Contents lists available at ScienceDirect

Integration, the VLSI Journal

journal homepage: www.elsevier.com/locate/vlsi

CDA-GC: An effective cache data allocation for garbage collection in flash-based solid-state drives

Keyu Wang [a], Huailiang Tan [a,*], Zaihong He [a], Jinyou Li [a], Keqin Li [b]

[a] Hunan University, College of Computer Science and Electronic Engineering, Changsha, 410082, Hunan, China
[b] Department of Computer Science, State University of New York, New Paltz, NY, 12561, United States of America

ARTICLE INFO

Keywords:
Cache management
Flash translation layer
Flash storage
Garbage collection
Solid state drive

ABSTRACT

In the research of solid-state drive (SSD) performance enhancement, constructing an efficient garbage collection (GC) mechanism is crucial for accelerating device operations and extending their service life, especially in large data processing applications like databases and file systems. Therefore, this paper conducts an in-depth study on the impact of cache management strategies on GC performance and proposes an innovative GC algorithm called Cache Data Allocation GC (CDA-GC). By optimizing data allocation and management within the cache, this algorithm reduces unnecessary data migration during the GC process, thereby improving data processing efficiency and reducing the impact of GC operations on device performance. The core of CDA-GC lies in its innovative cache data management strategy, which can significantly reduce the data migration demands during the GC process. This method not only improves the overall processing performance of SSDs but also reduces the adverse impact of GC activities on device performance by optimizing data access patterns. We implemented and validated the algorithm on the Cosmos+ OpenSSD platform and compared it with existing advanced SSD caching strategies in real-world scenarios. Experimental results show that in database and file system applications, the CDA-GC algorithm can effectively improve performance.

1. Introduction

NAND flash memory has attracted significant attention from academia and industry due to its advantages such as compact size, no mechanical noise, impact resistance, light weight, and low power consumption [1,2]. With the advancement of semiconductor technology, flash-based solid-state drives (SSDs) have gradually replaced traditional hard disk drives (HDDs) as mainstream storage devices. Flash-based SSDs are not only widely used in consumer electronics, such as desktops and laptops, due to their high performance and reliability, but also play a critical role in high-performance computing (HPC) and enterprise data centers [3–5].

A flash-based SSD consists of multiple flash memory chips, each comprising numerous blocks and pages. The erase-before-write mechanism of flash memory makes erase operations significantly slower compared to read/write operations. To mitigate this impact, SSDs adopt a remote update strategy [6]. Additionally, a GC mechanism is introduced to reclaim invalid pages generated during flash memory update operations by releasing block space [7]. The goal of GC is to consolidate valid data and reclaim blocks occupied by invalid data, thereby making them available for future writes.

However, the garbage collection process often involves moving valid data to new locations before erasing blocks, and this data migration process is a key factor in GC-induced performance degradation. During GC, valid data must be read from blocks targeted for erasure and rewritten to other blocks, which results in additional read and write operations that consume valuable I/O bandwidth. This data movement not only increases data access latency but also competes with normal I/O operations for system resources, which can lead to severe performance bottlenecks. The impact of GC is especially pronounced in high-load environments, such as databases and file systems, where large amounts of data migration lead to increased latency and reduced system throughput. Therefore, performance degradation due to GC is a primary concern affecting SSD efficiency [8,9].

A major challenge introduced by garbage collection is the significant performance overhead associated with data migration during the GC process. During GC, valid data must be relocated from blocks marked for erasure to new blocks, requiring frequent read and write operations that further impact SSD performance. The frequency of GC operations and the amount of data migration depend largely on how data is managed internally—specifically, how data is organized in the cache and subsequently written to flash blocks. Inefficient data management

* Corresponding author.
E-mail address: tanhuailiang@hnu.edu.cn (H. Tan).

https://doi.org/10.1016/j.vlsi.2025.102359
Received 4 July 2024; Received in revised form 25 November 2024; Accepted 12 January 2025
Available online 22 January 2025

during this process can lead to excessive GC operations, resulting in increased I/O overhead, higher latency, and reduced overall system performance.

Integrating DRAM as a buffer cache within SSDs effectively mitigates long access delays, significantly enhancing user I/O performance [10]. Nonetheless, Garbage Collection remains a critical component of SSD maintenance, directly impacting overall system performance. Existing research on GC has primarily focused on optimizing its implementation within various Flash Translation Layer (FTL) schemes. This includes strategies such as victim block selection [11,12], leveraging workload characteristics and internal parallelism [13–15], and refining the FTL layer for improved efficiency [16–18]. However, these studies often overlook the substantial impact that internal data distribution within SSDs has on GC efficiency.

The distribution of data in the cache and the process of writing it to flash memory are particularly critical to GC efficiency, especially in high-load or data-intensive scenarios. The use of cache, such as DRAM, helps buffer write operations and accelerates data retrieval. However, traditional cache replacement strategies, like LRU (Least Recently Used), are typically designed for general I/O workloads and do not consider the specific demands imposed by garbage collection. During GC, the cache often becomes polluted with data that needs to be temporarily moved, which can displace other valuable cached data and reduce cache hit rates. This ineffective utilization of cache resources increases the likelihood of cache misses and adds further overhead to the GC process. Such cache inefficiencies during GC contribute to increased I/O overhead, prolonging access times, and ultimately degrading system performance.

Additionally, traditional GC optimization methods tend to focus on block-level management strategies, such as victim block selection and maximizing internal parallelism, while overlooking the importance of optimizing data flow from cache to flash during GC. This oversight can lead to redundant cache operations and unnecessary data movement, which further exacerbates the performance issues faced by SSDs, especially under high I/O load conditions. For example, without careful cache management, the frequent eviction and reloading of data during GC operations result in suboptimal performance and increased latency. Therefore, there is a pressing need for strategies that not only optimize GC at the block level but also manage the cache in a manner that reduces the cost associated with frequent data migration during garbage collection.

To address the performance challenges caused by GC in SSDs, this paper proposes a new GC optimization strategy called Cache Data Allocation Garbage Collection (CDA-GC). CDA-GC focuses on the distribution of cache data and its writing process to flash memory. By effectively managing the distribution of data in the cache and optimizing the dispatch strategy, CDA-GC reduces unnecessary data migration, enhances processing efficiency, and mitigates the negative impact of GC on SSD performance. CDA-GC consists of two main components: the Cache GC Page Stager and the Cold and Hot Data Dispatcher. The Cache GC Page Stager temporarily holds GC-related pages in the cache to enhance scheduling efficiency and reduce unnecessary read and write operations. The Cold and Hot Data Dispatcher segregates data based on its temperature, optimizing cache dispatch, reducing data migration within flash blocks, and ultimately improving GC efficiency. We evaluated CDA-GC on the Cosmos+ FPGA OpenSSD platform and compared it with existing advanced cache management strategies. The results demonstrate that CDA-GC significantly enhances GC performance, offering a new perspective that breaks through the limitations of existing GC optimization methods and provides significant improvements in SSD efficiency.

The main contributions of this paper are:

(1) Exploration of the impact of cache data allocation on GC efficiency in SSDs, highlighting the importance of optimized data management strategies in enhancing system performance;

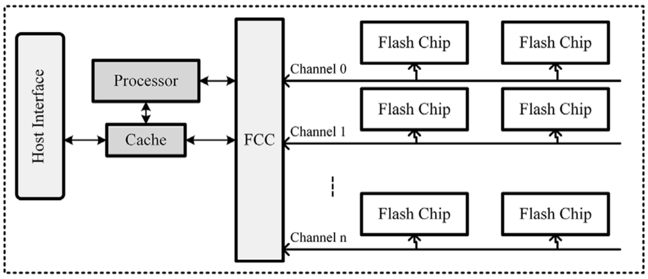


Fig. 1. A typical architecture diagram of an SSD.

(2) Proposal of the novel CDA-GC data management strategy, designed to optimize GC by improving the allocation of cache data, minimizing unnecessary data migration, and enhancing processing efficiency;

(3) Evaluation of the CDA-GC strategy on the Cosmos+ FPGA OpenSSD platform, demonstrating its effectiveness in enhancing GC performance compared to existing strategies, serving as a foundation for future SSD performance optimization research.

