



DAC: A dynamic active and collaborative cache management scheme for solid state disks[☆]

Hui Sun^{a,*}, Shangshang Dai^a, Jianzhong Huang^b, Yinliang Yue^{c,*}, Xiao Qin^d

^a Anhui University, Jiu long road 111, Hefei, 230601, Anhui, China

^b Huazhong University of Science and Technology, Luoyu Road 1037, Wuhan, 430074, HuBei, China

^c Zhongguancun Laboratory, Shuangqing Road 30, Beijing, 100084, Beijing, China

^d Auburn University, The Quad Center Auburn, Auburn, 36849, AL, USA

ARTICLE INFO

Keywords:

Caching scheme
Hot/cold
Collaborative
Active write back
NAND flash
Solid state disks

ABSTRACT

NAND Flash memory-based SSDs have been widely used in modern storage systems to improve the performance and energy consumption. Existing caching management schemes rarely consider the utilization of parallel resources in flash memory array inside an SSD while these parallel resources cannot be fully exploited. The I/Os-access conflicts deteriorate the SSD performance and enlarge the response time under write-intensive workloads. Most cache replacement strategies give priority to clean data to improve the I/O-access conflicts, thereby boosting the SSD performance. However, clean data entirely depends on missed read-I/Os requests. Ideally, data replacement ought to evict clean data in the cache thanks to low replacement cost. As such, flash memory-based caching schemes give the highest priority to clean data. However, the proportion of replaced clean data is low in cache data. To solve this issue, we propose a dynamic active and collaborative cache management named DAC, in which the cache is composed of cold cache, hot cache, ghost cold cache, and ghost hot cache. DAC determines hot-cold changes of the I/O requests in workloads based on ghost cache size, thereby adjusting the real cache size to well serve I/O requests. The dynamic write-back window (DWW) mechanism dynamically adjusts the write-back window size in the cold cache. The write-back thresholds automatically update according to the changes in I/O patterns in workloads. When a flash chip is in the idle state, DAC produces clean data by proactively writing normal cold cache data back into flash memory, and the clean data is migrated into the active cold cache. DAC prioritizes replacing data in the active cold cache, aiming to avoid evicting data that are just read from flash memory. This method is adept at boosting the replacement rate of clean data, thereby optimizing SSD performance. We undertake extensive experiments in the latest NVMe SSD simulator to validate the DAC's edge over the seven existing caching schemes including LRU, CFLRU, GCar-CFLRU, LCR, ARC, AD-LRU, and MQsim. The results unveil that our DAC shortens the response time by up to 60.8% with an average optimization rate of 24.4%. When it comes to response-time cliff, the maximum improvement offered by DAC reaches approximately 68.71%; meanwhile, DAC delivers an average optimization rate of 26.9%. Besides, the number of erase count reduces by up to 93.3% with an average improvement rate of 17.04%.

1. Introduction

Recently, NAND Flash-based NVMe SSDs [1] (i.e., NVMe SSD) are widely applied in data storage systems thanks to low latency, high random-access performance, small size, and low power consumption. To achieve high performance in storage systems, high-speed DRAM

is employed in NVMe SSDs to buffer hot data; thus, a portion of I/O requests are swiftly served in DRAM. Most caching management schemes are placed in DRAM to revamp read/write performance. The non-volatile memory (NVM) technology [2] (e.g., PCM [3] and STT-RAM [4]) were rapidly developed to eschew data loss in DRAM after a

[☆] This work is supported in part by National Natural Science Foundation of China under Grants 61702004, 62072001, State Key Laboratory of Computer Architecture (ICT, CAS) under Grant No. CARCH201915, and Natural Science Research Projects at Higher Institutions in Anhui Province (KJ2017A015). Xiao Qin's work is supported by the U.S. National Science Foundation under Grants IIS-1618669, OAC-1642133, CCF-0845257.

* Corresponding authors.

E-mail addresses: sunhui@ahu.edu.cn (H. Sun), shangshagndai@ahu.edu.cn (S. Dai), hjzh@hust.edu.cn (J. Huang), yueyl@zgclab.edu.cn (Y. Yue), xqin@auburn.edu (X. Qin).

<https://doi.org/10.1016/j.sysarc.2023.102896>

Received 4 January 2023; Received in revised form 20 April 2023; Accepted 6 May 2023

Available online 12 May 2023

1383-7621/© 2023 Elsevier B.V. All rights reserved.

power failure. Nevertheless, DRAM is still a popular cache mechanism to strike a trade-off between I/O performance and memory cost [5].

Although the overall performance of NVMe SSDs can be lifted by DRAM, a cache management scheme impacts on the SSD performance due to cache space limitation and high energy consumption [6,7]. Besides, the flash memory has imbalanced read–write performance and limited program/erasure cycles. Thus, the academia and industry have been concentrating on designing robust and high-performance cache management schemes catered for SSDs.

In this work, we divide the existing cache management schemes into three categories. The first group is extended from traditional host-side cache schemes. The objective is to curb the amount of data written from cache to main memory so that a majority of I/O requests are serviced in cache. Some pieces of work improve the cache hit ratio by labeling hot and cold data or partitioning multiple data regions to drive up cache performance. These caching schemes (e.g., LRU [8], LRU-2 [9], 2Q [10], LIRS [11], and ARC [12]) are deployed to optimize cache performance in common scenarios without considering the features of NAND flash memory. The second category of caching schemes strives to optimize cache performance for NAND flash-based SSDs. The popular method is to reduce replacement costs of clean data compared with dirty data. Notably, when page data in a cache is identical to that in the flash memory, the cache page is marked as a clean page; otherwise, the cache page data is referred to as dirty data. If an I/O request firstly enters the cache, the cache page will be dirty because there is no replicated data page in the flash memory. When a data page is retrieved from the flash memory into the cache, the data page is flagged as a clean one. It is noteworthy that the term *clean data* [13] are referred to as cached data, which are duplicated copies of those in flash memory. Dirty data is newly updated data of those in flash memory. Clean data that are evicted by a cache replacement strategy are merely dropped from cache. Thus, evicting clean data is the most cost-effective way of performing cache replacement. This category includes CFLRU [13], CFDC [14], CCF-LRU [15], and AD-LRU [16]. Orchestrated by the caching schemes, clean data in the cache are passively generated by reading requests. When cache replacement occurs, clean data often miss in the cache and dirty data is replaced in most cases. Third, these caching schemes considering I/Os-access conflicts in flash memory (e.g., garbage collection) replace candidate data with low access conflict from dirty data regions when clean pages are insufficient, thereby reducing replacement cost (see, for example, GCaR [17], LCR [18], and ECR [19]).

More often than not, the existing cache management schemes are focused on a single-level optimization rather than an overall system level. Consequently, the schemes are hardly adapt to various I/O access patterns in changing workloads. In workload, the average number of accessed sectors per second varies. In some cases, the amount of write accessed data is orders of magnitude higher than that of the normal stage in each trace (see, for example, burst access stage). When the number of write accessed sectors per second exceeds the most of the time, this stage accounts for approximately 1% of the total running time; meanwhile, the amount of write accessed data is about ten times larger than that of the normal stage. In this work, we define this period as *burst access stage*. From the perspective of the overall systems, we design DAC — a dynamically active and collaborative cache management scheme. DAC is on the basis of multiple system metrics such as hot/cold access patterns, intermittent-I/O access patterns, burst-I/O access patterns, cached-data hot/cold features, replacement strategy in the cache, and adaptability to the status of flash chips.

In DAC, we divide the cache into four regions: cold cache, hot cache, ghost cold cache, and ghost hot cache because of hot and cold data sizes dynamically change in real-time fashion. DAC is aware of hot-cold changes of I/O requests in workloads based on ghost cache size, which is helpful to resize the real cache to well serve the I/O requests. The dynamic write-back window (DWW) mechanism dynamically changes the write-back window size in the cold cache. The write-back thresholds

automatically update according to the changes of I/O patterns. When a flash chip is in the idle state, DAC produces clean data by proactively writing active cold cache data back into flash memory. Consequently, DAC avoids evicting data that are just read from flash memory, thereby improving cache hit rate and shortening response time. We summary the contributions in our study as follows.

► Our approach involves creating real-cache and ghost-cache regions that utilize a hot-cold mechanism. Specifically, we propose implementing this mechanism within the two regions of the DAC system to effectively store cache data and historical access information of a workload. By leveraging this historical information, our proposed mechanism can better govern cache replacement decisions. The real-cache region is divided into cold- and hot-cache regions to store hot and cold data.¹ DAC dynamically adjusts the proportion of hot and cold data space in response to access patterns, thereby improving space utilization and performance in the cache.

► We propose a cache update method to actively produce cold and clean data. The cold cache region is divided into the active-cold cache and normal-cold cache. The former is dedicated to clean data generated by an active-update mechanism. When cache replacement takes place in the cold region, clean data in the active-cold cache are firstly replaced to curb cache replacement cost. Besides, we set a dynamic write-back window (i.e., DWW) to dynamically adjust the triggering condition of active write-back operation. DWW aims to slash the number of write-back operations in flash memory to prolong SSD lifetime.

► We validate the benefits of DAC in terms of response time, response-time cliff, hit ratio, and SSD lifetime. The term response-time cliff refers to the response-time discrepancy among I/Os. We employ the standard deviation of response time to quantify response-time cliff. Compared with LRU, CFLRU, GCaR-CFLRU, LCR, ARC, AD-LRU, MQSim, DAC offers the maximum improvement of 60.8% with an average rate of 24.4% in response time. DAC optimizes the response-cliff performance by up to 68.71% with an average of 26.9%. DAC cuts back erase count by up to 93.3% with an average of 17.04%, thereby significantly promoting SSD lifetime.

The rest of the paper is organized as follows. Section 2 introduces the research background. We elaborate on the related work in Section 3. In Section 4, we describe the design details of DAC. We present a case study of DAC in Section 5. Section 6 is focused on the experimental setup and results to validate DAC. We provide discussion in Section 7. Finally, we conclude the paper in Section 8.

2. Background

Fig. 1 presents the structure of a modern NVMe-based Multi-Queue SSD. On the host side, the operating system issues I/O requests to the device governed by the NVMe protocol. The device embraces host interface logic (i.e., HIL), flash translation layer (FTL [20,21]), and NAND flash array.

When a user I/O request arrives at the SSD, the request is firstly inserted into the device-side I/O-request queue in the HIL module. The request is segmented into flash transactions according to the page size. A flash transaction can be distinguished by metadata information such as logical page address (LPA), bitmap, and time stamp, which can also be utilized to locate the address to store the transaction data. The FTL is responsible for managing resources inside an SSD; the services furnished by FTL include responding to I/O requests, garbage collection, and wear leveling. In the FTL module, a flash transaction initially passes through the cache management module. The cache exhibits high I/O-access performance and low latency compared with the NAND flash.

¹ Notably, a cold region stores cache data that have been accessed only once; a hot region is reserved for cache data that have been referenced at least twice.

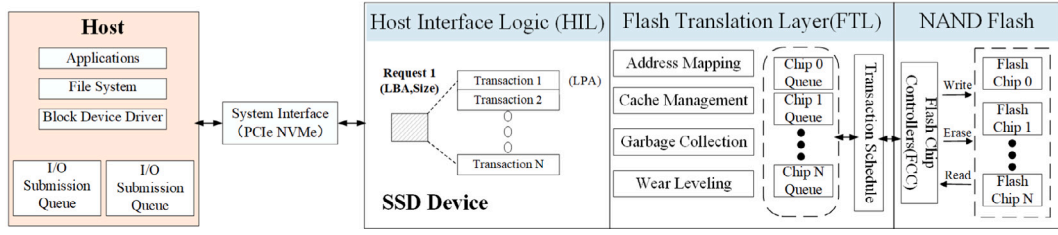


Fig. 1. A Typical Architecture of a Modern NVMe SSD.

There are pieces of research (for example, LRU, CFLRU, GCaR, MQSim) on cache management schemes tailored for different data organizations to enhance overall performance of SSDs.

A page is the basic unit for a program/write or an out-place update operation in the NAND flash. The address mapping mechanism manages a mapping table from LPA to PPA (physical page address). Each updated page can only be rewritten into free pages; afterward, the mapping entry from a LPA to a PPA is updated, marking the previous PPA as invalid. When the number of invalid pages in a block overtakes a preset threshold, a garbage collection operation is triggered to reclaim block [22]. With the address translation module in place, the flash transaction is stored in a chip queue when the chip contains the PPA. It must be noted that a queue is designated to each NAND Flash chip. The transaction scheduling unit (TSU) schedules requests in each queue according to a scheduling policy and data allocation schemes [23] while a page is delivered to NAND flash by flash channel controllers (FCCS) followed by the program or read operation.

3. Related work

In prior work [24], the authors comprehensively studied many crucial techniques for cache management inside an SSD and demonstrated that caching schemes differ in their optimization goals which impact the design of cache management. Notably, an efficient cache management significantly boosts the SSDs' performance, endurance, and robustness. In this paper, we introduce the related work about caching schemes from three aspects such as traditional caching schemes, caching schemes for NAND Flash-based SSDs, and I/O-access conflicts in caching schemes.

3.1. Traditional caching schemes

Cache management is one of the most efficient venues to speed up the performance of storage systems. Caching schemes are widely adopted in DRAM to orchestrate frequently accessed data, thereby shortening response time of accessing hot data, maximizing hit ratio, and boosting SSD performance [25]. Cache data is updated according to replacement policies, which include LRU [8], LRU-2 [9], 2Q [10], LIRS [11], Window-LRFU [26], and ARC [12]. LRU is one of the most popular caching schemes. Cache space in ARC is divided into four regions such as cold cache, hot cache, ghost-cold cache, and ghost-hot cache. An adaptive method to adjust the size of a historical cache stack and a hot cache stack is proposed to achieve high cache performance with the low-cost footprint. Inspired by the idea of ARC, a handful of caching schemes like H-ARC [27] and AD-LRU [16] were devised to optimize performance, such as cache hit ratio, latency, and write amplification (WA).

To bolster the cache hit ratio, Cho et al. [28] proposed a page replacement algorithm to evict a page using an online learning and prediction service: a neural network algorithm learns I/O patterns in real time and determines a victim page. In a prior study [29], Ebrahimi et al. devised a re-configurable caching architecture using a recursive neural network, named RC-RNN, to characterize real-time workload conditions and manage the SSD cache. The RC-RNN method – optimizing cache performance and SSD lifetime – aims to learn workload

characteristics and predict system behaviors. Collected information is gleaned to locate performance-critical data in the cache. Besides, existing hot page monitor mechanism-based caching schemes apply reuse distances or frequencies as the parameter to identify frequently updated pages/blocks. In addition, Li et al. [30] developed a hotness-aware write buffer management scheme or HWB, in which a write buffer is divided into hot, warm, and cold zones. The frequently updated data is stored in the hot and warm zones at the page granularity, whereas metadata of infrequently updated data is residing in the cold zone at the block granularity. A replacement and refresh policy combined with a temperature predictor is designed to enhance the hit rate. In another inspiring work, Yadgar et al. [31] authors studied characteristics relevant to SSD designs in the context of more than 100 traces, where the investigated traits include hot-cold ranges, logical locality, and sensitivity of page size in an SSD.

Nevertheless, the distinctive features of flash memory (imbalanced cost for read and write, GC operations) were overlooked in the design of traditional caching schemes, which are inadequate for managing cache inside an SSD.

Like LRU, ARC is a candidate cache management strategy inside an SSD. Both LRU and ARC overlook the underlying flash memory features. A cache management scheme that favors flash memory is obliged to address the following concerns. (1) In NAND flash-based SSDs, cache replacement algorithms ought to pay heed to hit ratio and cost caused by replacing dirty data. The replacement cost of dirty data is higher than that of clean data concerning response time, write amplification, and erase counts. (2) There are a pile of invalid pages in a block in NAND flash due to out-place updates. The garbage collection procedure that gleans valid pages in a block erases the original block. Thus, it is crucial to avoid evicting cache data to the chip amid garbage collection.

Inside an SSD, a cache management scheme must incorporate features of NAND flash; otherwise, the scheme is likely to cause high replacement costs, GC operations, and conflicts with GC I/Os, affecting the SSD performance and lifetime.

3.2. Caching schemes for NAND flash-based SSDs

With the evolution of flash memory, cache management becomes a common practice in SSDs. Due to the asymmetric read–write costs in NAND flash, a high hit ratio in cache hardly affects SSD performance. CFLRU (clean first LRU) [13] favors dirty data with a high priority in the cache. Amid cache replacement, clean data is placed with high priority to curb the overhead of writing cache data to flash memory. An early replacing clean data leads to prematurely replaced read data in the cache, resulting in a sharp decrements in cache hit rate. To reduce costs of cache replacement, the LRU linked list is partitioned into two segments in CFLRU. A segment near the most recently used (MRU) is referred to as the work-space whereas another segment near the LRU is named as clean first region. Upon cache replacement, it takes priority to replace clean data in the write-back region. In case of no clean data in this region, the least frequently used data will be replaced. LRU-WSR [32] scheme delays the procedure of replacing hot data by adding hot and cold tags. In CFDC [14], clean and dirty data is integrated into

two linked lists, which deteriorates data-retrieval performance when cache data is replaced. CCF-LRU [15] yet considers the replacement cost of cold data, hot data, and clean data. Two linked lists are put forward to maintain cache data. More specifically, the first list stores dirty data and hot clean data; the second one manages cold clean data. In the cache replacement, the cold clean data is firstly replaced from the second linked list. AD-LRU [16] conflates the advantages of the aforementioned caching schemes to revamp SSD performance. AD-LRU divides the cache into cold zone and hot zone. The cold zone stores cache data that have been accessed only once; the hot zone is reserved for cache data that have been referenced at least twice. When cache replacement occurs, AD-LRU firstly elects to evict the least recently used clean data from the cold space. When there is no clean data in the cold region, AD-LRU will kick in the second-chance policy [32] to replace dirty data.

