings of the 2016 International Conference on Supercomputing*. ACM. 2016, p. 28.

[125] Song Jiang et al. “S-FTL: An efficient address translation for flash memory by exploiting spatial locality”. In: *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE. 2011, pp. 1–12.

[126] Mingbang Wang, Youguang Zhang, and Wang Kang. “ZFTL: A Zone-based Flash Translation Layer with a two-tier selective caching mechanism”. In: *Communication Technology (ICCT), 2012 IEEE 14th International Conference on*. IEEE. 2012, pp. 578–588.

[127] Zhiwei Qin et al. “A two-level caching mechanism for demand-based page-level address mapping in NAND flash memory storage systems”. In: *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE. 2011, pp. 157–166.

[128] Chundong Wang and Weng-Fai Wong. “TreeFTL: Efficient RAM management for high performance of NAND flash-based storage systems”. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium. 2013, pp. 374–379.

[129] Mohit Saxena, Michael M Swift, and Yiying Zhang. “Flashtier: a lightweight, consistent and durable storage cache”. In: *Proceedings of the 7th ACM european conference on Computer Systems*. ACM. 2012, pp. 267–280.

[130] Wenji Li et al. “CacheDedup: in-line deduplication for flash caching”. In: *14th USENIX Conference on File and Storage Technologies (FAST 16)*. 2016, pp. 301–314.

[131] Qingpeng Niu et al. “PARDA: A fast parallel reuse distance analysis algorithm”. In: *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE. 2012, pp. 1284–1294.

VITA

Qianbin Xia was born on Januray 1, 1989, in Jiujiang City, Jiangxi Province, China, and is a Chinese citizen. He graduated from Dean No. 1 Middle School, Jiujiang, China in 2007. He received his Bachelor of Science degree in Electrical Engineering from Beijing Institute of Technology, Beijing, China in 2011. He was a PhD candidate in Electrical Engineering at Beijing Institute of Technology from 2011 to 2013, Beijing, China.