The remainder of this paper is organized as follows: Section 2 provides background and motivation for this study. Section 3 details the CDA-GC scheme. Section 4 discusses performance evaluation results. Section 5 compares related work, and Section 6 concludes the paper with directions for future research

2. Background and motivation

In this section, we first introduce the key features of SSDs related to this research. Then, we discuss the motivation of the CDA-GC scheme.

2.1. SSD architecture

Unlike traditional mechanical HHDs, flash-based SSDs are composed of semiconductor materials and contain no mechanical moving parts. This design eliminates the addressing overhead associated with HDD head movement. Additionally, SSDs exploit various forms of internal parallelism within flash memory, resulting in superior random read and write performance. Fig. 1 illustrates a typical SSD architecture, comprising components such as the host interface, microprocessor, onboard RAM, flash memory interface circuit, and flash memory chips.

The host interface connects the SSD to the host system at both logical and physical levels. While traditional SSDs use the Serial ATA (SATA) interface, its throughput limitations have led to the adoption of the Non-Volatile Memory Express (NVMe) protocol. NVMe has attracted significant attention from storage designers due to its high bandwidth and multi-queue technology. Interfaces based on this protocol offer bandwidths of dozens of gigabits per second, sufficiently meeting the parallelism requirements of SSD internal components. Consequently, PCIe interfaces based on the NVMe protocol are widely employed in high-performance SSDs.

The microprocessor acts as the execution unit for SSD firmware, which includes host interface logic, the FTL, and operations related to storage media management. The FTL is crucial in the storage system, performing operations such as address mapping, garbage collection, wear leveling, and bad block management. These microprocessors typically utilize low-power embedded CPUs, which constrains the firmware from implementing highly complex algorithms.

Onboard RAM serves a dual purpose: it forms an embedded system with the microprocessor to execute SSD firmware operations and acts as a data buffer cache (referred to as cache in this paper). The data cache bridges the performance gap between the host interface and the flash memory device, facilitating smoother data transfer processes. Particularly for write operations, it effectively conceals the write latency of

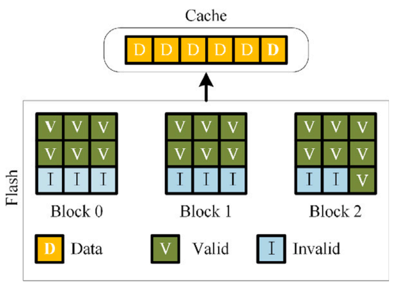


Fig. 2. Traditional GC Data Distribution.

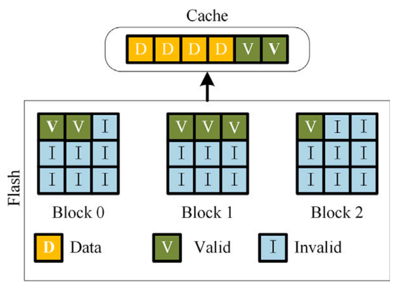


Fig. 3. CDA-GC Data Distribution.

flash memory. Similarly, during garbage collection, the cache provides temporary storage for valid data until it is relocated.

A flash chip is the fundamental data storage unit, with read and write operations performed on page units and erase operations on block units. Currently, there are four mainstream types of NAND flash memory cells: Single-Level Cell (SLC), Multi-Level Cell (MLC), Triple-Level Cell (TLC), and Quad-Level Cell (QLC). SLC stores one bit per cell, MLC stores two bits, TLC stores three bits, and QLC stores four bits per cell.

2.2. Motivation

With the growing advantages of solid-state drives in terms of performance, reliability, and durability, SSDs have become the main solution for modern storage systems. However, one of the key challenges faced by SSDs is the garbage collection mechanism, which is essential for maintaining the long-term efficiency and extending the lifespan of the device. Flash memory, as the core storage medium of SSDs, uses a "write-before-erase" mechanism. When data is updated, new data is written to a new location, and the old data becomes invalid. Over time, these invalid data accumulate, and while no longer in use, they occupy storage space and degrade performance. The goal of garbage collection is to reclaim this space by erasing blocks containing invalid data and consolidating valid data into other blocks to free up space.

However, the garbage collection process often comes with significant performance overhead, especially in high-load environments such as database and file system applications. GC requires migrating valid data to new blocks, and this data migration operation involves frequent read and write operations, occupying substantial I/O bandwidth, increasing data access latency, and impacting normal I/O operations. Particularly when GC involves large amounts of data, excessive data migration and repeated writing exacerbate performance bottlenecks. Therefore, reducing unnecessary data migration during GC and optimizing the garbage collection operation is a key issue for improving SSD performance.

Traditional GC optimization methods often focus on block selection strategies or memory management layer optimization. However, they rarely consider the distribution of data within the cache, especially the overhead associated with data migration during the process of reading data from flash to cache, which is often overlooked. Traditional solutions tend to concentrate on block-level memory management while ignoring the cost of reading data from flash into the cache during GC, leading to unnecessary performance degradation. This situation is depicted in Fig. 2, which illustrates the state of data distribution during the traditional GC process. As shown in Fig. 2, data migration between flash blocks and the cache is performed without considering the associated overhead. The cache is filled with data from different blocks, resulting in increased I/O load and additional system overhead. In the figure, a large number of valid and invalid pages are present within the blocks, demonstrating the inefficiency of traditional GC methods, leading to frequent data movement that significantly impacts overall system performance.

To address this issue, this paper proposes the CDA-GC algorithm. The core idea of CDA-GC is to reduce unnecessary data migration during GC by optimizing cache management and data distribution. Specifically, CDA-GC stores GC-related data in the cache in advance, thereby effectively reducing the frequent migration of hot and cold data from flash to cache during GC, improving cache utilization, lowering I/O bandwidth consumption, and reducing data access latency for each GC operation. This strategy is effectively illustrated in Fig. 3, which represents the state of data distribution in the CDA-GC approach. As shown in Fig. 3, the cache already contains GC-related pages, and the distribution of valid and invalid data within the flash blocks is more efficient, resulting in fewer valid pages remaining within the blocks. This optimized layout ensures that data migration during GC has as little impact as possible on normal I/O operations. By reducing the overhead associated with cache data migration, CDA-GC is able to enhance overall system performance, improve cache efficiency, and mitigate the adverse effects of GC on system I/O flows.

At the flash memory level, CDA-GC applies a hot/cold data partitioning strategy, ensuring that hot and cold data are written to different flash blocks or regions, preventing them from being mixed during GC. This separation strategy further reduces the amount of data migration from flash to cache during GC, reducing the impact of GC on normal I/O operations, minimizing unnecessary erase operations, reducing write amplification, and extending the lifespan of the SSD.

By optimizing the data path from flash to cache during GC, CDA-GC effectively reduces the data migration burden during GC, improves GC efficiency, and minimizes the impact on normal I/O operations, ensuring that the system can maintain high performance while performing garbage collection. This approach provides a novel method for SSD performance optimization, particularly in high-load applications like databases and file systems, where it can significantly enhance SSD response speed and stability.

3. Architecture of CDA-GC

In the previous section, we explored the significance of the garbage collection issue in SSDs and reviewed existing GC strategies. These strategies do not fully leverage DRAM to address the impact of data distribution on GC efficiency. In this section, we introduce a new approach named CDA-GC, which thoroughly considers the use of DRAM to manage the impact of data distribution on GC. The design principles and framework of CDA-GC will be presented herein.

