RDA: A Read-Request Driven Adaptive Allocation Scheme for Improving SSD Performance

Shujie Pang, Yuhui Deng, Zhaorui Wu, Genxiong Zhang, Jie Li, and Xiao Qin

***Abstract*—The parallel operation technology plays a pivotal role in enhancing performance of 3-D NAND flash-based SSDs. High-parallel distribution of consecutive pages places the pages on different parallel units, thereby improving the parallelism and throughput of read requests. However, the high-parallel distribution generates two problems: 1) aggravating data fragmentation and 2) exacerbating the impact of garbage collection (GC) on latency. Moreover, small reads only require a few parallel units, and thus the high-parallel distribution is redundant for the requests. To address this issue, we propose a read-request driven adaptive allocation scheme called RDA to bolster SSD performance by adaptively adjusting the parallel distribution of consecutive pages. The RDA scheme employs the size of historical read requests to gauge the level of parallelism for write requests with varying sizes. Then, RDA allocates the logical pages of writes to distinct parallel units according to the parallelism of the requests. In doing so, RDA effectively mitigates the performance degradation of SSDs caused by redundant parallel distribution, while preserving the parallelism of read requests. We compare RDA with the three state-of-art schemes *Amphibian*, SOML, and *Preemptive GC* in terms of GC-blocked read requests, GC counts, and read response time under eight real-world workloads. The experimental results unveil that compared with the existing schemes, RDA revamps the GC-blocked read requests, GC counts, and read response time by averages of 20.6%, 7.8%, and 15.8%, respectively.**

***Index Terms*—3-D NAND flash, data allocation, parallel distribution, request size.**

I. Introduction

NAND flash-based solid-state drives (SSDs) have become a mainstream storage medium, renowned for their fast access speed, stable performance, and low-energy consumption. SSDs are adopted in a wide variety of fields, such as mobile devices, embedded systems, personal computers, and data centers. SSDs support multiple levels of parallelism, including channel, chip, and plane, to facilitate request concurrency that achieves high performance [1], [2], [3]. Moreover, as the 3-D NAND technology increases store density by stacking multiple layers of planar NAND, 3-D NAND flash has replaced 2-D NAND as the mainstream storage device in the market [4]. Since a single 3-D NAND flash chip has a higher capacity compared with 2-D NAND, 3-D NAND flash-based SSDs have fewer chips in the same capacity, which suffer from reduced parallel units [5], [6]. Reasonable allocation of limited parallel resources significantly upgrades the overall performance of SSDs.

Flash manufacturers implement an advanced command (multiplane operations) to reshape the internal parallelism of a flash chip, thereby boosting SSD performance [7], [8]. NAND flash-based SSDs specify a page as the unit of read/write operations and the block as the unit of erase operations. Multiplane operations support multiple planes to perform read, write, or erase operations simultaneously, and take full advantage of the parallelism provided by SSDs. Meanwhile, data allocation strategies play a crucial role in deciding how to allocate logical data to physical addresses in SSDs. Generally, dynamic allocation strategies assign the consecutive pages of a write request to distinctive planes in chips. These pages are written to the planes simultaneously through multiplane program operations, thereby optimizing the parallel performance of SSDs by maximizing the parallel distribution of the consecutive pages. Then, subsequent read requests access the consecutive pages from multiple planes by multiplane read operations, thus effectively improving the read performance of SSDs. As a result, high-parallel distribution enhances the parallelism of read requests. Tavakkol et al. [9] experimentally reveal that dynamic allocation strategies can take full advantage of the internal parallelism of SSDs. Unfortunately, the read advantage of maximizing data parallel distribution is offset by garbage collection (GC) operations.

GC, as a critical function of SSDs, is tasked with eliminating invalid space in SSDs [10]. During a GC, the valid data of a block is migrated to other blocks, and then the block is erased to generate a new free block [11]. However, the high-parallel distribution of consecutive pages falls short on data locality and exacerbates data fragmentation, which expands migrated data and raises overhead during a GC. In addition, flash manufacturers implement an advanced command (Copyback) to compress GC operations in planes [12]. The GC-ing plane blocks requests that are assigned to the plane. Unfortunately, multiple planes accessed by a read request will surge the likelihood of the request being blocked by GCs. Therefore, GC operations impose a new requirement for the data allocation

Manuscript received 10 February 2024; revised 3 June 2024; accepted 26 July 2024. Date of publication 29 July 2024; date of current version 22 January 2025. This work was supported in part by the National Natural Science Foundation of China under Grant 62072214, and in part by the Guangdong Basic and Applied Basic Research Foundation under Grant 2021B1515120048. This article was recommended by Associate Editor C. Yang. *(Corresponding author: Yuhui Deng.)*

Shujie Pang, Yuhui Deng, Zhaorui Wu, Genxiong Zhang, and Jie Li are with the Department of Computer Science, Jinan University, Guangzhou 510632, China (e-mail: p_shujie@163.com; tyhdeng@jnu.edu.cn; diom_wu@163.com; zgx787839110@163.com; lijiegxmd11@163.com).