It must be noted that the hit rate is one of the metrics evaluating cache performance. Cache performance may be boosted from three aspects, namely, hit time, miss ratio, and miss penalty. Because hardware features of the cache determine hit time, it makes sense for cache management schemes to be focused on enhancing miss ratio and miss penalty [13,15,32]. The above schemes, however, are designed from a one-sided perspective. In NAND flash-based SSDs, a well-oiled cache management scheme should consider both I/O patterns in workloads (e.g., write-intensive and burst I/Os) and features in flash memory (e.g., read/write imbalance and garbage collection), which are combined to maximize cache performance. Recognizing that I/O patterns are diverse in workloads, we argue that it is challenging to devise cache management schemes to adapt to dynamic I/O patterns in workloads.

VS-Batch [33] – a visibility graph-based SSD cache scheme – pays heed to the temporal and spatial locality of workloads where the two localities are unified through visibility graphs. VS-Batch relieves cache performance jitter and reduces I/O latency. Chen et al. [34] designed a page type-aware read caching scheme – PACA – in SSDs. According to the read-latency variation of quad-level cell (QLC) pages, high-latency QLC pages are preferentially cached in a workload-adaptive manner. PACA optimizes read performance and cache hit ratio. To address the unstable write performance inside an SSD, Zeng et al. [35] proposed a flash memory-based caching scheme using page reconstruction and hot-data identification mechanisms. The page reconstruction mechanism cuts back the number of write operations. The hot-cold data identification mechanism marks hot-cold levels for cache pages. Then, the cache pages are written back into flash memory in a predefined priority manner. A cache-aware GC scheme [36] is advocated to make use of cache pages to suppress page movement during the course of GC. This scheme reshapes SSD performance and lifetime compared with the greedy algorithm. Rodriguez et al. [37] designed a lightweight reinforcement learning cache replacement scheme named *CACHEUS*, which leverages a combination of replacement algorithms to handle a diversity of workloads. *CACHEUS* switches between SR-LRU and CR-LFU using a new class of machine-learning-driven caching algorithms for workloads. Lin et al. [38] implemented a cache management approach powered by CFLRU, which considers both temporal and spatial locality in workloads. Moreover, the three-level linked lists is applied to shift hot and cold data pages in cache, thereby achieving optimized cache hit ratio and shortened I/O latency for SSDs compared with CFLRU.

In our work, we aim to overcome the above hindrance by the virtue of a dynamically active and collaborative cache management scheme named DAC. The DAC technique judiciously adapts to features in workloads and flash memory, thereby maximizing the cache performance.

3.3. I/O-access conflicts in caching schemes

The NAND flash memory-aware cache management schemes (e.g., CFLRU) are slated to suppress replacement cost of clean cache data to

optimize cache performance. To address the complexity of data flow inside an SSD, great efforts were devoted to reducing access conflicts by incorporating cache-data status and I/O-access characteristics [39–41]. A cache management scheme [42] was proposed to periodically write cache data into flash memory. An SSD has three status, i.e., idle, active, and standby, one of which is set in advance. Although the caching scheme achieves good performance, the active write-back method gives rise to an excessive number of write operations in flash memory, thereby limiting SSD lifespan. GCaR [43] incorporates the garbage collection (GC) state of flash memory during the course of cache replacement. To reclaim invalid pages in flash memory, a garbage collection is evoked to erase blocks in which the number of free pages drops to a preset threshold. It is noteworthy that valid pages in a block are copy-rewritten into other blocks with enough free pages. In the GC procedure, there are a large number of internal read, write, erase operations, which conflict with external user-I/O requests, impairing the performance of an SSD. In the cache replacement, GCaR gives high priority to enable cached data, the corresponding block of which is in the garbage collection, stay in the cache. Nevertheless, the cached data, the corresponding block of which is not in GC, is firstly replaced, thereby alleviating I/O competition and improving performance.

LCR [18] quantifies chip load caused by internal and external I/O-access requests in a chip queue governed by GCaR. During cache replacement, clean data is first replaced followed by data with low chip load. The ECR [19] caching scheme expands LCR to mitigate the impact of write-back-window size on performance. In Co-Active [44], I/O access patterns in workloads and the status of flash chips inside an SSD are used to determine which data pages in the cache need to be evicted. Then, Co-Active proactively evicts the cold cache data as the corresponding SSD channels are idle, which can reduce the waiting time of flushing data. Co-Active minimizes the latency of normal I/Os incurred by the cache eviction.

The cached data of small requests are more likely to be re-accessed than large write ones. To utilize the limited-size cache inside an SSD, researchers designed an adaptive request block-based cache management scheme with I/O request granularity (ARB) [45,46] to optimize cache hit rate and I/O latency. ARB relies on a multi-level linked list to filter cached request blocks. The size and intensity of write requests are key factors to allocate a high priority to small-sized request blocks. Evidence shows that [47] cached missing table misses result in extra reads data from flash memory, causing I/Os stalls in SSDs. As such, a cached mapping table prefetching strategy is proposed to decouple and prefetch logical-to-physical translation pages from processing read requests. The approach to prefetching mapping table entries can avoid stalls incurred by cached mapping table misses, thereby improving random-read performance in NVMe SSDs. A cache management mechanism named *Justitia* [48] was devised to mitigate the competition for cache resources among multiple applications and to improve access fairness in SSDs. When an SSD serves multiple applications simultaneously, cache misses prolong the response time of I/O requests, which in turn leads to severe access unfairness. *Justitia* reserves – over-provisioning (OP) – a cache area in DRAM inside an SSD to guarantee an acceptable cache hit ratio for multiple applications. Meanwhile, *Justitia* provides privileges for unfair applications to revamp the cache hit ratio for applications. A stream-aware write caching scheme (SAC) [49] addresses two issues in multi-stream SSDs: inter-stream conflicts in cache and inaccurate stream identification. The SAC scheme first uses the stream information in the write cache to reduce write amplification or WA. A feature-based I/O stream identification framework [50] was built to investigate the impact of stream allocation with I/O patterns in workloads on WA of multi-stream SSDs. In addition, an I/O pattern-based stream identification method was proposed to correlate tested attributes (e.g., I/O size, I/O rate) with inter-workload ones (e.g., frequency, sequential nature), thereby identifying a good combination of different stream features.

Table 1
Studying of popular caching schemes.

	Caching schemes	Handling burst I/Os	Page hotness	Page state	Chip state	Miss penalty	Dynamic adjustment
Traditional caching schemes	LRU [8]	×	×	×	×	×	×
	ARC [12]	×	×	×	×	×	✓
	LRU-2 [9]	×	✓	×	×	×	×
	2Q [10]	×	✓	×	×	×	×
	LIRS [11]	×	✓	×	×	×	×
Caching schemes for NAND Flash-based SSDs	AD-LRU [16]	×	✓	✓	×	×	×
	CF-LRU [13]	×	✓	✓	×	×	×
	LRU-WSR [32]	×	✓	✓	×	×	×
	CFDC [14]	×	✓	✓	×	×	×
	CCF-LRU [15]	×	✓	✓	×	×	×
I/O-Access conflicts in caching schemes	GCaR [43]	×	×	×	✓	✓	×
	LCR [18]	×	✓	✓	✓	✓	×
	ECR [19]	×	✓	✓	✓	✓	×
	ARB [45][46]	×	✓	✓	×	×	×
	Co-active [44]	✓	✓	✓	✓	✓	×
	DAC	✓	✓	✓	✓	✓	✓

Column 3: taking into account the burst behavior in workloads.

Column 4: considering hot and cold conditions of cached pages.

Column 5: coping with the clean, dirty state of cached pages.

Column 6: incorporating back-end chip-level busy/idle state.

Column 7: dealing with the cost of cache misses.

Column 8: supporting adaptive resizing of cache Windows.

In these cache scheme studies, the impacts of load in flash memory on cache performance were systematically investigated. The chip with light load is favored to cope with evicted data in the cache. However, these strategies merely deal with the load of a cache rather than I/O-access patterns that are altering. Due to the lack of adjustments for changing load patterns, these cache schemes fail in settling I/O conflicts between user requests and internal operations of flash memory.

3.4. Issues in caching schemes and motivation

Caching schemes are primed to minimize cache-replacement cost. In GCaR, LCR, and ECR, the priority of cache replacement is given to clean data. When there is no clean data in cache, it is prudent to take into account the I/O-access characteristics of the underlying flash memory, such as the GC state in GCaR, and the load on the flash chip in LCR and ECR. Thus, clean data, especially cold-clean data, lead to the lowest replacement cost. Much attention has been paid to novel approaches to discovering cold clean data. For instance, Co-active requires user-defined parameters: a static parameter setting offer subpar performance under dynamically changing workloads and cache sizes. Given such workloads [51], DAC shortens the response time of Co-active by an average of more than 20%. According to Table 1, we reckon that the existing caching schemes in flash memory-based SSDs confront the following roadblocks.

Problem 1. Without considering the diversity of I/O-access patterns in workload conditions, there are different hot- and cold-data access behaviors of applications. The available cache space inside SSDs is usually small, which makes it a hindrance to effectively distinguish hot and cold data. When we can accurately identify hot and cold I/O-access patterns and convert cold dirty data into clean data, it reduces replacement latency and avoids write amplification caused by writing hot data back to flash memory.

Problem 2. Clean pages are entirely generated by read requests. Few clean pages exist in the cache under write-intensive workloads. More times than not, dirty data in the cache are evicted. In the priority replacement of clean data, clean data produced by read data from flash memory in the cache are quickly rotated out. This procedure alters the replacement order in LRU, thereby adversely affecting cache hit rate and cache efficiency. Therefore, a large amount of dirty data can be converted into clean data when flash memory is idle, increasing the

amount of clean data. Thus, data just read from flash memory cannot be replaced, which avoids latency caused by repeatedly reading the same page from flash memory.

Problem 3. Clean pages and dirty pages are stored in the same linked list in the cache. For the priority replacement of clean data, some solutions [13,17,19] overlook the fact that hot clean cache data have a higher replacement cost than that of cold dirty data. It can achieve a lower possibility of mistaken data replacement by preferentially replacing cold dirty data instead of hot clean data, preventing multiple read operations from flash memory. In addition, the active write-back mechanism can write back cold dirty data early, minimizing cache-miss latency.

We propose a dynamically active and collaborative cache management named DAC to address the above obstacles from three perspectives. First, DAC divides a cache space into four regions, namely, cold, hot, ghost cold, and ghost hot regions. The ghost space alleviates Problem 1 by catering to hot- and cold-data access behaviors in the hot and cold cache regions. Second, DAC embraces a new active cache update mechanism to actively convert dirty data into cold clean data rather than simply relying on read I/O requests in the traditional cache schemes. DAC avoids cache data that are recently loaded into the cache to be evicted out due to the priority of replacing clean data. Consequently, DAC mitigates Problem 2 of interfering with the replacement order of clean data. To handle Problem 3, DAC gives a high priority to swap cold clean data when cache replacement occurs in the cold region. This replacement scheme prevents damaging temporal locality in the cache while lowering the replacement ratio of hot clean data compared with that of cold dirty data. Hence, the proportion of replaced clean data jumps up to cut back response time in SSDs.

4. Design

We present a system overview of the DAC cache management scheme, followed by the DAC's design details. Afterward, we elaborate the implementation of DAC from the perspective of three scenarios, namely, (1) real and ghost cache miss, (2) real cache hit, and (3) real cache miss and ghost cache hit.

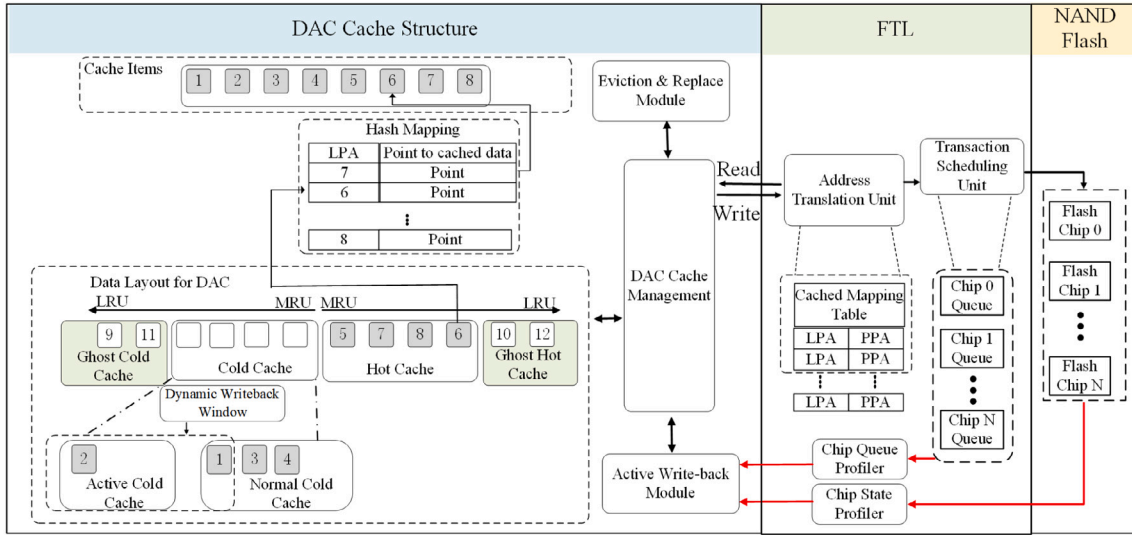


Fig. 2. Overall system of DAC in NAND flash-based SSDs.

Table 2
Notation and description.

Notation	Description
C_c	Cold cache
H_c	Hot Cache
G_c	Ghost cold cache
G_h	Ghost hot cache
C_n	Normal cold cache
N_{C_c}	The number of data entries in region C_c in the cache
N_{H_c}	The number of data entries in region H_c in the cache
N_{C_n}	The number of data entries in region C_n in the cache
N_{C_a}	The number of data entries in region C_a in the cache
N_{G_h}	The number of data entries in region G_h in the cache
N_{G_c}	The number of data entries in region G_c in the cache
d	Target size
C_a	Active cold cache
tr	Flash transaction

4.1. Overview system

In this section, we dive into the framework design of the DAC caching scheme depicted in Fig. 2. The cache management module is an essential component that is in charge of the logical data-flow in the cache. DAC orchestrates cache data through a new data layout and a hash mapping module. This module determines whether data of a LPA is placed in the cache along with the address of cached data. On the device side, the cache management module receives the flash transactions from HIL module and returns required data; meanwhile, read/write transactions are transferred to FTL performing the operations in flash memory.

The lineup of existing popular caching schemes includes LRU, ARC, AD-LRU, CFLRU, GCar, and LCR. The passive update mode is endorsed in the synchronous mode between the cache and back-end NAND flash memory. Only when the cache is full and a new write-I/O request is inserted into the cache, old data is replaced into the flash memory. The disadvantage in this update method, however, is that updating data from cache to flash depends on whether new data is written into flash memory. A large amount of data transferred from cache to flash memory inevitably cause underlying access conflict and high

response delay. To tackle this issue, we propose an active write-back module to proactively synchronizes a portion of cache data with that in flash memory at a period, reducing the response time of any following replaced cache data. Compared with the passive update mode, the flash memory endurance can be impaired by amount of cache data that are actively written into the flash memory. This observation motivates us to devise the DWW mechanism in the active write-back module where cache data is moved into flash memory on demand. Algorithm 3 outlines the data-flow control in Section 4.2.

The eviction and replacement module (1) elects a cache data, (2) evicts the data to free up space, and (3) serves the new coming transactions. In existing flash memory-based cache management strategies (e.g., cflru, GCar, and LCR), the eviction-and-replacement module replaces cache data according to data replacement cost. More specifically, the low-cost replacement data is firstly evicted out cache. The order of cache replacement cost for cache data from low to high is (1) clean data, (2) dirty data with access conflicts, and (3) the other dirty data. This order is effective in a majority of cases, but the replacement cost of clean data is not always lower than that of dirty data. For example, the replacement cost of cold dirty data is lower than that of hot clean data [32]. The evict-and-replacement module in DAC embraces the dynamic replacement strategy performing as follows. First, DAC decides whether to replace the cache data in the cold area or hot area according to access patterns. Second, when a cache replacement occurs in the cold area, cold clean pages are actively replaced and written back, thereby curtailing the cache replacement cost. The detailed procedure is elaborated in Algorithm 2.

Five LRU linked list managing cache data in DAC include (1) active cold cache, (2) normal cold cache, (3) hot cache, (4) ghost cold cache, and (5) ghost hot cache. The normal cold cache and hot cache reside in the real cache region whereas the ghost cold cache and ghost hot cache exist in the ghost cache region. We summarize the notations used in the rest of this paper in Table 2. We refer to cold cache and hot cache as C_c and H_c , respectively. Ghost cold cache and ghost hot cache are represented by G_c and G_h , respectively. The size of G_c represents the number of data entries in region G_c in the cache. Active cold cache and normal cold cache are denoted by C_a and C_n , respectively. Cache C_c is composed of two pieces C_a and C_n . The size of C_c captures the number of data entries in region C_c in the cache. Table 1 lists notation and description used in this paper. In what follows, we shed bright light on these cache concepts.