3.1. Design principles

In designing the CDA-GC scheme, our objective is to comprehensively enhance the performance and efficiency of SSDs in the garbage collection process. The core design principles focus on optimizing data read/write speeds, reducing operational latency, and enhancing overall system processing capability through meticulous data management and the application of efficient algorithms in the GC process. CDA-GC

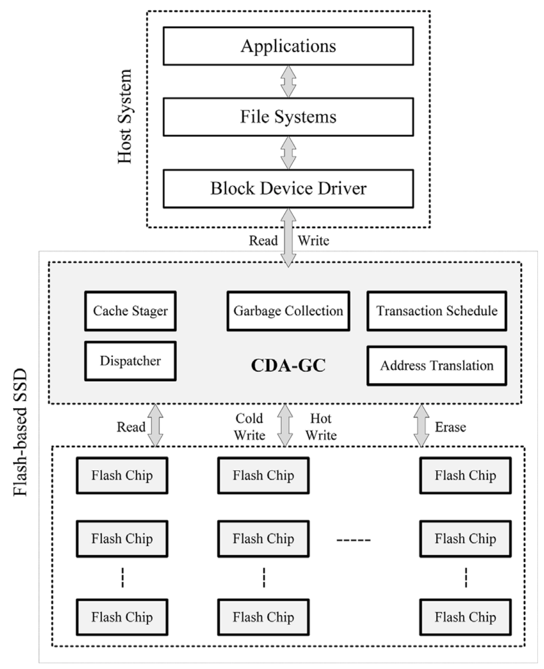


Fig. 4. Overview of CDA-GC.

endeavors to minimize unnecessary data migration and rewrite operations during GC by analyzing data distribution, thereby improving data processing speed and system responsiveness. Simultaneously, CDA-GC places importance on lightweight system design for practical application. Algorithm implementation emphasizes reducing reliance on computational resources and memory, ensuring that optimization measures do not impose excessive burdens on system resources. A lightweight design approach aims to mitigate potential overconsumption issues in real SSD environments. These design principles not only reflect our comprehensive considerations for improving SSD performance and efficiency but also underscore the importance of algorithm practicality and adaptability. Through such design considerations, CDA-GC aims to provide a highly efficient and flexible garbage collection optimization solution for SSDs, capable of demonstrating its performance advantages across various application environments.

3.2. CDA-GC architecture and abstractions

Fig. 4 illustrates the comprehensive system architecture of our proposed CDA-GC. Positioned within the FTL of flash-based SSDs, CDA-GC interfaces with the host system, which comprises three layers: application, file system, and block device. Through the PCIe interface, the SSD device connects to the host system. The FTL facilitates communication between host-side data and flash-side data. During read and write operations, the onboard cache serves as an intermediary, facilitating data transfer between the flash memory and the host system.

When the host performs read and write operations, the system first searches for the relevant data in the cache. If the data is in the cache, the system directly performs read and write operations. If the data is not in the cache, the SSD triggers cache replacement. During the cache replacement process, CDA-GC retains a certain amount of data related to GC and prioritizes the replacement of data unrelated to GC. Additionally, CDA-GC divides the data requests in the cache into hot data and cold data, and stores the divided data in isolation blocks when writing back to the flash memory. Unlike the traditional LRU strategy, CDA-GC maintains a certain proportion of data related to GC in the cache. When the SSD executes GC, CDA-GC effectively reduces the movement overhead of GC-related data in the cache. On the other hand, the traditional FTL strategy directly writes data to the flash memory without segregating hot and cold data. To reduce unnecessary data migration in erase blocks during GC, CDA-GC segregates hot and cold data in the cache and sends them to the flash memory separately. Compared to other advanced cache management and GC strategies, CDA-GC fully utilizes the cache to consider the impact of data distribution on GC.

The structure of CDA-GC comprises two components: Cache GC Page Stager and Cold and Hot Data Dispatcher. In the following two sections, we will elaborate and explain the contents of Cache Stager and Dispatcher in detail.

3.2.1. Cache GC page stager

In modern SSDs, the effectiveness of cache management is crucial for enhancing overall system performance. The SSD cache, serving as an important intermediary layer between the CPU and flash memory, can significantly reduce system read and write latency and improve I/O efficiency through efficient data allocation and management. However, one key challenge in cache management is how to effectively allocate and retain data during garbage collection. During the GC process, SSDs need to frequently move valid data to free up storage blocks, which leads to frequent changes in cache data and poses a significant burden on traditional cache management strategies. Currently, the most commonly used cache replacement strategy is the LRU strategy, which can effectively boost system read and write efficiency under normal, non-GC conditions. However, under GC conditions, due to significant changes in data access patterns and priorities, the traditional LRU strategy often fails to adapt flexibly to the requirements of GC, resulting in decreased cache efficiency. Therefore, it is necessary to reassess how to manage data in the cache during GC in order to reduce the negative impact of GC on system performance.

To address this issue, we propose the Cache GC Page Stager mechanism. The Cache GC Page Stager aims to enhance system performance by optimizing cache management strategies during the GC process. Fig. 5 shows the schematic representation of the Cache Stager. Cache Stager partitions the LRU linked list into two regions: a normal region and a GC region. Data in the normal region continues to follow the original LRU strategy, thus benefiting from a higher cache hit rate, while the GC region contains pages that are subject to eviction during GC, primarily managing GC-related data to enable more efficient handling. Within the GC region, Cache Stager increases the proportion of GC-related data in the cache by prioritizing the eviction of non-GC-related data. For example, as illustrated in Fig. 5, although E7 is at the tail of the LRU list and would typically be evicted first according to the traditional LRU strategy, the system refrains from immediately evicting it due to its association with GC. Instead, the system first evaluates whether the next node, E6, meets the eviction criteria. If E6 meets the criteria, the system proceeds with its eviction. In this way, Cache Stager effectively adjusts the eviction order, changing it from the traditional (E7, E6, E5, E4) to (E6, E4, E7, E5). This adjustment helps increase the utilization rate of GC data in the cache, reducing unnecessary data migration and mitigating the performance degradation caused by GC.

When searching for cached data that meets the replacement condition in a traditional LRU linked list, typical lookup algorithms (such as CF-LRU and GCaR) start the search from the tail node and proceed towards the head node, as shown in Fig. 6. This method can effectively find the qualifying data in a single search, but its drawback is that each search may require traversing the entire cache space. In some scenarios, for example, if D1 is the data that meets the replacement condition while D2 to D3072 do not, the system has to traverse the entire cache to identify D1 as the replacement target. Such traditional lookup methods incur unacceptable time overheads in embedded devices, such as SSDs.

To address this issue, Cache Stager proposes an improved cache lookup strategy, as illustrated in Fig. 7. Cache Stager uses an intermediate value, test_entry, to record the starting position for the next lookup. Initially, test_entry starts searching from the tail node of the LRU linked list. If no suitable entry is found after inspecting test_num

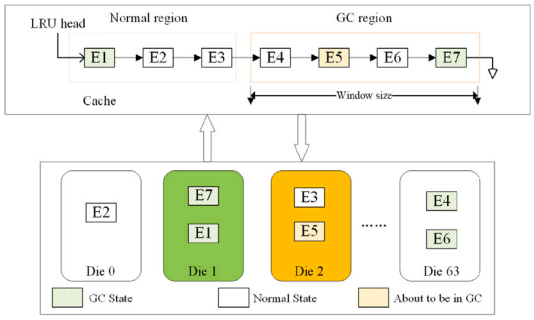


Fig. 5. Cache Stager data distribution status.

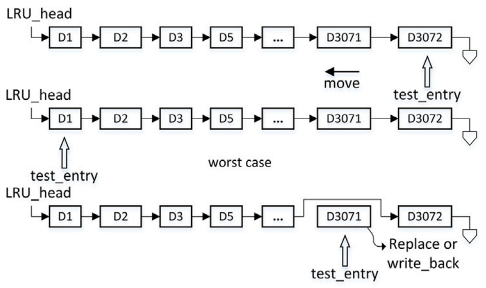


Fig. 6. Schematic diagram of traditional cache lookup mode.

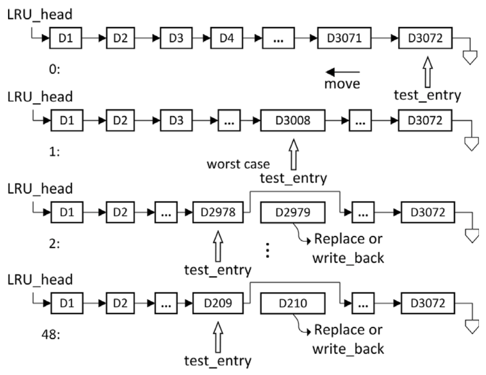


Fig. 7. Schematic diagram of Cache Stager's lookup mode.

entries, test_entry stops moving and designates the tail node of the list as the replacement entry. For instance, upon finding a suitable replacement entry like D2929, the system replaces D2929 and shifts test_entry to the subsequent node, D2928, and subsequent searches will begin from D2928. After the specified number of lookup cycles (cycle_num) is completed, the system will reset test_entry back to the tail node of the LRU linked list.