Normal Cold Cache (C_n). New read/write I/O data (i.e., the LPA of I/O that cannot be found in the global translation table [21]) is inserted

into the MRU in C_n in the cache. When the I/O request is a write transaction, the inserted data in C_n is dirty. If the request I/O is a read transaction, the data inserted into C_n is clean data. For Problem 3 (see Section 4.1), DAC sets the priority replacement for clean data written into cache by read I/Os.

Active Cold Cache (C_a). When the cache data in C_n meets the trigger condition in the active write-back procedure, the cache data is written into flash memory. The data in the flash memory is consistent with that in the cache; the version of cache data changes clean from dirty; and then the data is migrated from C_n to MRU in C_a .

Ghost Cold Cache (G_c) and Ghost Hot Cache (G_h). Cache G_c and G_h store metadata of evicted data from cache C_c and H_c , respectively. This method is slated to compress cache access information to conserve cache space. This information determines the cold and hot of I/Os requests. The proportion of C_c and H_c in the cache and the dynamic write-back window size are dynamically adjusted according to this information. The proportion of C_c stipulates the ratio of the number of data entries in region C_c to the total data entries in the cache.

Hot Cache (H_c). Hot cache data is stored in H_c . When the cache data is firstly accessed,² we refer to the cache data as cold data which is inserted into the C_c . If the cache data is accessed twice or more,³ it is transmitted to the H_c ; meanwhile, the cache data is postponed to be evicted from the cache when the cache replacement occurs. When an I/O transaction hits any region of C_a , C_n , H_c , G_c , and G_h , it means that the data has been accessed in a period. Then, we regard the cache data as hot data. When C_a , C_n , or H_c is hit, the data in the region transfers to the MRU side in H_c . If G_c or G_h is hit, it means that the data was previously placed in the cache but it has been evicted. Metadata of cache data is stored in the cache rather than the real cache data. If the data is the write transaction, it is inserted into the MRU side in H_c . If the data is the read transaction, data is read from the flash memory; and then, it is inserted into the MRU side in H_c .

4.2. DAC design

Existing NAND flash-based caching schemes pay heed to I/O access patterns in workload, cache replacement, or access conflict in flash memory. A few caching solutions integrate multiple levels to improve SSD performance. From the perspective of overall system performance, we design the DAC caching scheme that incorporates I/O-access patterns, data management in the cache, and I/Os-access behaviors in underlying flash memory. The overarching goal of DAC is to effectively lift the cache performance in terms of response time, performance cliff, and write amplification.

DAC addresses the above-mentioned problems discussed in Section 3.3. An array of prior studies show that I/O-access patterns are featured with cold and hot characteristics [40,52]. DAC divides the cache space into cold region and hot region and sets the ghost space for hot and cold cache region, which can deal with Problem 1. Ghost cache records the metadata (e.g., logical address page LPA) of the cache data that have been recently moved out of the cache space. Through the cache metadata entries (LPAs) in the ghost cache, DAC dynamically adjusts the proportion of hot and cold data in the cache as well as the size of the dynamic write-back window. Since the ghost cache only stores the logical addresses of cache data, the footprint of the ghost cache is fairly low. The amount of cache data access information can be recorded twice as much as the real cache size by means of the ghost-cache mechanism.

To handle Problem 2, we design a method to active update cache data in DAC. The active-update mechanism can convert the dirty data into clean data in the cache, which averts all clean data to be produced by the procedure of reading I/Os from flash memory. Thus,

DAC curtails the replacement ratio of clean data in cache replacement while shortening response time for user I/Os. The active-update method updates the cold cache data into flash memory when the flash chip is sitting idle. Therefore, data in flash memory is consistent with those in the cache. More prosaically, the data that are initially dirty become clean through the active-update operation.

When cache replacement is performed in C_c , cache data in C_a should be firstly replaced. The replaced cache data is cold clean data generated by active-update operations. Traditional cache clean data created by read I/O requests are stored in C_n . This replacement scheme avoids damaging temporal locality in the cache while keeping the replacement ratio of hot clean data higher than cold dirty data. However, frequent updating the cache data into flash memory inevitably brings write amplification and I/Os-access conflicts. When it comes to the active-update mechanism in DAC, we advocate for a dynamic write-back window – a profiler for flash memory status – to suppress the overhead of the active-update mechanism by limiting write-back data.

In this study, we assume that C_c and H_c are capable of caching the M number of data entries (or pages) at the most. In C_c , H_c , G_c and G_h , the total number of cached pages is no more than $2 \times M$. To dynamically adjust the proportion of cold and hot data in DAC (i.e., the proportion of C_c and H_c), we define d as the target size of region C_c . The initial value of d is 0, which can be adjusted during the workload. The detailed dynamic strategy is described in the eviction-replacement algorithm. The G_c and G_h register the information of the prior cache data that no longer exist in the cache. In doing so, hot data can be singled out from cold data. In Section 4.5, we introduce a way of leveraging the information of G_c and G_h to dynamically adjust the size of C_c and H_c in the cache. Section 4.3 delineates the method to configure the size of the dynamic write-back window. The real cache regions (C_a , C_n , and H_c) and ghost caches (G_c and G_h) are initialized to zero. There are three cases for each read/write transaction: (1) both real cache and ghost cache miss, (2) hit C_a , C_n , H_c in real cache, and (3) hit ghost cache such as G_c or G_h .

DAC achieves high efficiency with minimal time overhead. The time overhead of DAC primarily encompasses two aspects. The first aspect involves querying, inserting, deleting, and modifying cached pages. The second one is the traversal of page requests corresponding to the chip monitoring when an active write-back operation is initiated. The cache is managed through a set of linked lists and hash tables. The time complexity of accessing to the head or tail nodes of the linked list is $O(1)$. In case of accessing to nodes other than the head and tail, the complexity of searching the hash table is $O(n)$. Meanwhile, the average time complexity is close to $O(1)$. To facilitate actively writing back, it is essential to traverse data that meets the write-back conditions in the dirty linked list. In the worst case, the entire linked list ought to be traversed: the time complexity is $O(n)$. The amount of dirty data is small under active write-back, so the traversal time is low.

Let us assume that the cache size is 32 MB with the page size of 4 KB. We shed bright light on the overhead of the proposed DAC as follows. The size of the ghost area is equal to that of the real cache. Each real cache page corresponds to a ghost cache page, and the pointer size of the two-type cache page is 8 B. In DAC, a ghost-page data structure occupies 8 B, and the number of ghost pages is 8,192. Moreover, the chip state profiler employed in DAC uses seven pointers, the overhead of which is negligible. Therefore, the total overhead is 192 KB accounting for 0.6% of the cache space.

4.3. Procedure of real and ghost cache misses

Initially, cache C_n , C_a , H_c , G_c and G_h are all empty. A cache miss occurs when transaction tr accesses the real cache and ghost cache. If tr is a write request, the request is inserted into the MRU side in C_n and marked as dirty data. If tr is a read request, the data needs to be read

² The cache page fails to hit the real cache and ghost cache space.

³ The cache page hits the real cache or ghost cache.

Algorithm 1 Real and ghost cache misses workflow in DAC

Require: Flash Transaction $T_1, T_2, T_3, \dots, T_i, \dots$

```

1: procedure CACHE_ALGORITHM( $T_i$ )
2:   repeat
3:     /*For every  $T_i$ , one of the three cases occurs.  $T_i$  misses in
        $C_n, C_a, H_c, G_c$  and  $G_h$  */
4:     if  $N_{C_n} + N_{C_a} + N_{G_c} < M$  then
5:       if  $N_{C_n} + N_{C_a} + N_{H_c} + N_{G_c} + N_{G_h} \geq M$  then
6:         if  $N_{C_n} + N_{C_a} + N_{H_c} + N_{G_c} + N_{G_h} == 2 * M$  then
7:           Delete the bottom item (LRU)  $L_x$  in  $G_h$ 
8:         end if
9:         Eviction and Replacement Algorithm ( $T_i$ )
10:      end if
11:    else
12:      if  $G_c \neq 0$  then
13:        Delete the bottom item (LRU)  $L_x$  in  $G_c$ 
14:        Eviction and Replacement Algorithm ( $T_i$ )
15:      else
16:        /* $G_c == 0$  */
17:        /*Evict one data slot from  $C_c$ , Details see Lines 3-11 in
          Eviction and Replace Algorithm */
18:        end if
19:      end if
20:      Insert to the MRU position in  $C_n$  ( $T_i$ )
21:      Scan and Active Write-back ( $C_n, C_a$ )
22:    until
23:  end procedure

```

from flash memory; and then, the data is written into C_n with a mark of clean data.

When the cache space is non-empty, the total cache size can be set as the expression $-N_{C_c} + N_{H_c} + N_{G_c} + N_{G_h} \leq 2 \times M$ – to reduce the region sizes of G_c and G_h and to delete invalid data in the ghost cache.

It is worthy noting that N_{C_c} represents the number of data entries in region C_c in the cache. Similarly, we denote N_{H_c} , N_{C_n} , N_{C_a} , N_{G_h} , and N_{G_c} as the number of data entries in cache region H_c , C_n , C_a , G_h , and G_c , respectively.

Suppose we have the following conditions

$$N_{C_n} + N_{C_a} + N_{G_c} < M \quad (1)$$

$$N_{C_n} + N_{C_a} + N_{H_c} + N_{G_c} + N_{G_h} == 2 \times M.$$

Then, we obtain $N_{H_c} + N_{G_h} > M$, which means the hot data region in the cache cost much space and the cache space is full (see Lines 5–9 in Algorithm 1). DAC eradicates the page data at the LRU in G_h to conserve the cache space occupied by old data in the ghost cache. Afterward, the eviction-replacement module is prompted to evict a cache entry from the real cache.

In addition, suppose we have

$$N_{C_n} + N_{C_a} + N_{G_c} < M, \quad (2)$$

and

$$N_{C_n} + N_{C_a} + N_{H_c} + N_{G_c} + N_{G_h} \geq M \quad (3)$$

$$N_{C_n} + N_{C_a} + N_{H_c} + N_{G_c} + N_{G_h} \neq 2 \times M.$$

More times than not, an SSD's cache is full during runtime whereas a non-full cache exists during an initial phase referred to as a cold start. In the scenario of non-full cache, Eq. (4) is converted into Eq. (2); thus, we articulate the full and non-full cache cases in Algorithm 1 (see Line 10).

When the following condition holds $N_{C_n} + N_{C_a} + N_{G_c} \geq M$ and $N_{G_c} \neq 0$, the cold data region occupies half of the cache space. In this case, DAC deletes the cache data at LRU in G_c ; meanwhile, a cache entry from the real cache is evicted out (see Lines 13–15 in Algorithm 1).

Otherwise, when we have $N_{C_n} + N_{C_a} + N_{G_c} \geq M$ and $N_{G_c} == 0$, DAC replaces cache data from C_c . The LRU data in C_a is preferentially

replaced. When the data in C_a is empty, the LRU data in C_n is replaced (see Lines 17–19 in Algorithm 1).

When there is free space in cache, request data of missed transaction tr is inserted into the MRU in C_n (see Line 20 in Algorithm 1). If tr is a write operation, the data marked as dirty data can be placed into the MRU in C_n . If tr is a read, the data – noted as clean data – should be retrieved from flash memory to the cache. Given a full real cache, a cache page is evicted out from the cache if a new write request arrives at the cache. The metadata of the evicted page is inserted into the MRU at the ghost cache link list. In the cold cache, metadata are moved to the ghost cold cache, whereas metadata residing in the hot cache are positioned in the ghost hot cache. When a new read request is issued, the real cache is queried first; if the real cache misses, the request goes to the ghost cache. If the metadata of the requested data is accessed in the ghost cache, a ghost hit occurs meaning that the cache was accessed recently. The data being retrieved from the flash memory are inserted into cache H_c . If there is no hit in the ghost cache, a page evicted from the tail of the real cache is appended into the ghost cache. Upon arrivals of more requests, the data at the end of the ghost cache link list are evicted when the ghost cache is also full; then, no information of the data can be located in the cache.

4.3.1. Eviction and replacement workflow

Algorithm 2 Eviction and Replacement workflow in DAC

Require: Flash Transaction Tr

```

1: procedure EVICTION_AND_REPLACEMENT_ALGORITHM( $Tr$ )
2:   if  $(N_{C_n} + N_{C_a} > d \ \&\& \ N_{C_n} + N_{C_a} \neq 0) \ || \ (Tr \text{ hits } G_h \ \&\& \ N_{C_n} + N_{C_a} == d)$  then
3:     /*Evict one data slot form  $C_c$  */
4:     if  $N_{C_n} > 0$  then
5:       Remove bottom item(LRU) in  $C_a$ :  $L_x$ 
6:     else
7:       /* $C_a$  is empty, Evict one data slot form  $C_n$  */
8:       if The bottom item (LRU)  $L_x$  in  $C_n$  is dirty data then
9:         Sync the  $L_x$  to the NAND Flash Storage
10:      end if
11:      Remove bottom item(LRU) in  $C_n$ :  $L_x$ 
12:      /*Increase the Dynamic Write-back Window(DWW) */
13:      if  $N_{G_h} \neq 0 \ \&\& \ N_{G_c} > N_{G_h}$  then
14:         $DWW = \min\{N_{C_n} + N_{C_a}, DWW + \lfloor \frac{N_{G_c}}{N_{G_h}} \rfloor\}$ 
15:      else
16:         $DWW = \min\{N_{C_n} + N_{C_a}, DWW + 1\}$ 
17:      end if
18:    end if
19:  else
20:    /*Evict one data slot from  $H_c$  */
21:    if The bottom item (LRU)  $L_x$  in  $H_c$  is dirty data then
22:      Sync the  $L_x$  to the NAND Flash memory
23:    end if
24:    Remove bottom item (LRU) in  $H_c$ :  $L_x$ 
25:  end if
26: end procedure

```

The eviction-replacement algorithm aims to conserve space in real cache for new transaction tr . The algorithm not only minimizes cache replacement cost, but also avoids replacing hot data and data that are just loaded into the cache. Algorithm 2 outlines the data-flow of eviction and replacement procedure. This algorithm is one of essential approaches to adjusting the proportion of cache space occupied by hot and cold regions governed by target size d in region C_c . If condition $N_{C_n} + N_{C_a} > d$ is satisfied, the size of cache C_c is greater than target size. Therefore, the algorithm evicts cache entries in region C_c , forcing C_c and H_c to mimic target size (see Lines 2–18 in Algorithm 2). When the cache data is evicted from region C_c , it is imperative to curb cache replacement cost. For example, if region C_a is not empty, cache data at LRU in C_a is preferentially evicted. If data in C_a is empty, LRU-side

cache data should be replaced out from region C_n . When the transaction tr hits G_h and there is $N_{C_n} + N_{C_a} == d$, cache data pages in region C_c will be evicted. On the contrary, we have

$$N_{C_n} + N_{C_a} \leq d \text{ or } N_{C_n} + N_{C_a} == 0, \quad (4)$$

which signifies that the size of region C_c is smaller than target size in the cache. Hence, the cache entry in LRU is evicted from H_c (see Lines 20–25 in Algorithm 2).

The DWW size determines cache C_a 's capacity. More prosaically, expanding DWW size enlarges the size of cache C_a . We articulate the following two cases with respect to the DWW size.

Case 1 — An increasing size of DWW. When we have $N_{C_n} > d$, it implies that the cold cache size is large than the target size d in region C_c . The amount of cold data is much larger than that of hot data. The active write-back module writes cold data into flash memory. Because this case is conducive to active write-back operations, we should extend the active write-back cache (size of C_a) — increasing cache C_a 's size.

Case 2 — A decreasing size of DWW. When an I/O request hits cache C_a , the active write-back for the hit data ought to be canceled, and the data is moved to the cache H_c . Meanwhile, the size of DWW shrinks decreases in purpose of curbing the size of the active cold cache (the size of C_a).

4.3.2. Scan and active write-back workflow

Algorithm 3 Scan and Active Write-back workflow for DAC

Require: C_n, C_a

```

1: procedure SCAN AND ACTIVE WRITE-BACK( $C_n, C_a$ )
2:   repeat
3:     Get the cache entry from LRU in  $C_n$ :  $L_x$ 
4:     if  $N_{C_n} == 0$  || Dynamic Write-back Window (DWW)  $\leq N_{C_n}$  then
5:       RETURN & EXIT Scan and Active Write-back
6:     end if
7:     Get the LPA( $L_x$ )
8:     Channel_ID = LPA % Channel_N
9:     Chip_ID = (LPA/Channel_N) % Chip_N
10:    if The Chip(Channel_ID, Chip_ID) is in Busy then
11:      RETURN
12:    end if
13:    if The Chip Queue (Channel_ID, Chip_ID) is not empty then
14:      RETURN
15:    end if
16:    Active Write-back to NAND Flash Chip( $L_x$ )
17:    Migrate the  $L_x$  from  $C_n$  to the MRU in  $C_a$ 
18:     $L_x \leftarrow L_x.next$  /*Get the next cache entry in  $C_n$  */
19:    RETURN
20:  until  $L_x$  is in MRU
21: end procedure

```

When the data is written into the cache, DAC traverses region C_n to check whether the cache data meets the condition of active write-back operation in DAC. Fig. 3 shows the data flow of the active write-back, which updates cached data in the normal cold cache to flash memory when the flash chip is kept idle. Thus, the cache data is consistent with those stored in flash memory. When such cache data is removed, the cache data is directly deleted instead of synchronizing the data to flash memory. This procedure effectively reduces I/O-access conflicts when a large amount of data is written to flash memory, avoiding the high write latency.