Algorithm 1 is designed to efficiently manage cache operations within the Cache Stager mechanism. The algorithm starts by determining whether the requested data (*Req_i*) is present in the LRU list (*Lru_List*). If the data is found (*cache hit*), the algorithm processes the request directly: for read requests, it retrieves the data from the cache, and for write requests, it writes the data into the cache. If the data is not found (*cache miss*) and the cache is full, the algorithm selects a cache entry (*L_x*) for replacement based on the *test_entry* parameter. The replacement process involves iterating through candidates over a specified number of iterations (*test_num*). If *L_x* is categorized as GC-Die or High-probability-GC-Die, the *Loop_Lookup* function is invoked to assess its suitability for eviction. Otherwise, if *L_x* corresponds to a write request, its data is written back to flash memory. After determining the replacement, *L_x* is evicted, and the new request (*Req_i*) is added to the cache, with the *Lru_List* updated accordingly. For read requests, the data is loaded from flash memory and returned, while for write requests, the data is written directly to the cache. At the end of the operation, the algorithm resets the parameters (*cycle_num*, *test_num*, and *test_entry*) to prepare for subsequent requests. By combining flexible replacement strategies with efficient handling of cache hits and misses, this algorithm ensures optimal cache utilization and reliable data management.

Algorithm 1 Simplified Cache Management in Cache Stager

```
Require: User request Req_i, parameters cycle_num, test_num, test_entry
Ensure: Efficient handling of Req_i in cache
 1: if Req_i exists in Lru_List then
 2:   /* Cache Hit */
 3:   Handle read/write request directly in cache
 4: else
 5:   /* Cache Miss */
 6:   if Cache is full then
 7:     Select L_x for eviction based on test_entry
 8:   end if
 9:   for each iteration in test_num do
10:     if L_x is in GC-Die or High-probability-GC-Die then
11:       Evaluate eviction using Loop_Lookup
12:     else
13:       if L_x is a write request then
14:         Write L_x back to flash
15:       end if
16:     end if
17:     Replace L_x in cache with Req_i and update Lru_List
18:   end for
19: end if
20: Reset cycle_num, test_num, test_entry
21: End Procedure
```

3.2.2. *Cold and Hot Data Dispatcher*

To further optimize the GC process and reduce unnecessary data migration, CDA-GC introduces a Hot and Cold Data Dispatcher (Dispatcher) at the cache level. This dispatcher categorizes data into hot and cold groups within the cache and processes them separately when writing to flash memory, thereby reducing GC overhead and enhancing system performance.

The Dispatcher first divides data in the cache based on access frequency. Data that is frequently accessed or modified is labeled as hot data, while less frequently accessed data is labeled as cold data. When data needs to be written back to flash memory, the Dispatcher stores hot and cold data into separate flash blocks. Because hot data is updated frequently, the flash blocks storing hot data are more likely to accumulate invalid pages; conversely, cold data is updated less often, so the blocks storing cold data accumulate invalid pages more slowly.

According to the greedy algorithm, the GC process prefers to recycle blocks with a higher number of invalid pages, as these blocks contain less valid data and thus incur lower migration costs. By storing hot and cold data separately, blocks containing hot data accumulate more invalid pages due to frequent updates, while blocks containing cold data see a slower growth of invalid pages, thereby optimizing GC efficiency.

Fig. 8 illustrates the differences between the traditional data allocation approach and our improved hot and cold data separation method. As shown in Fig. 8(a), the traditional approach stores hot and cold data mixed together within the same flash block. This mixed storage strategy introduces several issues during the GC process, significantly increasing system overhead. When a GC operation is required, the system must first migrate all valid data within the block to a new physical block before the original block can be erased. This frequent migration of

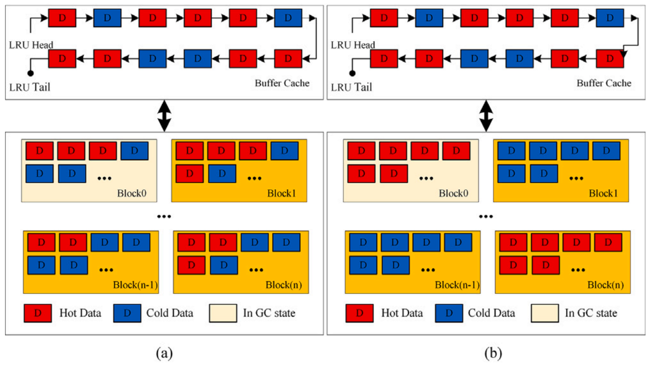


Fig. 8. Dispatcher schematic diagram.

data not only consumes substantial I/O bandwidth but also results in a noticeable decline in performance. For instance, when Block 0 contains both hot and cold data, the migration of valid pages during GC becomes a considerable burden, reducing the efficiency of the GC operation and negatively impacting overall system response time.

In Fig. 8(b), our improved method involves storing hot and cold data in separate blocks. This data separation storage strategy allows for better optimization based on the distinct characteristics of the data. In this arrangement, hot and cold data are no longer mixed in the same block, which leads to more efficient GC operations. Taking Block 0 as an example, when the system performs GC on a block storing hot data, the high update frequency of these pages means that many of them may already be invalid during the GC process. Consequently, the system can directly erase these blocks without the need to migrate a large number of valid pages, thereby simplifying the GC process. Furthermore, since cold data has a relatively low update frequency, storing it in separate blocks results in fewer valid pages requiring migration, thus making GC more efficient. By separating hot and cold data into distinct blocks, our approach effectively reduces data migration and significantly enhances the overall efficiency of GC operations.

To achieve efficient hot and cold data division while ensuring query accuracy, we employ a Bloom filter as the data classification mechanism. Rapid and accurate data classification is crucial for reducing unnecessary data migration during GC. However, in SSD research, algorithm design focuses more on optimizing time and space complexity rather than achieving absolute accuracy.

A Bloom filter is a probabilistic data structure with high space efficiency and fast query speed, suitable for quickly determining whether an element belongs to a set. Its core structure consists of multiple hash functions and a bitmap. When data is written to the cache, multiple hash functions generate hash values for the data, and the corresponding bits in the bitmap are set to 1. To query whether data belongs to the hot data set, the same hash functions compute hash values, and the corresponding bits in the bitmap are checked. If any bit is 0, it can be determined that the data is not in the set; if all bits are 1, the data is possibly in the set.

To reduce the false positive rate and improve query accuracy, we set the number of hash functions to 8. Using 8 hash functions effectively lowers the false positive rate for a given bitmap size. We also set the bitmap size to occupy approximately 5% of the cache space, keeping the false positive rate around 5%. This bitmap size is acceptable within the memory resources of modern SSD controllers and effectively enhances query accuracy.

Furthermore, we utilize the system's built-in memory function memset for resetting. This approach significantly reduces the system's reset time, avoiding unnecessary time overhead. By quickly clearing the bitmap, the system can efficiently update the Bloom filter's state, ensuring the accuracy of hot and cold data determination without impacting overall system performance. This optimization makes the hot and cold data dispatcher more efficient and reliable in practical applications.

The Dispatcher in an SSD system uses a Bloom filter to categorize logical addresses of write requests into hot and cold data segments. When a logical address is mapped, the Bloom filter determines its activity: hot data (status set to 1) is frequently accessed, whereas cold data (status set to 0) is less active. The Dispatcher dynamically tracks hot data, resetting its status to cold if not accessed within a set time frame. During data writing, hot data is placed into hot blocks, and cold data into cold blocks, which enhances garbage collection efficiency. By optimizing data placement based on activity levels, the system reduces GC workload, improves resource utilization, and maintains a balance between computational overhead and performance needs.

The operational algorithm, described in Algorithm 2, is triggered during write operations performed by the SSD. Dispatcher employs a Bloom filter to map the logical slice address associated with the write request. If the logical slice address is present in the Bloom filter's hash table, its status is set to 1, indicating hot data; otherwise, its status is set to 0, indicating cold data. Additionally, the Bloom filter monitors the temporal activity of hot data, identifying any data that remains unaccessed within a predefined timeframe. If hot data surpasses the maximum duration without access, Dispatcher reclassifies it as cold. Ultimately, when data associated with a particular logical slice address is written to flash memory, Dispatcher assesses its status as hot or cold. Hot data is allocated to hot data blocks, while cold data is assigned to cold data blocks, thus optimizing data placement according to activity levels. This approach significantly enhances the efficiency of garbage collection and overall SSD performance.