The first condition to spark an active write-back operation is that cache entries in C_n is larger than 0 (see Lines 4–6 in Algorithm 3 and ① in Fig. 3). In case of the number of cache entries C_n being 0, there is no cache data in region C_n ; thus, the active write-back operation will not be invoked. The second condition to trigger the active write-back operation is that the size of the dynamic write-back window (DWW) is greater than the number of data entries in C_a (see Lines 4–6 in

Algorithm 3 and ② in Fig. 3). DWW is put forth to prolong SSD lifetime and obtain sufficient amount of written-back data in C_a amid the cache replacement. Fig. 3 unveils that the DWW value is two and there is a data page in area C_a , which meets the second condition to provoke an active-write operation. The active write-back operation synchronously writes cache data into flash memory when the flash chip is idle, thus shortening the latency of evicting data from the cache.

Currently, the existing caching schemes employ the passive update method to synchronize data between the cache and NAND Flash. This involves overwriting old data in the cache with new data when the cache becomes full and needs to make space for additional data. In the end storage, the data is stored persistently. This update method suffers a drawbacks — updating data from the cache to the flash memory relies on whether or not new data is written to the flash memory. When a large amount of data is moved into the flash memory, a massive amount of data will be written into the flash memory by the cache causing access conflicts coupled with increased response delay. To address these issues, we implement an active update module that proactively updates data in the cache to idle flash memory chips during low periods of application load. This approach effectively alleviates access conflicts and reduces response time. However, it is important to note that active write-back operations may increase the frequency of cache writes to the flash memory chip.

Compared with the passive update mode, DAC suppresses the amount of cache data to be actively written into the flash memory. An excessive active write-back operations impose a hefty overhead on data transmission among flash memory chips, impairing SSD lifetime. Given TLC or QLC flash memory,⁴ there is a limited number of program/erase cycles. We put forward a dynamic write-back window (DWW), which is one of the components in the active write-back module. DWW caters to cracking down the number of write operations in the DAC. DWW obtains enough written-back data in C_a during the cache replacement, enhancing the performance of write operations, curbing write amplification, and pushing down response time. Increasing the DWW size tends to shorten the response time at the cost of increasing write amplification. Conversely, when the DWW size drops, the response time rises while suppressing write amplification. DAC dynamically adjusts the DWW value according to the clean cache size in area C_a . Incorporating DWW, DAC is slated to curtail extra updates in active-write operations, thereby bolstering flash memory endurance and prolonging the SSD lifetime. The DWW is adjusted as follows

$$DWW = \begin{cases} \min\{[N_{C_n}] + [N_{C_a}], DWW + \lfloor \frac{N_{G_c}}{N_{G_h}} \rfloor\} & \text{if } N_{G_h} \neq 0 \text{ \& } N_{G_c} > N_{G_h} \\ \min\{[N_{C_n}] + [N_{C_a}], DWW + 1\} & \text{otherwise} \end{cases} \quad (5)$$

where the initial value of DWW is set to 0. Upon cache replacement where data in C_n is evicted, there is no cache data in C_a ; meanwhile, data being actively written back cannot meet the cache-replacement requirement. Therefore, it is prudent to enlarge the DWW size (see Lines 13–17 in Algorithm 2). When we have $N_{G_c} > N_{G_h}$ and $N_{G_h} \neq 0$, the DAC evicts cold data. The access pattern in the workload contains cold data. Since data in C_a is empty, the value of DWW reaches $\lfloor \frac{N_{G_c}}{N_{G_h}} \rfloor$ to avoid evicting dirty data in C_n in the cache replacement, prolonging the response time. In addition, active write-back operations are dedicated to cold data. The maximum DWW is the size of C_c (i.e., $N_{C_n} + N_{C_a}$), which indicates that the entire cold data is covered by DWW. If the condition $N_{G_c} < N_{G_h}$ or $N_{G_h} == 0$ holds, the value of DWW is increased by one.

Notably, the cache space G_c is a portion of cache C_c . If the amount of cold data in cache C_a exceeds the targeted value d , we prioritize replacing data in a portion of cache C_c . A larger DWW results in a

⁴ TLC: Triple-level cell flash memory. QLC: Quad-level cell flash memory.

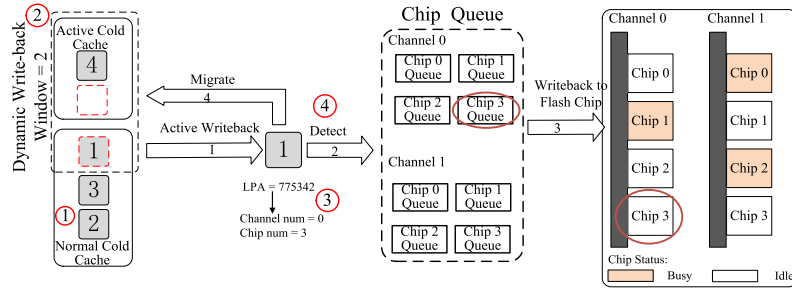


Fig. 3. The Active write-back flow in DAC.

higher number of active write-back operations when there is more cold data present. Therefore, we use N_{G_c}/N_{G_h} to represent the increment scale of DWW. Given a fixed sum of N_{G_c} and N_{G_h} , the ratio of the two metrics goes up when the N_{G_c} becomes large — meaning that a large amount of data needs to be actively written back in flash memory if cold data dominates I/Os access pattern in workloads. DAC adaptively adjusts the write-back threshold according to the hot-and-cold trends of the workloads. The size of the cold region expands in presence of an increasing amount of recently accessed cold data — and the growing DWW. The data in the window is updated to flash memory when the flash memory is idle. Then, clean data in C_a is generated. The cold and clean data in C_a is awarded a high priority to be replaced in the cache replacement procedure. In other words, the dirty data that would have been passively replaced is written back earlier when the flash memory is idle and simultaneously converted to clean data, minimizing the replacement delay.

Herein, we introduce the procedure to reduce the DWW size. A flash transaction hitting region C_a implies that C_a may have occupied a large cache space in C_c . Thus, it is crucial to restrict active write-back operations (see Lines 5–10 in Algorithm 4). The DWW value decrease by one under the condition that DWW is greater than 0. Thus, DWW decrements by 1 before drops to zero. DAC is adroit at adjusting DWW size to cater for dynamically changing access patterns. When the DWW condition is satisfied, the third condition to bring active write-back operations is to check whether cache data have any conflict with flash memory (see Lines 10–12 in Algorithm 3 and ③ in Fig. 3). The active-write back operation is in charge of actively updating cache data pages into the flash memory. When TSU schedules cache data pages in the flash memory chip queue, the channel and chip that are written into the data pages are occupied by the other transactions. An access conflict occurs in this scenario, delaying the writes of data pages into the flash memory. Parallel resources in the modern SSDs are very abundant. The culprit of access conflicts is that the data allocation schemes and TSU do not fully utilize parallel resources. Most existing SSD architectures employ static data allocation schemes. According to LPA of each cache data, we calculate chip addresses for to-be-written cache data. Fig. 3 depicts an example where the LPA of cache data entry 1 is 775342. The channel number and chip number, in this example, are 0 and 3, which are calculated by address allocation in CWDP [53,54].⁵ The CWDP formula is expressed as $(Channel_N = lpa \% Channel_N)$ and $(Chip_N = (lpa / Channel_N) \% Chip_N)$, where $Channel_N$ and $Chip_N$ represent the number of channels and chips in each channel, respectively. The address of the cache entry 1 to be written is chip 3 in channel 0. According to the $\bar{R}/B$ signal, the chip is idle — meeting the third condition of active write back. When the chip is busy, there is no active write-back operation while DAC is traversing the next cache entry in region C_n .

The fourth trigger condition of the active write-back operation is to determine that the chip queue of written-back data is empty (see Lines 13–15 in Algorithms 3 and ④ in Fig. 3). Before data entry is written

to flash memory, the entry is buffered in the corresponding queue of flash memory chip. The TSU module schedules the data to the flash memory chip. If the flash chip queue is non-empty, the data written to the corresponding chip is awaiting a scheduling decision. When an active write-back operation takes place, the cache entry initiating an active write back can be scheduled after the previous data scheduling is accomplished. The drawback of this strategy lies in long waiting time. When the flash memory chip queue is not empty, the active write-back operation will not be performed. When the cache entry meets the four trigger conditions (see Lines 16–21 in Algorithm 3), data will be actively written back to flash memory. Meanwhile, the cache entry is migrated from the C_n linked list to MRU in C_a . At this time, the active write-back operation for the cache entry is completed. When a new data item joins C_n , DAC traverses the linked list in C_n and performs an active write-back operation on the target cache entries.

4.4. Procedure of real cache hit

Algorithm 4 Real Cache Hit workflow for DAC

Require: *FlashTransaction* $T_1, T_2, T_3, \dots, T_i, \dots$

```

1: procedure CACHE_ALGORITHM( $T_i$ )
2:   repeat
3:     /*For every  $T_i$ , one of the three cases occur.  $T_i$  hit in  $C_n$ ,  $C_a$  or  $H_c$ */
4:     if  $T_i$  hit in  $C_a$  then
5:       /*Decrease the Dynamic Write-back Window (DWW) */
6:       if  $DWW > 0$  then
7:          $DWW = DWW - 1$ 
8:       end if
9:     end if
10:    Get the cache item from cache:  $L_x$ 
11:    Migrate the  $L_x$  to MRU in  $H_c$ 
12:    if  $T_i$  is Read Transaction then
13:      Return read data from Cache( $T_i$ )
14:    else
15:      Update cache data( $T_i$ ) /* $T_i$  is a write transaction*/
16:    end if
17:  until
18: end procedure

```

When read transaction tr hits C_a , C_n , or H_c , the cache entry has been accessed at least once (see Lines 13–15 in Algorithm 4). If there is a hit in C_a , the cache entry marked as clean is migrated to the MRU in H_c . Similarly, if tr hits C_n or H_c , the data may be a clean page or a dirty page, which depends on whether the data is brought into the cache due to a read miss or write miss. DAC does not mark the clean and dirty for the cache entry. The entry keeps the original state and migrates from C_n or H_c to MRU in H_c .

If transaction tr is a write request that hits C_a , C_n , or H_c , the cache page has been accessed at least once (see Lines 16–19 in Algorithm 4). Since the write request updates cache data, the cache entry is marked as dirty regardless of hits in the linked list; meanwhile, the cache entry is migrated from C_a , C_n or H_c to MRU in H_c . In DAC, the cache data accessed more than twice is referred to as hot data, which ought to be

⁵ In this case, there are two channels and four chips in each channel.

migrated to H_c . Access time is a summation of time spent in reading and writing data. Please refer to Algorithm 4 for the details.

4.5. Procedure of real cache miss and ghost cache hit

Algorithm 5 Real cache miss and ghost cache hit workflow for DAC

Require: *FlashTransaction* $T_1, T_2, T_3, \dots, T_i, \dots$

```

1: procedure CACHE_ALGORITHM( $T_i$ )
2:   repeat
3:     /*For every  $T_i$ , one of the three cases occur.  $T_i$  hit in  $G_c$  or  $G_h^*/$ 
4:     if  $T_i$  hit in  $G_c$  then
5:       if  $N_{G_c} \geq N_{G_h}$  then
6:          $d = \min\{d+1, m\}$ 
7:       else
8:          $d = \min\{d + \lfloor \frac{N_{G_h}}{N_{G_c}} \rfloor, m\}$ 
9:       end if
10:    else
11:      /* $T_i$  hit in  $G_h^*/$ 
12:      if  $N_{G_c} \leq N_{G_h}$  then
13:         $d = \max\{d-1, 0\}$ 
14:      else
15:         $d = \max\{d - \lfloor \frac{N_{G_c}}{N_{G_h}} \rfloor, 0\}$ 
16:      end if
17:    end if
18:    Eviction and Replacement Algorithm( $T_i$ )
19:    Insert to the MRU in  $H_c(T_i)$ 
20:    Scan and Active Write-back( $C_n, C_a$ )
21:  until
22: end procedure

```

When transaction tr hits the ghost cache and misses the real cache, there is no data in the cache. The access performance of tr is the same as that of real cache and ghost cache both miss. However, a hit in the ghost cache means that the data have been moved out of the real cache. The cache data is hot thanks to multiple recent accesses. When fetched into the real cache again, the data is inserted into MRU of H_c . Upon hits in the ghost cache, access patterns are determined. According to the hit ratio, DAC adjusts the sizes of C_c and H_c . Let us elaborate the process of hit in the ghost cache and miss in the real cache in DAC. First, DAC configures the target size d (see Lines 5–18 in Algorithm 5) in the cold link list. The size of the cold data link list (N_{G_c}) is constantly close to the target size d in this procedure. Second, a cache entry is evicted from the real cache by the eviction-replacement algorithm to free up cache space (see Line 19 in Algorithm 5). Transaction tr writes a new cache entry into the MRU of H_c in the real cache (see Line 20 in Algorithm 5). Finally, since a new data is loaded into the cache, DAC executes the scan& active write-back module to determine whether any cache needs to be actively written back. In what follows, We articulate DAC's strategy for adjusting target size d in the cold linked list.

When tr hits G_c , it presages that workload tends to access cold data. DAC enlarges the proportion of C_c in the cache by increasing the target size d of C_c . When tr hits G_h , DAC increases the proportion of H_c in the cache by cutting back the target size d of C_c ($(M - d)$ is the target size of H_c).

When tr hits G_c , if $N_{G_c} \geq N_{G_h}$ and target size d of C_c plus one does not exceed the size of cache space m , the value of d adjusts to $d+1$. If the value of $(d+1)$ exceeds m , the target size d will be set to the maximum value m . In case $N_{G_c} < N_{G_h}$, DAC increases the size of the cold data list and; then, the increment coefficient is set to $\lfloor \frac{N_{G_h}}{N_{G_c}} \rfloor$. If there is $d + \lfloor \frac{N_{G_h}}{N_{G_c}} \rfloor$ is lower than m , the value of d becomes $d + \lfloor \frac{N_{G_h}}{N_{G_c}} \rfloor$. In addition, target size d is lifted to the maximum value m when $d + \lfloor \frac{N_{G_h}}{N_{G_c}} \rfloor$ surpasses m .

If tr hits G_h , value d will be decreased to $(d-1)$ when condition $N_{G_c} \leq N_{G_h}$ holds and C_c 's target size is not below 0 after decreasing by one. The value of $(d-1)$ being lower than 0 prescribes that target size d must be dragged down to the minimum value of 0. In case

of $N_{G_c} > N_{G_h}$, DAC shrinks the cold data list's size. The reduction coefficient is set to $\lfloor \frac{N_{G_c}}{N_{G_h}} \rfloor$. When $d - \lfloor \frac{N_{G_c}}{N_{G_h}} \rfloor$ is 0 or above, the value of d is adjusted to $d - \lfloor \frac{N_{G_c}}{N_{G_h}} \rfloor$. If $d - \lfloor \frac{N_{G_c}}{N_{G_h}} \rfloor$ is smaller than 0, target size d will be configured to 0.

5. A case study for DAC

In this section, we employ a use case to present the benefits of DAC compared with the traditional LRU caching scheme. The advantage of DAC is centered around active write-back operations; thus, the cache data is updated to the flash memory on demand to originate clean data pages. The cache replacement firstly evicts clean data pages to enlarge the ratio of replacing clean data, which slashes response times. Through a case study we analyze the benefits of the DAC cache management scheme compared against the traditional LRU. Let us consider three write requests #1, #2, and #3 arriving at timestamp t_0 , t_1 , and t_2 , respectively. There is a long time interval from t_0 to t_1 and t_2 , but the distance between t_1 and t_2 is close. Request #1 includes one write transaction and its corresponding LPA is one. Request #2 is comprised of four write transactions and their LPAs are 24, 25, 26, and 27, respectively. Request #3 consists of three write transactions with LPAs 51, 52, and 53. Figs. 4 reveals the procedures of the LRU and DAC caching scheme. Each caching strategy includes the cache data-flow in a chronological order and $\bar{R}/B$ status of each chip within a period.

5.1. LRU caching scheme

The left-hand side of Fig. 4 delineates the procedure of the LRU. It is assumed that the cache stores eight data entries. Prior to timestamp t_0 , seven data entries are fetched into the cache. All the cached data are dirty data with LPAs 12, 13, 14, 15, 21, 22 and 23, respectively.

Timestamp t_0 : Write request #1 arrives, LPA 1 is loaded into MRU at stage 1.1 (see s1.1 on the left in Fig. 4) because there is free cache space. No other requests exist in a short period after request #1 is processed; thus, the cache and flash memory are in the idle state during the period from t_0 to t_1 .

Timestamp t_1 : Request #2 arrives. This request contains four write transactions that miss the cache. Four cache data entries (12, 13, 14, and 15) are evicted from the LRU. Per CWDP-allocation rule, these four cache data are placed into queues A2, A3, B2, and B3 (see s2.1). Since chips A2, A3, B2, and B3 are idle, TSU schedules the transaction in the corresponding queue and writes the data to the flash chip in stage 2.2 (see s2.2). In the experiment, we set the program latency as 750us in flash memory. Besides, the time interval between t_1 and t_3 is the duration of a page program. Meanwhile, the interval between t_0 and t_1 is the same as the duration of a page program. At this time, the cache space frees up four cache entries. Request #2 is written to the MRU in the cache. After written in the cache, the cache entries' order becomes 21, 22, 23, 1, 24, 25, 26, 27 (from LRU to MRU).

Timestamp t_2 : Request #3 arrives. The procedure is similar to that of timestamp t_1 . Request #3 embraces three write transactions, each of which misses the cache. Therefore, these free cache entries are used by request #3. Three evicted cache entries (21, 22, and 23) are placed into queues A3, B2, and B3. However, the chip profiler discovers that the chip is busy according to $\bar{R}/B$ signal because the cache entries 12, 13, 14, and 15 are written into flash memory. Thus, the cache data are blocked to be written into flash memory in stage 3.1 (see S3.1). There are no free cache space, request #3 becomes pending until the cache can free space up. At timestamp t_3 , cache data 12, 13, 14, and 15 are written to flash memory. Then chips A2, A3, B2, and B3 are idle while the TSU schedules transactions 21, 22, and 23 in the queue and flushes them to flash memory at stage 3.3 (see S3.3). The cache space is freed to service request #3 in stage 3.2 (see S3.2). Cache entries include 21, 22, 23, 1, 24, 25, 26, and 27. Meanwhile, flash chips A3, B2, and B3 are busy until timestamp t_4 . With the LRU scheme, requests #1, #2, and #3 are serviced at timestamp t_3 .

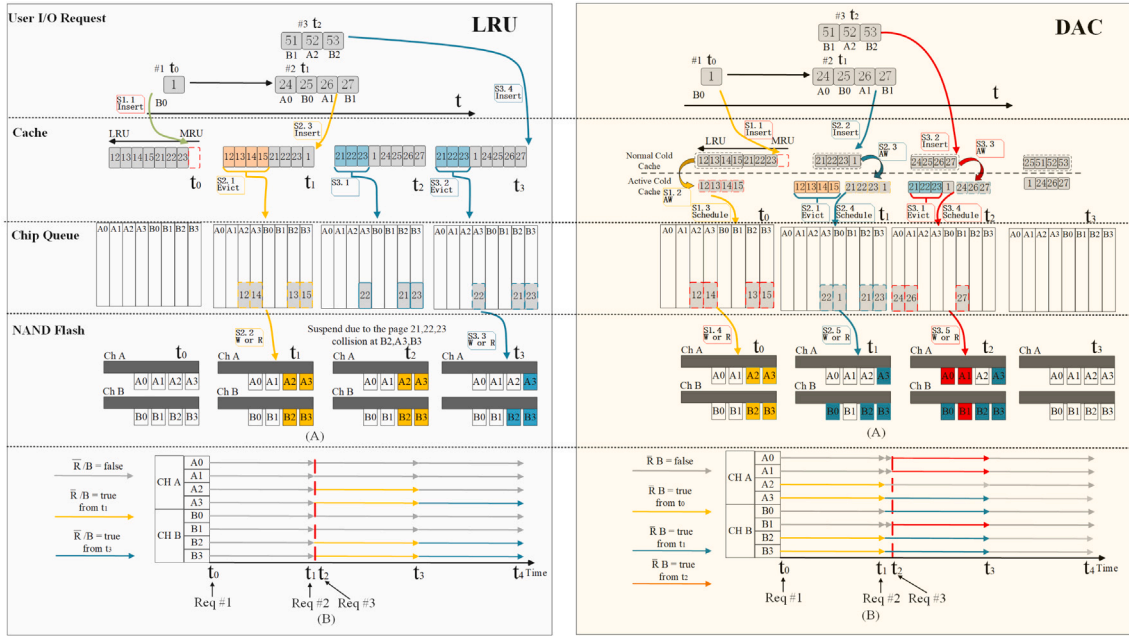


Fig. 4. The comparison between data-flow of LRU and DAC caching scheme in a use-case.

5.2. DAC caching scheme

Now, we are positioned to articulate the data flow in the DAC caching scheme. Initially, the cache size is the same as that of LRU. Data entries 12, 13, 14, 15, 21, 22, and 23 are loaded in C_n . The sizes of C_a , H_c , and d are 0 and the DWW size is set to 4. To highlight the advantages of DAC in this case, regions G_c , G_h , and H_c are not presented in Fig. 4.

Timestamp t0: Request #1 arrives. Because cache space is available, request #1 is directly inserted into the MRU in C_n (see S1.1). Cache entries *i.e.*, 12, 13, 14, 15, 21, 22, 23 are pushed into C_n ; then DAC embarked on the scan and active write-back module. Because DWW is 4, DAC checks whether there are four cache data meeting the active write-back condition in LRU's C_n . Since the flash chip and chip queue is idle, cache entries 12, 13, 14, and 15 satisfy the active write-back condition and the active write-back procedure is running at stage 1.2. The cache entries that are actively written back are placed in chip queues A2, A3, and B3 (see S1.3). The TSU schedules data entries to be written to the corresponding flash memory chips (see S1.4). The active write-back procedure migrates cache entries to MRU of C_a from C_n . Data entries 12, 13, 14, 15 are placed in C_a , whereas data 21, 22, 23, and 1 are in C_n . Compared with LRU at timestamp t0, DAC actively writes back four cache pages during the interval from t0 to t1, producing four clean data; however, LRU with the empty space is waiting during the period between t0 and t1.

Timestamp t1: Request #2 arrives. Four cache entries need to be freed up. Governed by the replacement algorithm in DAC, four cache data (12, 13, 14, and 15) are evicted from C_a . Since these cache data have been actively written back into flash memory, the data are directly removed from the cache and the corresponding LPA metadata are funneled into G_c (see S2.1). When free space is handy, the four write transactions of request #2 will be added into the MRU of C_n (see S2.2). These inserted cache entries *i.e.*, 21, 22, 23, 1, 24, 25, 26, and 27 channeled in C_n .

The DAC sparks the scan and active write-back module in stage 2.3 (see S2.3) to detect whether there exist data entries to be actively written back into flash memory. Since there is a page-program latency between t1 and t0, cache entries 12, 13, 14, and 15 initiated at timestamp t0 have been actively written back into flash memory. At this time, both the chip and chip queue are in the idle state and; then, data

entries 21, 22, 23, and 1 satisfy the active write-back condition. The active write-back procedure is undertaken in stages 2.4–2.5. Afterward, C_a contains cache data 21, 22, 23, and 1. C_n holds cache data 24, 25, 26, and 27. Compared with LRU in timestamp t1, DAC directly evicts 12, 13, 14, and 15 amid the cache replacement. However, LRU must write the four cache data to corresponding flash memory, which occupies the four chips from timestamp t1 to t3. In contrast, DAC proactively writes back four data pages at timestamp t1, yielding four clean cache data.

Timestamp t2: It is similar to the procedure at timestamp t1. Request #3 arrives at the cache and data entries ought to be free up. DAC kicks out three cache data entries in stage 3.1 (see S3.1) from C_a , *i.e.*, 21, 22, and 23. Since these cache data have been actively written back to flash memory, the data are eliminated from the cache and their LPA metadata are channeled into G_c . When there is sufficient free space, the three write transactions in request #3 will be inserted into the MRU of C_n (see S3.2). Requests #1, #2, and #3 are all serviced. At this point, the cache space is composed of C_n and C_a . Region C_a stores cache data entry 1. C_n retains cache data 24, 25, 26, 27, 51, 52, and 53. DAC invokes the scan and active write-back module (see S3.3). DWW has the same value (*i.e.*, 4) as the initial one. The active write-back module can write three cache entries back into flash memory. Since flash chips A3, B0, B2, and B3 actively conduct write-back operations for data originated at timestamp t0. Data entries 24, 25, 26, and 27 are written into chips A0, B0, A1, and B1, respectively. Therefore, data entry 25 does not meet active write back condition. Chips A0, A1, and B1 are empty; thus, data entries 24, 26, and 27 should be registered in these chips under the active write-back condition (see S3.3–S3.5). When the active write-back operation is completed, region C_a stores cache data 1, 24, 26, and 27. Region C_n keeps 25, 51, 52, and 53. All the flash chips finish program operations before timestamp t3. Compared with LRU at timestamp t2, DAC accomplishes request #3 at timestamp t2; meanwhile, the three cache entries are written back into flash memory. Upon the arrival of an incoming new request, DAC responds to the request in a more swift way than LRU.

With respect to request #1, #2, and #3, DAC and LRU complete all the requests at timestamp t2 and t3, respectively. All the operations in flash memory are finished in the DAC and LRU cases at timestamp t3 and t4, respectively. When a new request #4 arrives at timestamp t3 or t4, there are cache data in C_a . Nevertheless, DAC can evict these

Table 3
Simulation parameters in MQSim.

SSD Organization	Host interface: PCIe 3.0 (NVMe 1.2). Capacity: 24 GB. 6 channels 4 chips/channel
Flash Interface	ONFI 3.1
Flash Structural	4 KB/page with 448 B metadata, 64 pages/block, 1024 blocks/plane, 2 planes/die, 2 dies/chip
Flash Latency [57]	Read latency: 75 μ s, Program latency: 750 μ s, Erase latency: 3.8 ms
Flash translation layer	GC Policy: Greedy [58]. GC Threshold: 0.05. Mapping: DFTL [21]. TSU Policy: Sprinkler [59]. Over provisioning Ratio: 0.07

Table 4
Workload Characteristics.

Trace	Total I/Os ($\times 10^6$)	Write size (GB)	Write ratio	Avg.Size (KB)
hm_0	3.99	20.47	0.64	7.99
proj_0	4.22	144.26	0.87	38.03
prxy_0	12.52	53.79	0.96	4.76
stg_0	2.03	15.09	0.84	11.57
ts_0	1.80	4.13	0.82	9.00
usr_0	2.24	35.30	0.59	22.66
web_0	2.03	11.67	0.70	14.99
Fin1	5.33	14.57	0.77	3.38
src2_0	1.56	9.34	0.89	7.21
hm_1	0.61	0.54	0.04	15.16
websearch1	1.06	0.0017	0.0002	15.24
2016021615-LUN0	1.07	5.68	0.73	13.42
2016031614-LUN0	2.31	18.90	0.70	15.47
2016030117-LUN2	2.70	13.82	0.69	12.50
2016022113-LUN3	4.24	2.22	0.06	25.06
2016030415-LUN2	1.83	7.97	0.66	13.57
2016030819-LUN2	2.90	13.90	0.79	10.91
2016030917-LUN2	2.47	10.37	0.69	11.32

cache data rather than write back into flash memory. LRU may cause access conflicts while enlarging response time due to page programming in some flash chips. In this case study, we reckon that DAC has a competitive edge over LRU in terms of I/O response time.

6. Experimental setup and evaluation

The experimental results presented in this section quantitatively demonstrate the DAC's performance strengths. We start this section with the experimental setup followed by the performance and scalability analysis.

6.1. Experimental setup

We implement DAC in an open-source multi-queue SSD simulator MQSim [55,56] to carry out performance evaluation. MQSim accurately simulates the latest NVMe SSD prototypes with front-end and back-end components in multi-queue SSDs. In our experiments, the default cache size is 32 MB; Table 3 summarizes a list of simulation parameters in MQSim.

We utilize realistic enterprise-scale traces to evaluate the performance of caching schemes. In the empirical studies, we elect traces from the UMass trace [60], the I/O traces of enterprise servers at Microsoft Cambridge Research [61,62], and the Syster'17 trace [51]. DAC is adroit at boosting cache performance under workloads with high write-ratio I/Os because the cache mostly stores clean data under high-read workloads. However, the DAC introduces a dynamically active-update operation to convert dirty data in the cache to clean ones, optimizing the cache performance. We employ 11 workloads with nine high-write-ratio workloads and two high-read-ratio workloads. In

addition, we choose seven traces reported in prior studies [51], which unveil that I/Os pattern are burst with a larger amount of data per unit time. The extensive experiments are conducted to validate our implemented DAC. Characteristics in traces are listed in Table 4. The workloads evaluated in this study primarily include write-intensive and burst-access requests. Under these workloads, clean data rarely exists in the cache, incurring a high cost in cache replacement and long response latency. In addition, the large-scale cache replacements generated by burst access tend to significantly increase the performance cliff in term of response time. DAC is designed to revamp the performance of workloads with unbalanced write requests and burst-access write I/Os. DAC takes advantage of the idle time of flash memory to write cold and dirty data back to flash memory earlier — converting such data to clean ones to minimize the cost of cache replacement and mitigating the performance cliff during burst accesses. In addition, these workloads were tested in the previous studies on alternative caching strategies in NAND flash-based SSDs. Hence, we opt for validating the benefits of DAC by applying these workload conditions.

In the experiments, we compare DAC against LRU, CFLRU [13], GCaR-CFLRU [17], LCR [18], ARC [63], AD-LRU [16], and MQSim [55]. Please refer to Section 4 for the introduction on LRU, CFLRU, GCaR-CFLRU, LCR, ARC, and AD-LRU. MQSim — a default caching scheme in the MQ-SSD simulator — adopts a variant of write-through strategy combined with a cold-and-hot separation unit. Such a strategy orchestrated in MQSim differs from the traditional write-through cache management. MQSim injects new cache data into the head of the LRU link list while discerning whether the data is hot or cold according to the bloom filter. The default scheme promptly writes back cold-dirty data and deletes data that have been written to flash memory. Afterward, corresponding cache data that have been written back into the flash memory are swiftly deleted to maximize free space for newly arriving data in the cache. The new data is directly written into the cache rather than evicting any cache data. Compared with the traditional write-through scheme, MQSim's cache manager greatly curbs data replacement cost and response time. MQSim's mechanism, however, sparks a large amount of data written into flash memory, deteriorating the write amplification (5 (a)), erase count (5 (b)), and SSD lifetime. It is undoubtedly true that MQSim optimizes SSD performance at the cost of write amplification and SSD lifetime. The empirical experiments demonstrate that DAC achieves optimal performance in terms of the five metrics (C_a , C_n , H_c , G_c , and G_h). Initially, we set the crucial parameters such as C_a , C_n , H_c , G_c , and G_h , d , and DWW to zero. In the case of dynamically changing workloads, these parameters are adjusted according to I/O patterns to lift system performance.

The Co-Active cache scheme [44] proactively writes cold and dirty data back into flash memory that is sitting idle through the active write-back module. Idle bandwidth is utilized to produce clean data, thereby suppressing the cost of cache replacement. Under complex workloads, however, I/Os patterns change so frequently that Co-Active is forced to manually adjust the static write-back thresholds to maintain high performance under workloads with varying I/Os patterns. Co-Active faces a predicament to distinguish hot and cold data in the workload, making it arduous to adapt to any changes in I/Os patterns. We propose the dynamically active and collaborative cache management — DAC — to address this issue. DAC is adept at effectively regulating the threshold of active write-back — the dynamic write-back window — according to I/O patterns of the workloads. One strength of DAC is to avert manual threshold tunings under dynamically changing workloads. DAC progressively obtains hot and cold I/O trends, which guides the adjustment of the ratio between hot and cold regions in the cache to achieve high cache performance.

In our experiments, we randomly pick an array of traces to be blended into I/O streams, confining varying I/O patterns in the workload. The wide variety of traces are listed in Table 5 as Mix1-Mix10. We delve into the performance comparison between DAC and Co-Active under these mixed workloads. To tune cache performance, we manually configure the write-back thresholds in Co-Active.

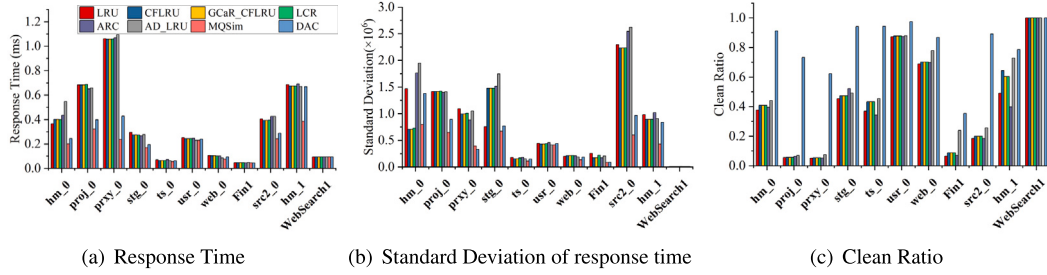


Fig. 5. Average Response Time (a), Standard Deviation of Response Time (b) and Clean Ratio (c) of DAC and the alternative caching schemes under eleven traces. The low response time and standard deviation of response time signifies good performance.

Table 5
Mixed Workloads.