Algorithm 2 Description of Dispatcher Algorithm

```
Require: Logical slice address requests Lsa_1, Lsa_2, ...Lsa_i stored in
    cache
Ensure: Write cached data Lsa_i to appropriate flash block (hot or cold)

 1: State = Bloom_filter(Lsa_i)
 2: if Bloom_filter_reset_time(Lsa_i) is triggered then
 3:     Reset Bloom_filter_time(Lsa_i)
 4: end if
 5: if State == 1 then
 6:     Write cached Lsa_i data to hot flash block
 7: else
 8:     Write cached Lsa_i data to cold flash block
 9: end if
10: End Procedure
```

4. Performance evaluation

In this section, we first describe the experimental setup and test benchmarks. Then we use the three test benchmarks of Sysbench combined with MySQL, FIO, and TPC-H to evaluate the performance of CDA-GC.

4.1. Experimental setup

The Cosmos+ FPGA OpenSSD development platform is equipped with HYNIX H27Q1T8YEB9R flash memory chips. These flash memory chips are of MLC NAND type with 16 KB pages and 128 pages per block. The number of effective blocks in each die is 8192, and there are a total of 64 dies. The flash memory module adopts 8 channels and 8 ways. The microcontroller of Cosmos+ OpenSSD uses Xilinx's ZYNQ-7000 series chip which contains two ARM Cortex-A9 embedded CPUs, and the controller has 1 GB DRAM to store the metadata like FTL mapping Table and the buffer cache data.Cosmos+ FPGA OpenSSD uses the PCIe interface with NVMe protocol to connect to the host. The NVMe protocol is version 1.2, and the PCIe interface uses Xilinx 7 series

IP cores (PCIe 2.0 version).

The host machine is an Intel Core i7-4790K 4.4 GHz processor with 16 GB DRAM and 256 GB SSD. The operating system is Ubuntu 16.04 based on Linux kernel 4.15 with ext4 file system.

4.2. Benchmark

In order to conduct a comprehensive test of the system, we used three types of benchmarks.

Sysbench [19] is an open source, modular, cross-platform multi-threaded performance testing tool that can be used to perform performance testing of CPU, memory, disk I/O, threads, and databases. We use Sysbench combined with MySQL database for testing. In addition, all the data of pre-warming SSD through block device in the experiment are generated by Sysbench.

FIO-flexible I/O tester [20] is a tool used to generate a large number of threads or processes that perform specific types of I/O operations specified by users. The FIO benchmark test is usually used to test the performance of files and storage systems. We use FIO to evaluate the effect of each scheme on the random read and write performance of the file system.

TPC-H [21] is one of the benchmark programs developed by the Transaction Processing Performance Council. The main purpose of TPC-H is to evaluate the decision support capabilities of specific queries and emphasize the capabilities of the server in data mining, analysis and processing. The benchmark simulates the database operations in the decision support system, tests the response time of complex queries in the database system, and uses the number of queries executed per hour as a metric. The TPC-H test revolves around 22 SELECT statements. Each SELECT is strictly defined, complies with SQL-92 syntax, and does not allow users to modify it. We use TPC-H combined with Postgresql for testing.

4.3. Comparison

Cosmos+ FPGA OpenSSD. Cosmos+ FPGA OpenSSD is an open source SSD development platform. The platform is designed to support the research and education of flash-based solid-state drive technology. The strategy algorithm adopted by its internal firmware is currently the most mainstream and also the most advanced solution.

GCaR implemented in Cosmos+ FPGA OpenSSD. GCaR is proposed by research [22] to improve the performance of SSD systems in garbage collection. The algorithm was initially implemented in the simulator Disksim, and we implement GCaR on OpenSSD. By modifying the data buffer of Cosmos+ FPGA OpenSSD, when the system performs cache replacement, the cache entries related to garbage collection will be temporarily stored to improve the response speed of the system during garbage collection. We embed GCaR into the firmware part of Cosmos+.

Co-Active implemented in Cosmos+ FPGA OpenSSD. Co-Active is proposed by research [23] to improve the response performance of SSD. The algorithm was initially implemented in the MQSIM simulator [24], and we implement Co-Active on Cosmos+ FPGA OpenSSD. Two linked lists are used to store different types of data by setting up two linked lists in the data buffer of Cosmos+ FPGA OpenSSD. The clean linked list is used to store read request type data or free entries, and the dirty linked list is used to store write request type data. Co-Active uses Bloom filters to classify hot and cold data, and uses active write-back to preferentially write the cold data in the data buffer under the free channel to the flash memory particles. We embed Co-Active into the firmware part of Cosmos+.

4.4. Run-time performance

In this part, we will compare the performance of each strategy in the actual test. Including Sysbench test on MySQL performance, FIO test on ext4 file system, and TPC-H test on Postgresql database.

4.4.1. Sysbench benchmark

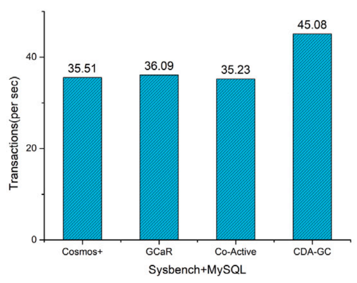
The text outlines the significant benefits of the CDA-GC strategy for enhancing MySQL database performance, particularly in the context of Sysbench testing. This strategy not only improves the transaction and query processing capabilities of the MySQL database but also effectively reduces system latency, internal data migration within SSDs, and the number of block erasures. By taking into account the distribution of GC-related data in the cache and the state of data movement from the cache to flash memory, the CDA-GC method significantly mitigates the impact of GC data streams on normal I/O data flows, thereby enhancing transaction processing and query performance. Additionally, the reduction in GC data flow decreases internal data migration within the SSD, thus reducing write amplification and the number of block erasures. Moreover, the lightweight design of the algorithm minimizes latency impacts on I/O performance, avoiding unnecessary overhead and achieving excellent results on actual physical SSDs.

Fig. 9 illustrates the performance of the database in the context of Sysbench and MySQL. In Fig. 9(a), compared to OpenSSDs employing other schemes, the OpenSSD utilizing the CDA-GC scheme improved the database transaction processing capacity by 27%, 25%, and 28%, respectively. Similarly, Fig. 9(b) demonstrates that CDA-GC achieved comparable advantages in database querying capability. The number of transactions and queries executed per second directly reflects the performance of the database, and the OpenSSD based on the CDA-GC scheme achieved the best results in both metrics. The reason behind this phenomenon is that data migration during GC can severely affect the normal read and write operations of the database. By reasonably allocating data between the cache and flash memory, CDA-GC effectively reduces the overhead of migrating invalid data during GC, minimizing the impact of GC data streams on normal data flows. It is noteworthy that the OpenSSD based on GCaR did not achieve the expected advantage in this test. Although GCaR considered the impact of data distribution in the cache on GC performance, it overlooked the importance of lightweight algorithm design, resulting in performance gains being offset by increased algorithm latency. In Fig. 9(c), the OpenSSD employing the CDA-GC scheme achieved the best results in terms of average database latency compared to other schemes, reducing latency by 21.2%, 20%, and 21.4%, respectively, compared to Cosmos+, GCaR, and Co-Active's CDA-GC.

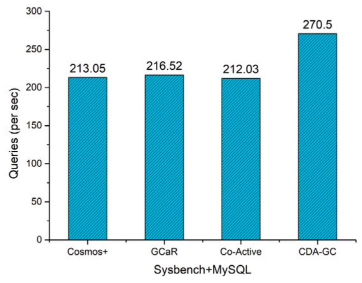
Fig. 10 depicts the performance of the OpenSSD side under the combination of Sysbench and MySQL. In Fig. 10(a), there is little difference in cache hit counts among the various schemes, as they are all based on the LRU design. However, CDA-GC improved the cache hit count by 6%, 4.6%, and 15.7% compared with Cosmos+, GCaR, and Co-Active, respectively. In Fig. 10(b), due to the rational allocation of hot and cold data, CDA-GC reduced the number of block erasures by 10.5%, 8.3%, and 10.9%, respectively, compared with other schemes. Fig. 10(c) reflects the number of page migrations associated with GC within the OpenSSD, which effectively measures the efficiency of data migration for each algorithm during GC. Compared with Cosmos+, GCaR, and Co-Active, the CDA-GC scheme reduced this metric by 25.5%, 21%, and 25%, respectively. This clearly reflects the advantage of the rational data distribution and allocation method employed by CDA-GC. A well-designed data distribution mechanism can indeed reduce unnecessary data migration during GC, thereby enhancing normal read and write performance.