Mixed workload	Workloads
Mix1	src2_0, web_0
Mix2	hm_0, stg_0
Mix3	hm_0, stg_0, src2_0
Mix4	hm_0, stg_0, WebSearch1
Mix5	hm_0, hm_1, stg_0, ts_0
Mix6	hm_0, ts_0, src2_0, WebSearch1
Mix7	2016021615-LUN0, 2016031614-LUN0, stg_0, ts_0
Mix8	2016021615-LUN0, 2016031614-LUN0, 2016030117-LUN2, 2016022113-LUN3
Mix9	2016021615-LUN0, 2016031614-LUN0
Mix10	2016031614-LUN0, 2016030117-LUN2, 2016022113-LUN3

6.2. Tested metrics

Average response time indicates the performance of an SSD from the perspective of individual I/O requests. This metric measures the time interval between an I/O request entering the SQ queue and the host receiving feedback from the SSD. In this test, we standardize experimental parameters (e.g., average response time, the standard deviation of response time, erase count) to study the performance of DAC and alternative caching schemes. We use tested metrics of LRU under hm_0 as the standardized one. **The standard deviation of response time** quantifies the degree of an SSD performance cliff. **Write amplification** is a ratio between the amount of data written to the flash memory and the amount of data issued by a host. A low value indicates that the amount of data written to the flash memory is small. Write amplification reflects impacts of the tested cache management schemes on SSD lifetime. **Erase count** in flash memory is the most conspicuous manifestation of SSD lifetimes. Each block in flash memory has a maximum number of erase operations. **Write (Read) hit ratio** is the probability of a I/O request hitting the cache. **Clean ratio** suggests a proportion of clean data that are evicted when cache replacements occur.

6.3. Performance evaluation

6.3.1. Response time

The term response time refers to a time interval for an SSD to react after I/O requests enter the SQ queue. In the MQSim simulator, response time is a period between the time when an I/O request is inserted into the SQ queue and the time when the data is injected into the head of the cache link list. The SSD returns a CQ signal to the host before the SSD control writes data to flash memory.

Fig. 5(a) unveils the comparison of DAC with the other caching schemes in terms of response time under an array of eleven workloads. The response time of DAC is faster than those of the other caching schemes in most cases. For example, DAC improves the response time of LRU, CFLRU, GCar-CFLRU, LCR, ARC, and AD-LRU under the prxy_0

workload by 41.8%, 41.8%, 41.6%, 42.06%, 38.98%, and 39.60%, respectively. DAC has a higher response time by 17.5% than MQSim but MQSim shortens response time at the cost of SSD lifetime. In Figs. 7(a) and 7(b) in Section 6.3.2, under hm_0, the write amplification and erase count in MQSim is 35.1% and 52.5% higher than those of DAC, respectively. The experimental results are evident that DAC has an obvious edge in response time over the other caching schemes. DAC slightly reduces the response time of LRU, CFLRU, GCar-CFLRU, LCR, and ARC by 2.4%, 0.8%, 0.8%, 0.8%, and 3.4%, respectively, under hm_1 with high-read I/Os. The response time performance of all the caching schemes share similar results under Websearch1 because the write ratio is around 0.02%. This trace hardly triggers active-write operations in DAC, but the caching scheme cannot impair performance.

The standard deviation of response time is applied to gauge the performance cliff of the caching schemes. Fig. 5(b) reveals the performance cliff of DAC and the other caching schemes. Under the prxy_0 workload, DAC optimizes the response time cliff of LRU, CFLRU, GCar-CFLRU, LCR, ARC, AD-LRU, and MQSim by up to 69.98%, 66.92%, 66.93%, 67.76%, 68.71%, and 16.4%, respectively.

In addition, under write-intensive workloads where there is more dirty data, it is favorable to invoke active write-back procedure to yield a large amount of clean data amid an idle period. Consequently, access conflicts between user-level I/Os and flash memory I/Os are mitigated while shortening response time. Under read-intensive workload, DAC is dynamically cognizant of I/O-access locality derived from the cache metadata in the ghost regions. DAC perceives whether the subsequent access I/O is a recent I/O or the frequently accessed one; then, cache replacement is performed to bolster the hit ratio.

The clean ratio represents the proportion of clean data replaced during cache replacement. Recall that replacement cost of clean data is lower than that of dirty data (see Section 3). CFLRU, GCar-CFLRU, LCR, and AD-LRU preferentially replace clean data, but clean data is entirely generated by reads that fetch data into cache from flash memory. This replacement method is passive in nature, causing a low proportion of replaced clean data. Fig. 5(c) shows the clean ratios of DAC along with the other caching schemes. The active write-back mechanism is adroit at producing clean data. There is no doubt that the clean ratio climbs under write-intensive workloads rather than read-intensive workloads. Because MQSim only stores written data, all cached data is marked as dirty. Hence, only dirty data pages are replaced. In this experiment, the clean ratio in MQSim is 0, which is a default setting. In contrast, under proj_0, DAC improves the clean ratio of LRU, CFLRU, GCar-CFLRU, LCR, ARC, AD-LRU by up to a factor of 12x, 11x, 11x, 10x, 9x. Under high-read-ratio hm_1 and Websearch1, the clean ratio of DAC has no obvious advantages and the ratio of clean data is high, which is hardly trigger active-write operations. The results reveal that the mechanism of dynamic updating cache data into flash memory effectively enhances the replacement ratio of clean data and curtails cache replacement cost.

The default cache management of MQSim is a variant of the write-through mode, in which data is fetched from flash memory to server user requests rather than storing data in the cache. Thus, the default clean ratio is envisioned as 0 in MQSim.

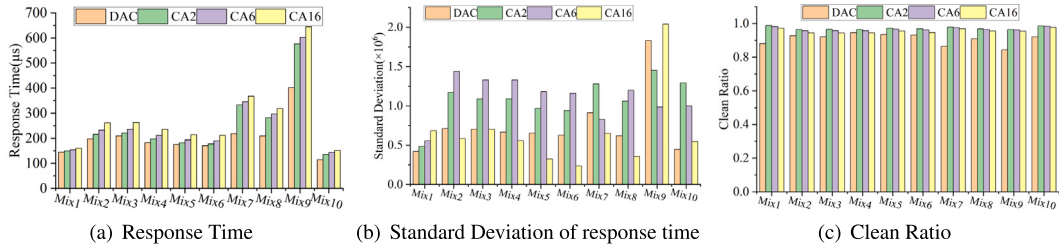


Fig. 6. Average Response Time (a), Standard Deviation of Response Time (b) and Clean Ratio (c) of DAC and the alternative caching schemes under ten mixed traces. The low response time and standard deviation of response time signifies good performance.

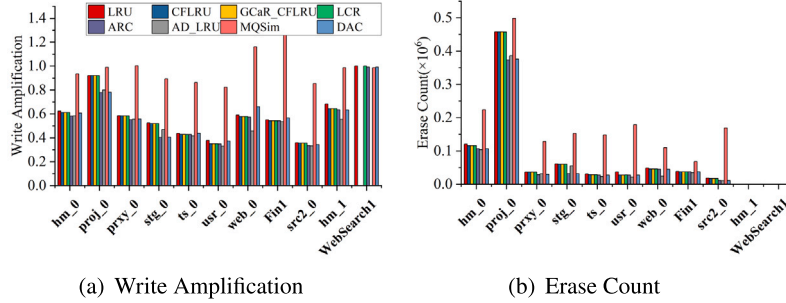


Fig. 7. Write Amplification (a) and Erase Count (b) of DAC and the alternative caching schemes under eleven traces. The low Write Amplification and Erase Count signifies good performance.

In addition, we study the performance comparison between DAC and Co-Active under the mixed workloads tabulated in Table 5. In this test, we manually set up three static values of write-back thresholds to optimize cache performance. Under the combined workload, we configure Co-Active's threshold as 1/2, 1/6, and 1/16, which are denoted as CA2, CA6, and CA16. Fig. 6 shows that compared with CA2, CA6, and CA16, DACs curtails the response time by an average of 13.7%, 18.5%, and 25.1%, up to 34.2%, 36.7%, and 40.4%, respectively. When it comes to the Microsoft's mixed traces and the Syster'17 traces, DACs optimizes CA2 by averages of 5.3% and 26.4%, respectively. In terms of standard deviation of response time, DAC enhances the above solutions by averages of 30.1%, 26.9% and -11.2%. DAC can reduce the performance cliff of response time in most cases. The experimental results unveil that DAC delivers high performance under mixed workloads thanks to judiciously regulated write-back windows, catering to various I/Os patterns. DAC's clean ratio is low because clean pages are mostly generated by active write-back operations. Higher clean rates indicate that more active write-back operations are triggered — increasing write amplification with deteriorated SSD lifetime. When the write-back threshold is small, DAC is superior to Co-Active in term of response time. In summary, DAC delivers optimized performance without having to manually adjust write-back thresholds.

6.3.2. SSD lifetime

NAND flash memory has a limited number of program and erasure cycles (P/Es). In addition to response time, lifetime is one of the critical metrics to evaluate the quality of cache design. Write amplification and erase count have a paramount effect on SSD lifespan. Write amplification refers to the ratio between the amount of data written into a flash memory chip and host-side data in workload. A small write amplification value signifies less data written to flash memory. Erase count — an intuitive metric measuring SSD lifetime — is defined as the number of block erasures. In flash memory, manufacturers specify a limited number of P/Es as an endurance. When a tested cache scheme consumes more erase counts after runtime, the remaining lifetime in the SSD becomes shorter. Thus, it can be found that the lower erase count indicates the longer SSD lifetime. Figs. 7(a) and 7(b) unravel the write amplification and erase count comparisons between DAC and the other

competitors. MQSim offers a clear weakness in SSD lifetime compared with the other counterparts. Under the eleven workloads, DAC reduces erase count by 52.5%, 24.6%, 76.4%, 79.1%, 81.1%, 84.4%, 59.2%, 45.3%, and 93.3%, respectively compared with MQSim. Under hm_1 and Websearch1, there are too few write I/Os to trigger garbage collection, thereby the number of erase count is zero. In addition, when there are no I/Os written into flash memory, the value of write amplification is also zero. DAC delivers an erase-count optimization over the alternative solutions by an average of 17.04%, because DAC dynamically and judiciously make replacement decisions of hot and cold data according to the ghost cache's historical data, which help in lifting hit ratio and curbing the amount of data written to flash memory. Besides, the active write-back module is adroit at configuring the sizes of cold data and the DWW window, idle flash chip, and empty chip queue, thereby prolonging SSD lifespan.

Under write-intensive workloads, the active write-back module in DAC flushes data into flash memory in advance. Therefore, DAC expands clean data volume to prune data that are passively written back. In other words, the amount of written data is compressed in DAC. Under read-intensive workload, a large clean data size diminishes of the advantages of DAC. Nevertheless, DAC — is slated to sense the hot and cold data distribution and to evict data, boosting the hit rate and curbing write amplification. Under the tested workloads, the active write-back module slashes the number of write operations and write amplification, which in turn cuts back erase count.

Both write amplification and erase count entail the SSD lifespan, so we study the lifetime impact of DAC through the perspective of write amplification under mixed traces. Fig. 8 reveals that DAC embraces a similar write amplification to the other alternative caching schemes. DAC reduces the average write amplification by 1.7% under the Microsoft traces, and that measure is increased by an average of 4.6% in the SYSTOR'17 case. DAC greatly slashes the response time at a small cost of lifetime, thereby revamping the SSD overall performance. DAC adaptively adjusts the hot-and-cold linked list's length according to I/Os access patterns. DAC is primed to bump up the write-back threshold when there is an excessive amount of cold data; otherwise, the threshold is downshifted to accommodate an increasing amount of hot data. Then, old data are written back to the flash memory as much as possible — cutting down response time with a slight increment in write amplification.

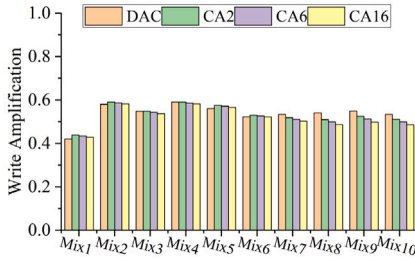


Fig. 8. Write Amplification Comparison between DAC and Co-Active (CA for short) with active write-back thresholds.

6.3.3. Hit ratio

Fig. 9(a) shows the write hit ratios of all the evaluated caching policies. DAC lifts write hit ratios of LRU, CFLRU, GCaR-CFLRU, LCR, ARC, AD-LRU, and MQSim by averages of 22.4%, 21.6%, 22.4%, 21.1%, -0.9%, 2.1% and 2.7x, respectively. The write hit ratio of MQSim is the lowest among its peers, because MQSim deletes cache data when the data is written to flash memory. The cache space of MQSim is under utilized in most cases, thereby pushing down write hit ratio. Fig. 9(b) illustrates the read hit ratios of DAC as well as all the alternatives under the eleven workloads. Compared with LRU, CFLRU, GCaR-CFLRU, LCR, ARC, AD-LRU and MQSim, DAC revamps the read hit ratios by averages of 39.4%, 35.2%, 39.1%, 78.6%, 2%, 2.4x, and 37.3x. Interestingly, the read hit ratio of MQSim is significantly lower than that of DAC under the tested workload conditions. This trend is attributed to the fact that MQSim has no intent to cache any read data. Thanks to the spatial and temporal locality, read request data is likely to be accessed in the future; thus, MQSim exhibits the dismay performance in read hit ratio.

DAC has similar data replacement strategy to that of ARC; thus, the read hit ratio of DAC is close to that of ARC. DAC wisely evicts cache data in cold/hot regions catered for access patterns. Thus, the hit ratio of cached data is boosted while coming I/O requests are serviced. But DAC is equipped with the active-write-back module which improves response time by 43.5%, 38.9%, 59.9%, 26.69%, 12.5%, 2.83%, 6.93%, 4.3%, and 2%, compared with that of ARC. DAC is designed with dynamically partitioning the hot and cold region, which is favor to aware the access locality in workloads and the cold data or hot data that will be accessed recently. Then, DAC dynamically manages cache space and increases the hit ratio.

Fig. 10(a) illustrates that DAC's write hit ratio is reduced by an average of 8.1% compared to the other competitors. Nevertheless, DAC greatly improves the read hit ratio by a factor of 5 thanks to the two-fold reason. First, Co-Active adopts a write cache mode, and data read from flash memory is not cached. It has much space to cache write requests; therefore, the write hit ratio is higher than DAC. Second, the DAC separates the clean data generated by active write-back from the clean data read by the flash memory — averting an immediate replacement of data that are just read from the flash memory. Thus, DAC optimizes the read hit ratio, see Fig. 10(b).

6.4. Scalability

We also study scalability performance of DAC from the perspective of cache size, DWW window size, and the eviction policy. We opt for a workload condition with two high write ratio (*i.e.* prxy_0 and hm_0) and another two presenting a high read ratio (*i.e.*, hm_1 and websearch1) to perform the scalability tests.

6.4.1. Cache size

In this study, we aim to explore that the DAC can demonstrate a lower response time using the same cache size compared with other alternative cache schemes. Meanwhile, DAC occupies a smaller memory space to achieve the same response time. We set four cache size – 2 MB, 8 MB, 32 MB, and 128 MB – to investigate the performance in terms of response time, erase count, and clean ratio. Figs. 11(a) and 11(b) demonstrate that the response time is shortened with an increasing cache size under prxy_0. The 128-MB cache enables DAC to cut back the response time by approximately 33.5%, 27.33%, and 7.7% compared with 2-MB, 8-MB, and 32-MB cache. Unsurprisingly, the 128-MB cache improves the erase-count performance of the other smaller caches by 63.9%, 59.05%, and 53.3%. It is undeniable that DAC's cache performance can be straightforwardly enhanced by increasing cache size. Under Websearch1, cache size has no obvious impact on the performance of DAC. The reason is that Websearch1 has a high read ratio and low read-hit ratio, see Fig. 9(a); thus, increasing cache size cannot boost the performance. In addition, there is low write ratio in hm_1 and Websearch1, which cannot trigger garbage collection inside an SSD. Thus, the value of erase count is zero.

Under the hm_1 workload, more clean data is accessed from flash memory when the cache size expands; consequently, the clean ratio grows. Given a large-sized cache accompanied by enormous active write-back operations, the clean ratio surges under high-write ratio workloads. As the cache size gradually increases, however, the cache with a low clean ratio still embraces much clean data, which limit active write-back operations. In case of a large cache size, a whole lot data is hardly evicted from the cache. Thus, there is no doubt that massive amount of hot data suppress the active write-back operations. As a result, the clean ratio slightly abates to a stable value.

6.4.2. Benefit of DWW

DWW is one of decisive parameters for active write-back operations in DAC. Because DAC dynamically adjusts DWW, we set two groups of experiments to explore the impact of DWW on performance. First, we elect a DWW window of 0 (DWW_min), which means no active write back. Second, we bump DWW to the maximum level (DWW_max), implying that active write-back operations have unrestricted window size. Fig. 12(a) unveils that under hm_0 workload. We find that DWW_min increases response time of DAC by 43.6% and slashes the erase count by 7.5%. The results entail that the active write-back operations in DAC can incur the cost of very few erase counts, but the operations significantly shorten response time by 43.6%.

DWW_Max maintains a similar response-time performance as that of DAC; nonetheless, but DWW_Max's erase count is 10.0% higher than that of DAC. We observe that a small DWW window size causes stringent effect for active write-back operations. Under read-intensive hm_1 and Websearch1, different size of DWW has little impact on performance because there is few dirty pages under read-intensive workloads, which hardly triggers active-write operations. The results confirm that the dynamic adjustment of DWW is reasonable and desirable for speeding up cache performance.