4.4.2. FIO benchmark

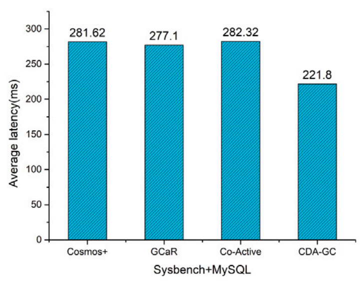
In the FIO workload testing, we evaluated the performance of various schemes by setting read and write ratios to 25%, 35%, 50%, 65%, and 75%. Prior to the testing, the initialized SSD was preheated with 1TB of data. During the testing process, FIO utilized 100 GB of data. We recorded the total read and write response time of the SSD during runtime, as well as the IOPS and bandwidth statistics obtained from the FIO testing program. At the file system level, the tested metrics included IOPS and bandwidth, while at the SSD level, the tested metrics



(a) Transaction

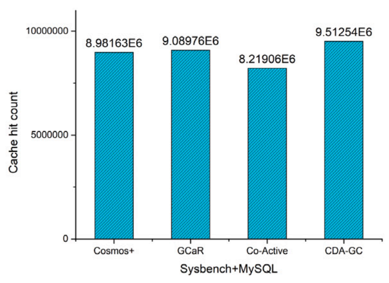


(b) Queries

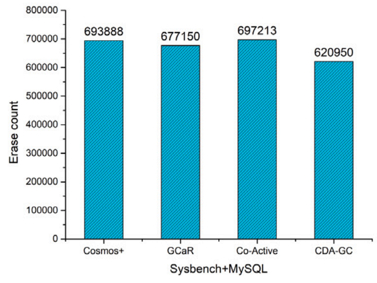


(c) Latency

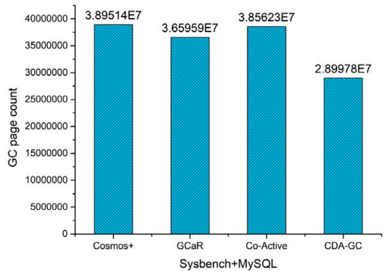
Fig. 9. The performance of the database side under Sysbench combined with MySQL.



(a) Cache hit count



(b) Erase count



(c) GC page count

Fig. 10. The performance of OpenSSD side under Sysbench combined with MySQL.

included read and write response times.

Fig. 11 shows the read response times of each scheme under different read and write ratios. From Fig. 11, it is evident that CDA-GC outperforms Cosmos+, GCaR, and Co-Active in terms of read response times across all read and write ratios. Specifically, the average read response time of CDA-GC is 31.3% lower compared to Cosmos+, 37.9% lower compared to GCaR, and 52.5% lower compared to Co-Active. Particularly in scenarios with a higher proportion of writes, CDA-GC's advantage becomes even more pronounced. When the write ratio reaches 75%, the read response time of CDA-GC is reduced by 41.9%, 48.3%, and 48.6% compared to Cosmos+, GCaR, and Co-Active, respectively. These results indicate that CDA-GC has a significant performance advantage in high write ratio scenarios, especially in terms of read request response.

Fig. 12 presents the write response times of each scheme under different read and write ratios. From Fig. 12, it can be observed that CDA-GC also exhibits better write response times compared to Cosmos+, GCaR, and Co-Active across different read and write ratios. The average write response time of CDA-GC is reduced by 10% compared to Cosmos+, by 48.1% compared to GCaR, and by 19.3% compared to Co-Active. Specifically, when the write ratio is 75%, the write response time of CDA-GC is reduced by 12.5%, 50%, and 21.2% compared to Cosmos+, GCaR, and Co-Active, respectively. These data demonstrate that in high write ratio scenarios, CDA-GC can significantly reduce write response times, resulting in better performance under strong GC triggering conditions.

Fig. 13 illustrates the read IOPS of each scheme under different read and write ratios. Fig. 13 clearly shows that CDA-GC demonstrates superior read IOPS performance compared to Cosmos+, GCaR, and Co-Active. At a 75% write ratio, CDA-GC's read IOPS are 79.6% higher than Cosmos+, 89% higher than GCaR, and 103.9% higher than Co-Active. This indicates that CDA-GC has a significant advantage in maintaining high read throughput, particularly in write-intensive scenarios. Fig. 14 shows the write IOPS of each scheme under different read and write ratios. Fig. 14 shows that CDA-GC also exhibits excellent performance in write IOPS. At a 75% write ratio, CDA-GC's write IOPS are 79.6% higher than Cosmos+, 89% higher than GCaR, and 103.9% higher than Co-Active. This demonstrates that CDA-GC is highly effective in managing write operations, especially in scenarios with a high write ratio

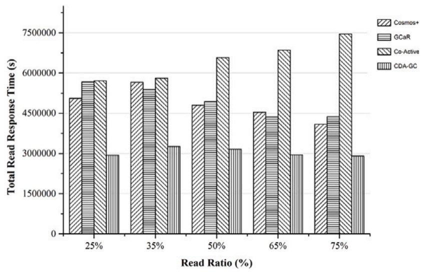


Fig. 11. Read response time under FIO.

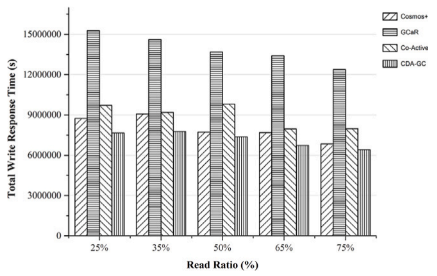


Fig. 12. Write response time under FIO.

Fig. 15 shows the read bandwidth of each scheme under different read and write ratios. CDA-GC outperforms Cosmos+, GCaR, and Co-Active in terms of average read bandwidth. At a 75% write ratio, CDA-GC's read bandwidth is 76.2%, 89%, and 103.8% higher compared to Cosmos+, GCaR, and Co-Active, respectively. Fig. 16 presents the write bandwidth of each scheme. CDA-GC's average write bandwidth is significantly higher than the other algorithms. At a 75% write ratio, CDA-GC's write bandwidth is 79.6%, 89%, and 103.9% higher

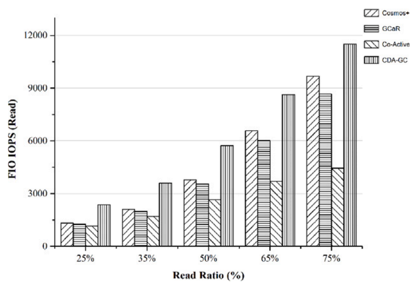


Fig. 13. Read IOPS under FIO.

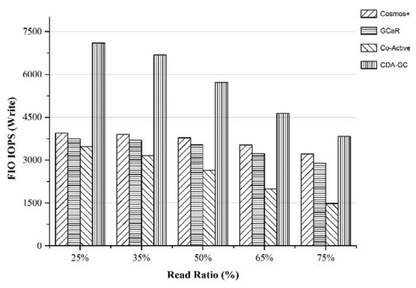


Fig. 14. Write IOPS under FIO.

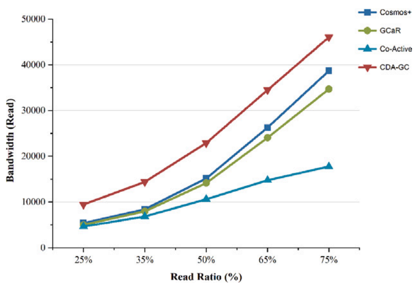


Fig. 15. Read bandwidth under FIO.

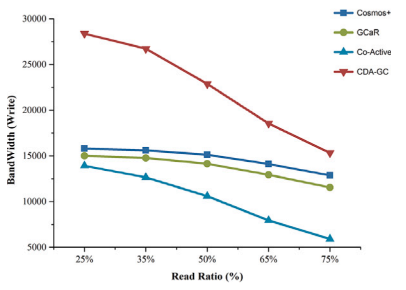


Fig. 16. Write bandwidth under FIO.

than Cosmos+, GCaR, and Co-Active. The performance of SSD directly determines the bandwidth, and OpenSSD employing CDA-GC exhibits superior performance, leading to an enhancement in bandwidth. The CDA-GC approach effectively boosts the read and write bandwidth of the SSD system during garbage collection.

CDA-GC demonstrates superior performance across key metrics such as read and write response time, IOPS, and bandwidth, particularly excelling in high write ratio scenarios. As the write ratio decreases, CDA-GC's advantage diminishes gradually, yet it consistently outperforms other solutions. This is largely because CDA-GC effectively manages GC by considering data distribution at the cache level, thereby minimizing conflicts between user I/O and GC processes, enhancing SSD read and write performance, and improving data migration efficiency. In the FIO test, despite weaker data correlation compared to the MySQL database test, the strong GC-triggered scenario still highlighted CDA-GC's effectiveness. By optimizing caching and reducing interference from invalid data streams, CDA-GC maintains significant performance advantages. In contrast, GCaR's lack of lightweight design and its complex algorithms resulted in a decline in performance, while Co-Active, although incorporating I/O optimizations at the cache level, lacked specific enhancements for GC scenarios and thus demonstrated average results. Overall, CDA-GC excels in file system testing, particularly under high write ratios and strong GC conditions, showcasing its superiority in optimizing I/O performance and reducing system response time compared to other existing schemes.

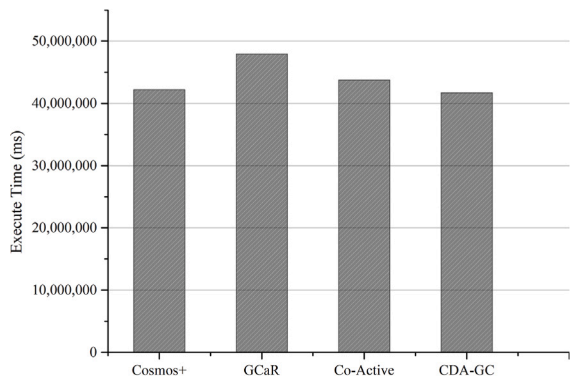


Fig. 17. TPC-H 22 query statements execution time.

4.4.3. *TPC-H benchmark*

In the TPC-H test, to verify the generality of the CDA-GC scheme, we evaluated the performance of each algorithm under low GC trigger conditions. In the TPC-H workload, we deployed a PostgreSQL database on the Cosmos+ FPGA OpenSSD to execute TPC-H queries. Similarly, TPC-H used 1TB of data for preheating and 100 GB of data for testing. During the TPC-H test, we recorded the execution times of the 22 query statements in TPC-H.

Fig. 17 shows the read response time and write response time of each scheme running TPC-H. From Fig. 17, it can be observed that the read response times of CDA-GC, Cosmos+, and Co-Active are almost identical, while the write response time of CDA-GC is reduced by 6.1%, 10.2%, and 20.1%, respectively, compared to Cosmos+, GCaR, and Co-Active. The reason why the performance of CDA-GC in TPC-H is not as strong as in other test scenarios lies in the fact that the TPC-H workload exhibits a much higher read ratio compared to the write ratio. Applications with a relatively high read ratio tend to modify less data, resulting in a lower proportion of garbage collection triggered by the system. Consequently, when the system triggers less garbage collection, CDA-GC cannot fully leverage its advantages.

Furthermore, the performance of Cosmos+, GCaR, and Co-Active is closely related to their respective complexities. In the TPC-H application, characterized by a high read ratio, the benefits of each algorithm cannot compensate for the performance degradation caused by their inherent complexities. Despite the improvements in database performance not being as significant as in other tests, CDA-GC still achieved the best results in terms of performance.

5. Related work

In modern computer systems, storage has become a serious shortcoming of computer systems. The emergence of SSD has effectively alleviated this gap. Usually, simulators are used to evaluate their designs in SSD research. Commonly used simulators are SSDModel [5], DISKSim [22], SSDSim [25], WiscSim [26], FlashSim [27]. With the popularity of NVMe SSDs, the existing SSD emulators have fallen behind and have several shortcomings. Recently, the first simulator MQSim [24] supporting the NVMe protocol was proposed. Although MQSim has greatly improved compared with previous simulators, there are still some errors compared with real SSDs. However, these models have certain limitations, and they are not open source to the public, so they cannot be used by other research teams. Recently, the first open source SSD named Cosmos+ that supports the NVMe protocol was designed. Cosmos+ OpenSSD can freely modify its software and hardware parts and open its source code to the public [28].

In order to improve the performance of the SSD, the buffer cache is introduced into the SSD. However, common cache replacement algorithms are proposed for disks and cannot fully adapt to flash-based SSDs. Flash memory has asymmetric read and write characteristics. To optimize SSD cache performance, CFLRU [29] splits the LRU into two areas to store clean and dirty pages respectively, and preferentially select clean pages when selecting eviction pages. However, because the frequency of accessing buffer pages is not considered, CFLRU tends to retain older dirty pages when replacing eviction pages, but evokes hot clean pages, which will reduce the cache hit rate. In order to reduce the number of write operations, LRU-WSR [30] adopts a method of delaying eviction of a dirty page with higher access frequency to effectively reduce the number of write operations and erase operations. However, LRU-WSR does not consider the access frequency of clean pages. Compared with expelling cold clean pages, expelling hot clean pages is more expensive. AD-LRU [31] divides the buffer into a cold zone and a hot zone. The cold zone stores pages that have been accessed only once, and the hot zone stores pages that have been accessed at least twice. CASA [32] proposes a cost-based adaptive buffer replacement scheme, which can be applied to flash memory devices with different read and write cost ratios. GC-ARM [33] dynamically destroys consecutive pages in the entire block or a single page from the write buffer to improve GC efficiency. In order to improve the performance of random writes, BPLRU [34] proposes a block-level page filling technology based on the FTL layer to minimize the buffer refresh cost, but it will increase unnecessary read and write operations. Although the above caching strategies can improve the performance of SSD in certain situations, however, these strategies lack the consideration of the impact of GC. GC-Cache [35] introduces a novel approach of incorporating additional cache to address GC issues, yet this cache management strategy may encounter performance bottlenecks under extreme load conditions, increases system complexity, and relies heavily on a larger RAM cache. GFTL [36] uses a group-level mapping to reduce the mapping table size while maintaining flexibility similar to page-level mapping, thus improving cache hit ratios and performance. However, the additional computation for group-level address translation and the need for garbage collection in low-space scenarios can result in performance overheads. DL-FTL [37] employs a dual-locality approach to exploit both temporal and spatial locality, significantly improving cache hit ratios and reducing average response times, especially for small sequential requests. However, managing the sequential mapping tables introduces additional data structures and overhead, which can complicate the implementation and lead to space inefficiencies in certain cases. CRFTL [38] employs a dynamic cache reallocation strategy that adapts to different I/O request types based on their life cycles. This approach combines heuristics with reinforcement learning to effectively optimize cache usage and enhance overall performance for smartphones. However, this strategy introduces additional complexity to the cache reallocation process, and may lead to significant computational overhead, especially in scenarios with frequent state changes. GCaR [22] considers the impact of data distribution in the cache on GC and gives higher priority to the cached data blocks belonging to the flash memory chip in the GC state to reduce the conflict between user input I/O operations and GC-induced I/O operations. However, GCaR uses a sequential method to search for GC-related data in the cache. This sequential search scheme will add additional time overhead and cause performance degradation. Caching-aware GC [39] while enhancing SSD performance, introduces increased system complexity and a heavy dependence on large RAM caches. Co-Active [23] adopts an active write-back strategy to preferentially write the cold and dirty cache data in the idle channel in the cache back to the Flash. However, Co-Active also lacks the consideration of GC and performs generally in GC-triggered scenarios. In addition, most of the above algorithms have not been designed and verified on real SSDs, and their practicability needs to be verified.