6.4.3. Benefits of adjusting Hot/Cold region

DAC is conducive to evicting cache entries from the cold and hot regions according to access patterns and hot-and-cold data distribution in the cache. We undertake two groups of experiments to quantify DAC's benefits of adjusting cold and hot regions. First, we set the strategy (DAC-C) to evict cache data in the cold region. Second, we test an eviction policy (DAC-H) to evict cache data from the hot region. Fig. 13(a) illustrates that under hm_0, the response time of DAC is 49.2% lower than that of DAC-C; DAC-C reduces the erase count of DAC by 4.2%. The response time of DAC is 6.4% smaller than that of DAC-H, whereas DAC-H offers an erase-count increase over DAC by 55.16%. Erase count of DAC-H sharply deteriorating because the priority of replacing data in the hot region leads to a significant decrease in hit

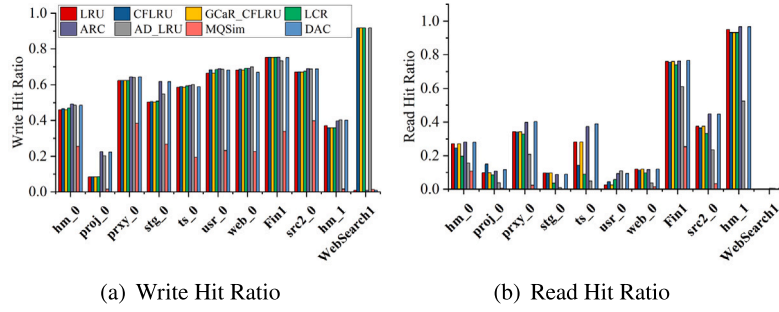


Fig. 9. Write Hit Ratio (a) and Read Hit Ratio (b) of DAC and the alternative caching schemes under eleven traces. The high hit ratio signifies good performance.

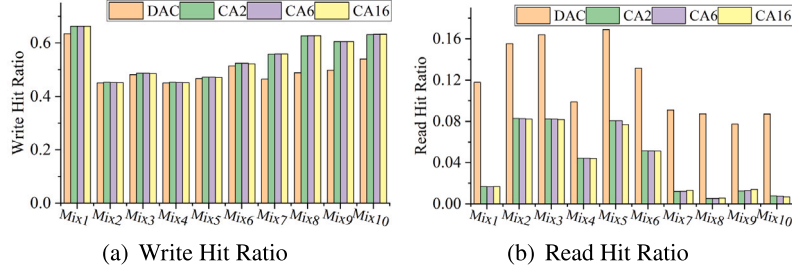


Fig. 10. Write Hit Ratio (a) and Read Hit Ratio (b) of DAC and the alternative caching schemes under ten mixed traces.

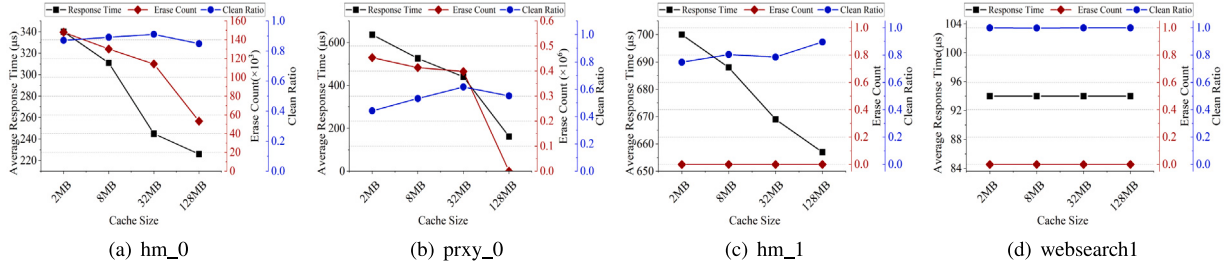


Fig. 11. The impact of cache size on DAC performance. Superb performance is manifested by low average response time, low write amplification, and high clean ratio.

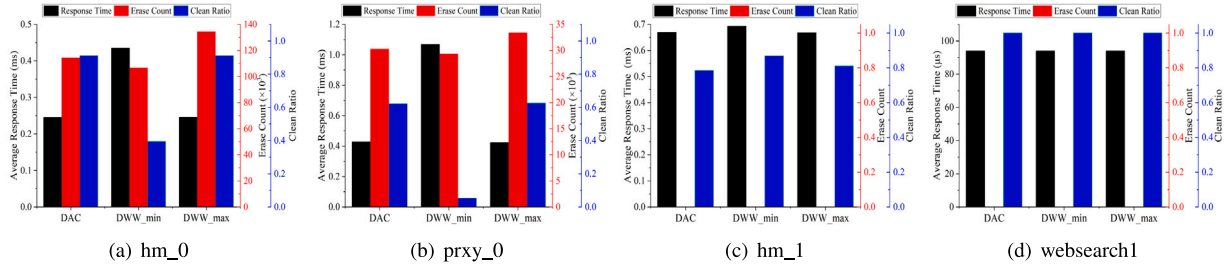


Fig. 12. The impact of dynamic write-back window (DWW) on DAC performance. Superb performance is manifested by low average response time, low write amplification, and high clean ratio.

ratio. Cache-miss I/Os aggravate the burden on the performance of flash memory. Under *hm_1* and *Websearch1*, adjusting the hot/cold region cannot obviously impact the performance of DAC because DAC strengthens the system performance under high-write-ratio workloads. The experimental results undoubtedly show that DAC has an edge over DAC-C and DAC-H thanks to DAC's dynamic adjustment of cold/hot regions.

In addition, we studied the effect of increasing G_c and G_h on the cache hit rate. The ghost cache is composed of G_c and G_h regions. In DAC, C_c and H_c stipulate the number m of cache pages. The total number of pages in cache C_c , H_c , G_c , and G_h is no more than $2m$. When it comes to G_c and G_h in the ghost cache, increasing the G_c size causes the size of another one to drop and vice versa. The data of G_h

and G_c is the metadata of data page evict from H_c and C_c ; thus, the G_h and G_c sizes largely depend on the size of H_c and C_c . If a cache page is replaced from H_c , G_h increases by 1 and G_c decreases by 1, and the opposite is true. Whether to replace a cache page from H_c or C_c is orchestrated by the eviction and replace algorithm. For this reason, we design a comparison test of DAC with DAC-C and DAC-H. We make use of DAC_C and DAC_H to investigate G_c and G_h 's effect on write/read hit ratios.

As shown in Fig. 14, the results reveal that DAC has a marginal difference in write hit rate compared with DAC-C, but the read hit rate is immensely improved by 33.7%, 22.3%, 83.8%, and 0.1% under *hm_0*, *prxy_0*, *hm_1*, and *WebSearch1*, respectively. The write and read hit ratio improvements are 309.8% and 30.9% under *hm_0*; the two

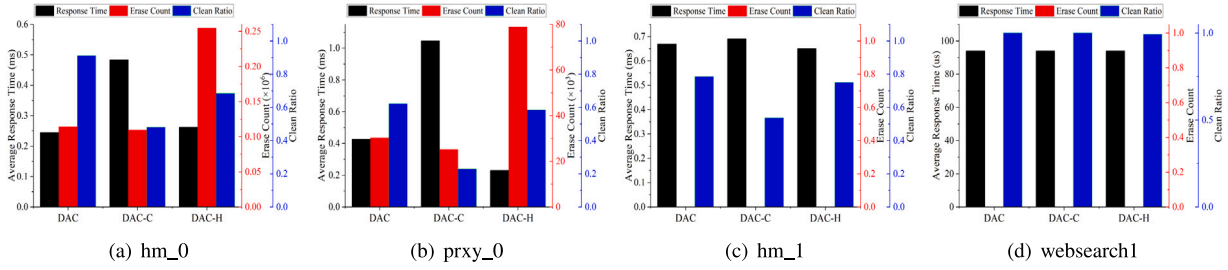


Fig. 13. The impact of eviction strategy on DAC performance. Superb performance is manifested by low average response time, low write amplification, and high clean ratio.

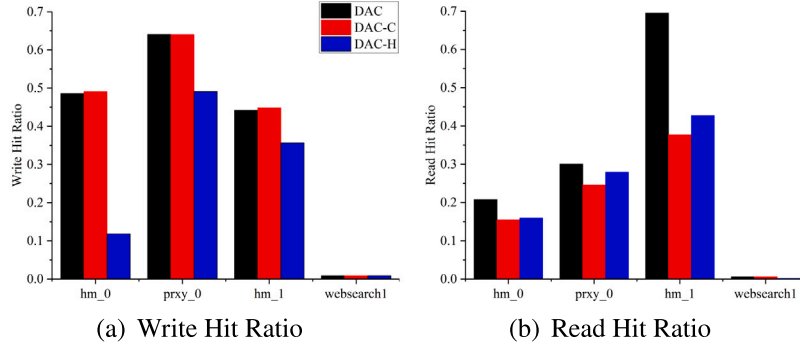


Fig. 14. The impact of eviction strategy on write hit ratio and read hit ratio.

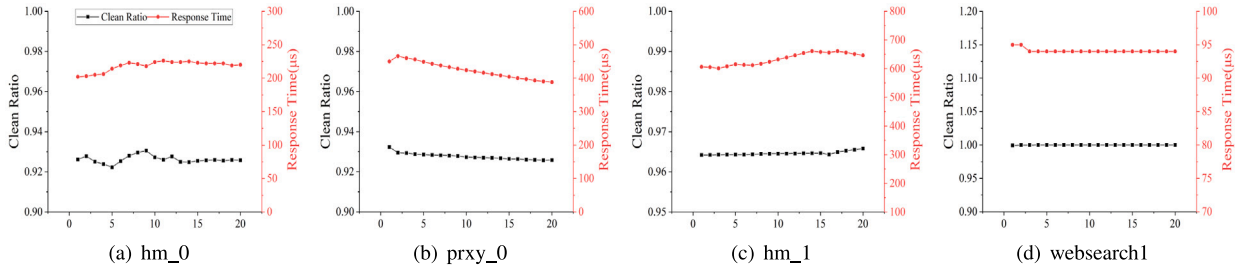


Fig. 15. The impact of clean ratio on response time.

hit ratios surge by 24.1% and 62.6% under hm_1, respectively. We conclude that significantly increasing the size of G_c and G_h affects the DAC overall performance.

Notably, the I/Os access pattern of the workload is changing, which can be perceived by DAC. When more cold data is accessed recently, the size of the cold region should be increased and vice versa for the hot region. We should apply different strategies for different I/O access patterns instead of replacing data from the cold region studied in our extended experiment. DAC-C outperforms DAC-H in terms of erase count but is weaker in terms of response time. The overall performance of both strategies is lower than DAC. This is because hot data is preferred to be replaced in DAC, which enlarges the write-back window. The increased active write-back data will be actively updated when the flash memory is idle, which will not increase the response time. However, frequent write-back of hot data significantly increases the erase count and impairs the SSD lifetime. In addition, DAC-H has a lower write-hit ratio than DAC-C and a higher read-hit. In a write-intensive scenario, hot data is more likely to be a write request, and DAC-H preferentially replaces hot data, which affects the write-hit ratio.

6.4.4. Impact of clean ratios on response time

In addition, we exploited the impact of the clean ratio on the average response time (see Fig. 15). In this test, we record both values of clean ratio and average response time while fixing the number of

requests in workloads. It must be noted that the clean ratio affects response time because clean data is directly deleted as the cache replacement occurs with a large clean ratio. This feature averts a high response time caused by an amount of data written into the flash memory. DAC governs DWW to dynamically adjust the write-back threshold according to I/O patterns under various workloads, thereby achieving a stable clean ratio (e.g., 0.92, 0.92, 0.96, and 0.99 under hm_0, prxy_0, hm_1, and WebSearch1, respectively). The clean ratio in DAC is stable during the running workload; thus, the response time of DAC varies within a small range. The experimental results unveil that DAC delivers a high clean ratio accompanied by a low response time.

6.4.5. Performance study under traces collected from Systor'17 traces

To study the performance of DAC in multiple application scenarios, we selected three virtual desktop workloads 2016030415-LUN2, 2016030819-LUN2, and 2016030917-LUN2 under Fujitsu Laboratories Japan [51] in the experiment. The tested metrics include response time, the standard deviation of response time, write amplification, clean ratio, write hit ratio, and read hit ratio.

DAC exhibits comprehensive performance advantages under all the workloads. DAC slashes the response time in most scenarios, with maximum improvements of 29.4%, 33.4%, and 24.4% under traces 2016030415-LUN2 (see Fig. 16(a)), 2016030819-LUN2 (see Fig. 17(a)), and 2016030917-LUN2 (see Fig. 18(a)), respectively. In addition, DAC optimizes response time and reduces the performance

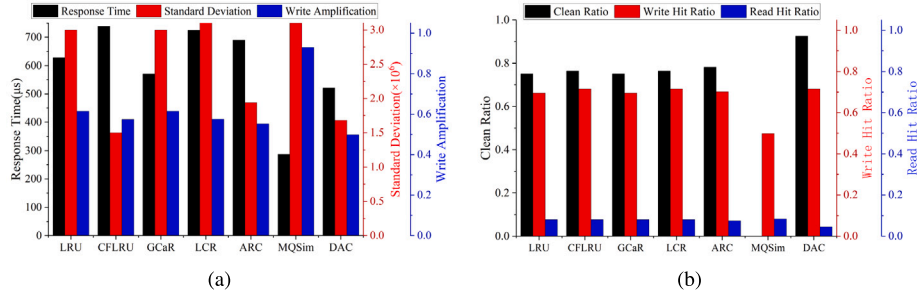


Fig. 16. Performance under 2016030415-LUN2 trace.

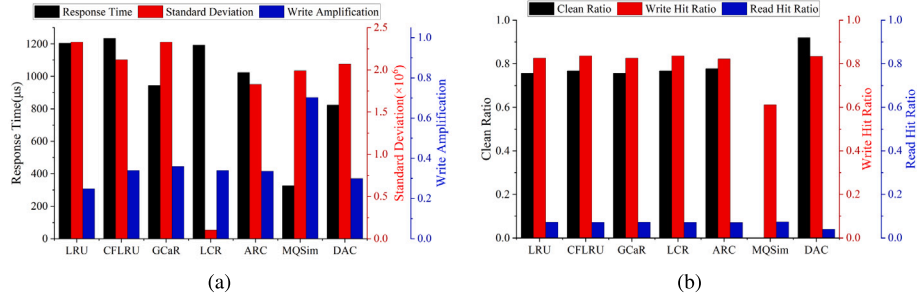


Fig. 17. Performance under 2016030819-LUN2 trace.

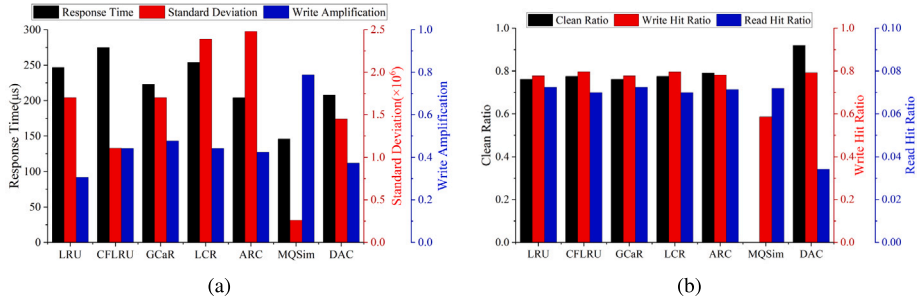


Fig. 18. Performance under 2016030917-LUN2 trace.

cliff under burst-access scenarios with maximum reductions of 46.2%, 11.6%, and 41.5%, respectively. MQSim minimizes response time at the expense of SSD lifetime, but MQSim fails to address the response time cliff in burst-access scenarios. MQSim increases response time by 84.2% compared with DAC. In addition, MQSim worsens write amplification by 87.8%, 141.3%, and 110.8% compared to DAC under traces 2016030415-LUN2, 2016030819-LUN2, and 2016030917-LUN2, thereby deteriorating SSD lifetime. In DAC, the active write-back converts dirty data to clean data, achieving the highest clean data ratio (see Fig. 17(b)) and the minimum response time of cache replacement. The DAC's adaptive adjustment of the write-back threshold according to the hot and cold changes of the workload improves the response time without significantly deteriorating the SSD lifetime. Therefore, the overall performance of the SSD results in a significant improvement.

Due to the high percentage of access-once data in VDI workloads, the hit ratios in most experiments are relatively similar. Compared to the other caching schemes, DAC enhances the write-hit ratio by an average of 8.6% (see Fig. 16(b)). When applied to MQSim, DAC demonstrates a substantial enhancement, increasing the write-hit ratio by 43.5% (see Fig. 18(b)). This is achieved by writing data that is accessed less than once back to flash memory and removing it from the cache. While DAC does not enhance the read-hit ratio compared to the other caching schemes, as it prioritizes replacing clean data, it still improves overall SSD performance in terms of response time due to its

active write-back operations. Although the data evicted by DAC may be requested again, the active write-back operations help to mitigate any negative impact on response time.

7. Discussion

In this section, we first discuss the impact of active write-back mechanism in DAC on the amount of write data in flash memory. Second, we present our considerations in the design of the DAC.

7.1. Active write-back mechanism in DAC

When a cache is full upon the arrival of a new request, a page will be replaced to flash memory. This procedure is referred to as the proactive write-back mechanism. The active write-back mechanism aims to actively write dirty data into the flash memory, lessening the small cache size for clean data and I/O conflicts in the cache.