6. Conclusion

Garbage collection is the most crucial factor affecting the performance of SSDs. As flash-based SSDs increasingly replace HDDs in the market, enhancing garbage collection performance has become an immediate and critical challenge. In SSDs, the use of cache to intelligently distribute data greatly impacts garbage collection efficiency. Our CDA-GC strategy effectively leverages cache to account for the internal data distribution within SSDs, optimally allocating data across both cache and flash memory to improve garbage collection efficiency. We have implemented CDA-GC on a real device, the Cosmos+ OpenSSD, and extensive testing on real file systems and databases has shown that our CDA-GC approach significantly boosts SSD performance. By focusing on cache optimization, this research not only deepens our understanding of garbage collection mechanism optimization in SSDs but also introduces a new avenue for enhancing SSD performance.

Despite the significant progress our CDA-GC strategy has made under current experimental loads, several key factors must be considered for its application in actual products. Future work will focus on simplifying parameter adjustments to ensure the algorithm can adaptively adjust based on different usage scenarios, while also paying attention to long-term performance stability. This will ensure that the CDA-GC strategy can continue to provide optimized performance in real-world environments, including conducting long-term stability tests to evaluate efficiency under various workloads and data distribution conditions. Through these efforts, we aim to make CDA-GC not only theoretically effective but also a reliable and sustainable improvement in actual products, offering users a superior SSD experience.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

[1] S.H. Lim, K.H. Park, An efficient NAND flash file system for flash memory storage, IEEE Trans. Comput. 55 (7) (2006) 906–912.
[2] J.H. Kim, S.H. Jung, Y.H. Song, Cost and performance analysis of nand mapping algorithms in a shared-bus multi-chip configuration, in: IWSSPS'08: The 3rd International Workshop on Software Support for PorTable Storage, vol. 3, (3) 2008, pp. 33–39.
[3] F. Chen, D.A. Koufaty, X. Zhang, Hystor: Making the best use of solid state drives in high performance storage systems, in: Proceedings of the International Conference on Supercomputing, 2011, pp. 22–32.
[4] J. Hu, H. Jiang, P. Manden, Understanding performance anomalies of ssds and their impact in enterprise application environment, ACM SIGMETRICS Perform. Eval. Rev. 40 (1) (2012) 415–416.
[5] N. Agrawal, V. Prabhakaran, T. Wobber, et al., Design tradeoffs for SSD performance, in: USENIX Annual Technical Conference, 2008, p. 57.
[6] J. Lee, Y. Kim, G.M. Shipman, et al., A semi-preemptive garbage collector for solid state drives, in: (IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software, IEEE, 2011, pp. 12–21.

[7] S. Lee, D. Shin, J. Kim, BAGC: Buffer-aware garbage collection for flash-based storage systems, IEEE Trans. Comput. 62 (11) (2012) 2141–2154.
[8] L.P. Chang, T.W. Kuo, S.W. Lo, Real-time garbage collection for flash-memory storage systems of real-time embedded systems, ACM Trans. Embed. Comput. Syst. (TECS) 3 (4) (2004) 837–863.
[9] X.Y. Hu, E. Eleftheriou, R. Haas, et al., Write amplification analysis in flash-based solid state drives, in: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference, 2009, pp. 1–9.
[10] M. Huang, L. Men, Improving the performance of on-board cache for flash-based solid-state drives, in: 2012 IEEE Seventh International Conference on Networking, Architecture, and Storage, IEEE, 2012, pp. 283–287.
[11] B. Van Houdt, A mean field model for a class of garbage collection algorithms in flash-based solid state drives, ACM SIGMETRICS Perform. Eval. Rev. 41 (1) (2013) 191–202.
[12] W. Bux, I. Iliadis, Performance of greedy garbage collection in flash-based solid-state drives, Perform. Eval. 67 (11) (2010) 1172–1186.
[13] 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, IEEE, 2014, pp. 524–535.
[14] N. Shahidi, M.T. Kandemir, M. Arjomand, et al., Exploring the potentials of parallel garbage collection in ssds for enterprise storage systems, in: SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2016, pp. 561–572.
[15] Y. Kim, S. Oral, G.M. Shipman, et al., Harmonia: A globally coordinated garbage collector for arrays of solid-state drives, in: 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies, MSST, IEEE, 2011, pp. 1–12.
[16] S. Lee, D. Shin, Y.J. Kim, et al., LAST: locality-aware sector translation for NAND flash memory-based storage systems, Oper. Syst. Rev. 42 (6) (2008) 36–42.
[17] D. Park, D.H.C. Du, Hot data identification for flash-based storage systems using multiple bloom filters, in: 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies, MSST, IEEE, 2011, pp. 1–11.
[18] M. Jung, W. Choi, S. Srikantaiah, et al., HIOS: A host interface I/O scheduler for solid state disks, ACM SIGARCH Comput. Archit. News 42 (3) (2014) 289–300.
[19] A. Kopytov, Sysbench: a system performance benchmark, 2004, http://sysbench.sourceforge.net/.
[20] J. Axboe, FIO (Flexible IO Tester), http://git.kernel.dk/?p=fio.git;a=summary.
[21] http://www.tpc.org/tpch/.
[22] S. Wu, B. Mao, Y. Lin, et al., Improving performance for flash-based storage systems through GC-aware cache management, IEEE Trans. Parallel Distrib. Syst. 28 (10) (2017) 2852–2865.
[23] H. Sun, S. Dai, J. Huang, et al., Co-active: A workload-aware collaborative cache management scheme for NVMe SSDs, IEEE Trans. Parallel Distrib. Syst. 32 (6) (2021) 1437–1451.
[24] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, et al., 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.
[25] Y. Hu, H. Jiang, D. Feng, et al., 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.
[26] J. He, S. Kannan, A.C. Arpaci-Dusseau, et al., The unwritten contract of solid state drives, in: Proceedings of the Twelfth European Conference on Computer Systems, 2017, pp. 127–144.
[27] Y. Kim, B. Tauras, A. Gupta, et al., Flashsim: A simulator for nand flash-based solid-state drives, in: 2009 First International Conference on Advances in System Simulation, IEEE, 2009, pp. 125–131.
[28] J. Kwak, S. Lee, K. Park, et al., Cosmos+ OpenSSD: Rapid prototype for flash storage systems, ACM Trans. Storage (TOS) 16 (3) (2020) 1–35.
[29] S. Park, D. Jung, J. Kang, et al., 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.
[30] H. Jung, H. Shim, S. Park, et al., LRU-WSR: integration of LRU and writes sequence reordering for flash memory, IEEE Trans. Consum. Electron. 54 (3) (2008) 1215–1223.
[31] P. Jin, Y. Ou, T. Härder, et al., AD-LRU: An efficient buffer replacement algorithm for flash-based databases, Data Knowl. Eng. 72 (2012) 83–102.
[32] Yi Ou, Theo Härder, Clean first or dirty first? a cost-aware self-adaptive buffer replacement policy, in: Proceedings of the Fourteenth International Database Engineering & Applications Symposium, 2010.
[33] J. Hu, H. Jiang, L. Tian, et al., GC-ARM: garbage collection-aware RAM management for flash based solid state drives, in: 2012 IEEE Seventh International Conference on Networking, Architecture, and Storage, IEEE, 2012, pp. 134–143.
[34] H. Kim, S. Ahn, BPLRU: A buffer management scheme for improving random writes in flash storage, in: FAST, vol. 8, 2008, pp. 1–14.
[35] Wei Xie, Yong Chen, A cache management scheme for hiding garbage collection latency in flash-based solid state drives, in: 2015 IEEE International Conference on Cluster Computing, IEEE, 2015.
[36] Yubiao Pan, et al., GFTL: Group-level mapping in flash translation layer to provide efficient address translation for NAND flash-based SSDs, IEEE Trans. Consum. Electron. 66 (3) (2020) 242–250.
[37] Yuhan Luo, et al., Dual locality-based flash translation layer for NAND flash-based consumer electronics, IEEE Trans. Consum. Electron. 68 (3) (2022) 281–290.
[38] Jianpeng Zhang, et al., CRFTL: cache reallocation-based page-level flash translation layer for smartphones, IEEE Trans. Consum. Electron. 69 (3) (2023) 671–679.
[39] Yubiao Pan, et al., Caching-aware garbage collection to improve performance and lifetime for NAND flash SSDs, IEEE Trans. Consum. Electron. 67 (2) (2021) 141–148.