The active write-back mechanism cannot increase the number of write-back operations. The active write-back mechanism untimely writes back data compared with the passive write-back manner. There is a high probability of sparking more proactive write-back operations in the absence of the active write-back mechanism because there will be few clean pages when new requests arrive under high-write ratio workloads. In this case, a large number of I/O request conflicts occur,

affecting the SSD performance in terms of the response time. The active write-back operations produce clean data amid idle time period. As I/O requests arrive, the number of clean pages increases. When a cache replacement occurs, clean data is preferentially evicted out from the cache to slash the number of write-back operations and to alleviate I/O requests conflicts. As a result, response time is shortened.

In summary, the active write-back mechanism will by no means to stimulate the number of write-back operations; rather, this mechanism decreases the number of write-back operations thanks to an increasing number of clean pages. As shown in Figs. 4 and 5, our experimental results demonstrate that DAC has an impressive edge over the other caching algorithms in terms of the amount of data written into flash memory and the response time under the workload with burst I/O requests. In addition, DAC averts the additional write operations caused by the active update mechanism in DAC.

7.2. Further considerations in DAC

Our proposed DAC aims to improve the cache performance inside an SSD, which is dependent of the read–write imbalance cost and load in flash memory chip.

Owing to the read–write imbalance, the write latency is higher than that of read; the cache replacement of dirty data is greater than that of clean data. An SSD is conducive to evicting clean data, but clean data rarely exist in the cache under burst-I/Os and write-intensive workloads. Upon an idle chip, cold and dirty data is updated back into flash memory; meanwhile, the cold and dirty data is converted to clean data. This strategy is slated to alleviate write conflicts and high access latency under burst-I/O workloads.

The separation of hot and cold data is one of salient traits in the design of cache management schemes. The replacement cost of hot data is greater than that of cold data. The hot and cold separation module in DAC prioritizes to write cold data back, thereby curtailing the probability of the data being re-accessed after updating back into flash memory. This mechanism is adroit at cutting back the replacement cost in turn.

In addition, two conditions that spark the active write-back operations includes the chip idle and the chip queue empty statuses. The two conditions depend on load in flash memory chips. The active write-back operation, which is invoked upon idle flash, considers the load state in flash memory chip while revamping chip resources utilization.

8. Conclusion

In this work, we proposed a dynamically active and collaborative cache management scheme (DAC) for SSDs from the perspective of multiple-level systems. DAC overcomes the shortcomings of existing caching schemes that confront high access conflicts and low clean-data replacement ratios. DAC divides cache space into five regions with different attributes, namely, C_n , C_a , H_c , G_C , and G_h . The ghost cache registers cache entries' historical information, which enhances space utilization and retrieval performance in the cache. The scan & active write-back module actively writes data in C_n back into flash memory when the cache is sitting idle, producing sufficient amount of clean cache data. DAC adjusts the active-write-back window in the cache according to access patterns in workloads, flash chip state, and historical information in the ghost cache. DAC creates on-demand clean data to prolong SSD lifetime. In the eviction & replacement module, the dynamic decision on the replaced cache data boosts cache hit ratios through historical information.

We undertake extensive experiments to validate the benefits of DAC compared against the seven existing caching schemes including LRU, CFLRU, GCaR-CFLRU, LCR, ARC, AD-LRU, and MQSim. The results unveil that DAC shortens the response time by up to 60.8% with an average of 24.4%. When it comes to response-time cliff, DAC offers the maximum improvement by approximately 68.71% with an average optimization rate of 26.9%. To extend SSD lifespan, DAC cuts back the number of erase count by up to 93.3% with an average of 17.04%.

Declaration of competing interest

These authors declare that there is no conflict of interests regarding the publication of this article.

Data availability

Data will be made available on request.

References

- [1] G2M research releases fall 2018 NVM express market sizing report, 2018.
- [2] A. Chen, A review of emerging non-volatile memory (NVM) technologies and applications, *Solid-State Electron.* 125 (2016) 25–38.
- [3] G.W. Burr, M.J. Brightsky, A. Sebastian, H.-Y. Cheng, J.-Y. Wu, S. Kim, N.E. Sosa, N. Papandreou, H.-L. Lung, H. Pozidis, et al., Recent progress in phase-change memory technology, *IEEE J. Emerg. Sel. Top. Circuits Syst.* 6 (2) (2016) 146–162.
- [4] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, O. Mutlu, Evaluating STT-ram as an energy-efficient main memory alternative, in: 2013 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, IEEE, 2013, pp. 256–267.
- [5] C. Gao, L. Shi, C. Ji, Y. Di, K. Wu, C.J. Xue, E.H. Sha, Exploiting parallelism for access conflict minimization in flash-based solid state drives, *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* 37 (1) (2018) 168–181.
- [6] J. Wan, W. Wu, L. Zhan, Q. Yang, X. Qu, C. Xie, DEFT-cache: A cost-effective and highly reliable SSD cache for RAID storage, in: 2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2017, pp. 102–111.
- [7] A. Jain, C. Lin, Back to the future: leveraging Belady's algorithm for improved cache replacement, in: 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture, ISCA, IEEE, 2016, pp. 78–89.
- [8] A. Dan, D. Towsley, An approximate analysis of the LRU and FIFO buffer replacement schemes, in: Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, 1990, pp. 143–152.
- [9] E.J. O'neil, P.E. O'neil, G. Weikum, The LRU-K page replacement algorithm for database disk buffering, *Acm Sigmod Rec.* 22 (2) (1993) 297–306.
- [10] T. Johnson, D. Shasha, et al., 2Q: a low overhead high performance buffer management replacement algorithm, in: Proceedings of the 20th International Conference on Very Large Data Bases, Citeseer, 1994, pp. 439–450.
- [11] S. Jiang, X. Zhang, LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance, *ACM SIGMETRICS Perform. Eval. Rev.* 30 (1) (2002) 31–42.
- [12] N. Megiddo, D.S. Modha, ARC: A self-tuning, low overhead replacement cache, in: *FAST*, Vol. 3, No. 2003, 2003, pp. 115–130.
- [13] S.-y. Park, D. Jung, J.-u. Kang, J.-s. Kim, J. Lee, CFLRU: a replacement algorithm for flash memory, in: Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, 2006, pp. 234–241.
- [14] Y. Ou, T. Härder, P. Jin, CFDC: a flash-aware replacement policy for database buffer management, in: Proceedings of the Fifth International Workshop on Data Management on New Hardware, 2009, pp. 15–20.
- [15] Z. Li, P. Jin, X. Su, K. Cui, L. Yue, CCF-LRU: a new buffer replacement algorithm for flash memory, *IEEE Trans. Consum. Electron.* 55 (3) (2009) 1351–1359.
- [16] P. Jin, Y. Ou, T. Härder, Z. Li, AD-LRU: An efficient buffer replacement algorithm for flash-based databases, *Data Knowl. Eng.* 72 (2012) 83–102.
- [17] S. Wu, Y. Lin, B. Mao, H. Jiang, GCaR: Garbage collection aware cache management with improved performance for flash-based SSDs, in: Proceedings of the 2016 International Conference on Supercomputing, 2016, pp. 1–12.
- [18] C. Liu, M. Lv, Y. Pan, H. Chen, Y. Li, C. Li, Y. Xu, LCR: load-aware cache replacement algorithm for flash-based SSDs, in: 2018 IEEE International Conference on Networking, Architecture and Storage, NAS, IEEE, 2018, pp. 1–10.
- [19] H. Chen, Y. Pan, C. Li, Y. Xu, ECR: Eviction-cost-aware cache management policy for page-level flash-based SSDs, *Concurr. Comput.: Pract. Exper.* (2019) e5395.
- [20] H. Chen, C. Li, Y. Pan, M. Lyu, Y. Li, Y. Xu, HCFTL: A locality-aware page-level flash translation layer, in: 2019 Design, Automation & Test in Europe Conference & Exhibition, DATE, IEEE, 2019, pp. 590–593.
- [21] A. Gupta, Y. Kim, B. Urganekar, DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings, *Acm Sigplan Notices* 44 (3) (2009) 229–240.
- [22] S. Yan, H. Li, M. Hao, M.H. Tong, S. Sundararaman, A.A. Chien, H.S. Gunawi, Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs, *ACM Trans. Storage* 13 (3) (2017) <http://dx.doi.org/10.1145/3121133>.
- [23] H. Sun, X. Cheng, C. Zhang, Y. Yue, X. Qin, HIPA: A hybrid load balancing method in SSDs for improved parallelism performance, *J. Syst. Archit.* 131 (2022) 102705.
- [24] S. Tripathy, M. Satpathy, SSD internal cache management policies: A survey, *J. Syst. Archit.* 122 (2022) 102334.

- [25] N. Beckmann, H. Chen, A. Cidon, {LHD}: Improving cache hit rate by maximizing hit density, in: 15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18), 2018, pp. 389–403.
- [26] S. Bai, X. Bai, X. Che, Window-LRFU: a cache replacement policy subsumes the LRU and window-LFU policies, *Concurr. Comput.: Pract. Exper.* 28 (9) (2016) 2670–2684.
- [27] Z. Fan, D.H. Du, D. Voigt, H-ARC: A non-volatile memory based cache policy for solid state drives, in: 2014 30th Symposium on Mass Storage Systems and Technologies, MSST, IEEE, 2014, pp. 1–11.
- [28] M. Cho, D. Kang, ML-CLOCK: Efficient page cache algorithm based on perceptron-based neural network, *Electronics* 10 (20) (2021) 2503.
- [29] S. Ebrahimi, R. Salkhordeh, S.A. Osia, A. Taheri, H.R. Rabiee, H. Asadi, Rcrnn: Reconfigurable cache architecture for storage systems using recurrent neural networks, *IEEE Trans. Emerg. Top. Comput.* 10 (3) (2021) 1492–1506.
- [30] P. Li, A hotness-aware write buffer management scheme for the lifetime extension of flash-based solid state drives, in: Proceedings of the International MultiConference of Engineers and Computer Scientists, IMECS 2021, 2021, pp. 1–6.
- [31] G. Yadgar, M. Gabel, S. Jaffer, B. Schroeder, SSD-based workload characteristics and their performance implications, *ACM Trans. Storage (TOS)* 17 (1) (2021) 1–26.
- [32] H. Jung, H. Shim, S. Park, S. Kang, J. Cha, LRU-WSR: integration of LRU and writes sequence reordering for flash memory, *IEEE Trans. Consum. Electron.* 54 (3) (2008) 1215–1223.
- [33] Z. Sha, Z. Cai, F. Trahay, J. Liao, D. Yin, Unifying temporal and spatial locality for cache management inside SSDs, in: 2022 Design, Automation & Test in Europe Conference & Exhibition, DATE, IEEE, 2022, pp. 891–896.
- [34] Q. Chen, S. Wang, Y. Zhou, F. Wu, S. Li, Z. Wang, C. Xie, PACA: A page type aware read cache scheme in QLC flash-based SSDs, in: 2022 IEEE 40th International Conference on Computer Design, ICCD, IEEE, 2022, pp. 59–66.
- [35] Z. Xiangwei, D. Yuhui, Constructing cache algorithm for flash by leveraging page reconstruction and data temperature recognition, *J. Front. Comput. Sci. Technol.* 15 (1) (2021) 84.
- [36] Y. Pan, M. Lin, Z. Wu, H. Zhang, Z. Xu, Caching-aware garbage collection to improve performance and lifetime for NAND flash SSDs, *IEEE Trans. Consum. Electron.* 67 (2) (2021) 141–148.
- [37] L.V. Rodriguez, F.B. Yusuf, S. Lyons, E. Paz, R. Rangaswami, J. Liu, M. Zhao, G. Narasimhan, Learning cache replacement with CACHEUS, in: FAST, 2021, pp. 341–354.
- [38] H. Lin, J. Li, Z. Sha, Z. Cai, J. Liao, Y. Shi, A novel CFLRU-based cache management approach for NAND-based SSDs, in: IFIP International Conference on Network and Parallel Computing, Springer, 2021, pp. 214–225.
- [39] C. Gao, L. Shi, Y. Di, Q. Li, C.J. Xue, K. Wu, E. Sha, Exploiting chip idleness for minimizing garbage collection—Induced chip access conflict on SSDs, *ACM Trans. Des. Autom. Electron. Syst.* 23 (2) (2017) 1–29.
- [40] J. Guo, Y. Hu, B. Mao, S. Wu, Parallelism and garbage collection aware I/O scheduler with improved SSD performance, in: 2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS, IEEE, 2017, pp. 1184–1193.
- [41] C. Gao, L. Shi, C.J. Xue, C. Ji, J. Yang, Y. Zhang, Parallel all the time: Plane level parallelism exploration for high performance SSDs, in: 2019 35th Symposium on Mass Storage Systems and Technologies, MSST, IEEE, 2019, pp. 172–184.
- [42] M. Huang, O. Serres, V.K. Narayana, T. El-Ghazawi, G. Newby, Efficient cache design for solid-state drives, in: Proceedings of the 7th ACM International Conference on Computing Frontiers, 2010, pp. 41–50.
- [43] S. Wu, B. Mao, Y. Lin, H. Jiang, Improving performance for flash-based storage systems through GC-aware cache management, *IEEE Trans. Parallel Distrib. Syst.* 28 (10) (2017) 2852–2865.
- [44] H. Sun, S. Dai, J. Huang, X. Qin, Co-active: A workload-aware collaborative cache management scheme for NVMe SSDs, *IEEE Trans. Parallel Distrib. Syst.* 32 (06) (2021) 1437–1451.
- [45] H. Lin, J. Li, Z. Sha, Z. Cai, Y. Shi, B. Geroft, J. Liao, Adaptive management with request granularity for DRAM cache inside NAND-based SSDs, *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* (2022).
- [46] H. Lin, Z. Sha, J. Li, Z. Cai, B. Geroft, Y. Shi, J. Liao, DRAM cache management with request granularity for NAND-based SSDs, in: Proceedings of the 51st International Conference on Parallel Processing, 2022, pp. 1–10.
- [47] X. Ruan, X. Jiang, H. Chen, Cached mapping table prefetching for random reads in solid-state drives, in: 2021 IEEE International Conference on Networking, Architecture and Storage, NAS, IEEE, 2021, pp. 1–6.
- [48] R. Liu, Z. Tan, L. Long, Y. Wu, Y. Tan, D. Liu, Improving fairness for SSD devices through DRAM over-provisioning cache management, *IEEE Trans. Parallel Distrib. Syst.* 33 (10) (2022) 2444–2454.
- [49] B. Zhou, C. Ding, Y. Lv, C.J. Xue, Q. Zhuge, E.H.-M. Sha, L. Shi, Sac: a stream aware write cache scheme for multi-streamed solid state drives, in: Proceedings of the 26th Asia and South Pacific Design Automation Conference, 2021, pp. 645–650.
- [50] J. Bhimani, N. Mi, Z. Yang, J. Yang, R. Pandurangan, C. Choi, V. Balakrishnan, FIOS: Feature based I/O stream identification for improving endurance of multi-stream SSDs, in: 2018 IEEE 11th International Conference on Cloud Computing, CLOUD, IEEE, 2018, pp. 17–24.
- [51] C. Lee, T. Kumano, T. Matsuki, H. Endo, N. Fukumoto, M. Sugawara, Understanding storage traffic characteristics on enterprise virtual desktop infrastructure, in: Proceedings of the 10th ACM International Systems and Storage Conference, 2017, pp. 1–11.
- [52] J.-W. Hsieh, T.-W. Kuo, L.-P. Chang, Efficient identification of hot data for flash memory storage systems, *ACM Trans. Storage (TOS)* 2 (1) (2006) 22–40.
- [53] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, S. Zhang, Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity, in: Proceedings of the International Conference on Supercomputing, 2011, pp. 96–107.
- [54] A. Tavakkol, P. Mehrvarzy, M. Arjomand, H. Sarbazi-Azad, Performance evaluation of dynamic page allocation strategies in SSDs, *ACM Trans. Model. Perform. Eval. Comput. Syst. (TOMPECS)* 1 (2) (2016) 1–33.
- [55] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, S. Ghose, O. Mutlu, MQSim GitHub repository, 2018, <https://github.com/CMU-SAFARI/MQSim>.
- [56] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, S. Ghose, O. Mutlu, Mqsim: A framework for enabling realistic studies of modern multi-queue SSD devices, in: 16th USENIX Conference on File and Storage Technologies, FAST 18, 2018, pp. 49–66.
- [57] Micron, NAND flash memory MLC MT29F256G08CKCAB datasheet, 2014, <https://datasheetspdf.com/datasheet/MT29F256G08CKCAB.html>.
- [58] W. Bux, I. Iliadis, Performance of greedy garbage collection in flash-based solid-state drives, *Perform. Eval.* 67 (11) (2010) 1172–1186.
- [59] M. Jung, M.T. Kandemir, Sprinkler: Maximizing resource utilization in many-chip solid state disks, in: 2014 IEEE 20th International Symposium on High Performance Computer Architecture, HPCA, 2014, pp. 524–535.
- [60] UMass Trace Repository, 2002. <http://traces.cs.umass.edu/index.php/Storage/Storage>.
- [61] SNIA, MSR cambridge traces, 2008, <http://iotta.snia.org/traces/388>.
- [62] D. Narayanan, A. Donnelly, A. Rowstron, Write OFF-loading: Practical power management for enterprise storage, in: Proceedings of the 6th USENIX Conference on File and Storage Technologies, 2008, pp. 253–267.
- [63] N. Megiddo, D.S. Modha, Outperforming LRU with an adaptive replacement cache algorithm, *Computer* 37 (4) (2004) 58–65.