Xiao Qin is with the Department of Computer Science and Software Engineering, Auburn University, Auburn, AL 36830 USA (e-mail: xqin@auburn.edu).

Digital Object Identifier 10.1109/TCAD.2024.3435681

1937-4151 © 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:22:49 UTC from IEEE Xplore. Restrictions apply.

schemes: ensuring the read parallelism while reducing data parallel distribution.

Furthermore, it is worth noting the parallel distribution of consecutive pages only offers a limited improvement for the parallelism of read requests. The primary reason is that the high-parallel distribution of consecutive pages is redundant for small read requests. Specifically, the consecutive pages of a large write request are sequentially allocated to multiple planes. When subsequent small reads only access the pages on the partial planes, the parallel distribution of the remaining pages on the other planes becomes unnecessary. The redundant parallel distribution will exacerbate data fragmentation and the impact of GCs on I/O latency.

Motivated by the above observation and analysis, we propose RDA—a read-request driven adaptive allocation scheme for maximizing the utilization of limited parallel resources in 3-D NAND flash-based SSDs. To magnify the advantages of parallel distribution, RDA dynamically adjusts the parallelism of write requests via the size of historical read requests, thereby safeguarding read parallelism for reads. Meanwhile, according to the adjusted parallelism and page distance, this novel scheme places the consecutive pages of the write requests on different parallel units, effectively mitigating redundant parallel distribution of the consecutive pages. Our contributions can be summarized as follows.

1) We unveil two intriguing observations: a) maximizing the parallel distribution of consecutive pages exacerbates the adverse impact of GCs on reads and b) the majority of read requests are small in size.
2) We propose a read-request driven adaptive allocation scheme—RDA—that reasonably allocates the plane addresses of consecutive pages by the size of read requests. The scheme effectively alleviates the impact of the redundant parallel distribution on read performance, thereby optimizing the performance of 3-D NAND flash.
3) We exhibit the superiority of RDA over the other schemes in terms of GC-blocked read requests, GC counts, and read response time under the real-world workloads.

The remainder of this article is organized as follows. Sections II and III present in related work and background, respectively. Our motivation is discussed in Section IV. We introduce the design of RDA in Section V and the performance evaluation is presented in Section VI. Finally, Section VII concludes this article with comments.

II. Related Work

Parallel operation and GC, as the pivotal functions of NAND flash-based SSDs, have received considerable attention from researchers and institutions. Moreover, data allocation, as an important part of assigning physical addresses to logical pages, has garnered significant interests from academia and industry.

A. Parallel Operation

To take full advantage of the parallel operations in SSDs, a handful of optimization approaches have been proposed to improve the utilization of parallel units [1], [13]. Gao et al. [14] stimulated the potential of multiplane commands by distributing requests to multiple planes at one time. Furthermore, Gao et al. [15] proposed a from-plane-to-die parallel optimization framework to further exploit plane-level parallelism. Since the increased single-chip capacity reduces parallelism, Kim et al. [6] implemented a novel NAND flash memory architecture to exploit plane-level parallelism. These parallel strategies serve requests by maximizing plane-level parallelism, thus enhancing the efficiency of processing requests issued by a host. However, plane-level parallel operations have some negative impacts on SSD performance. In order to alleviate page waste incurred by multiplane operations, Seong et al. [16] designed a super page to group all pages at the same in-chip location as a large logical page. And Tavakkol et al. [17] designed a twin block management scheme to symmetrically read/write and reclaim the flash block addresses on the planes of a die. In addition, Liu et al. [5] proposed a novel parallel partial read strategy-SOML, which groups subpage-sized read requests from different layers to upgrade the operation parallelism and read performance. Unfortunately, aforementioned schemes are focused on the benefits of maximizing read/write parallelism without recognizing the adverse impact of excessive parallel distribution on SSD performance. In contrast, our RDA not only maintains the high parallelism inside an SSD but also alleviates conflicts between GCs and the parallel distribution of consecutive pages, resulting in a substantial advancement over previous techniques in SSD performance optimization.

B. Garbage Collection

GC is one of the primary factors affecting SSD performance. Although SSD manufacturers bring forth Copyback to alleviate the negative impact of GCs, GC still blocks I/O requests issued by a host [8]. There have been numerous works on the in-depth study of GC optimization [10], [11], [18], [19]. To alleviate the adverse impact of erase operations on I/O requests during the GC process, schemes, such as partial erase [11], preemption GC [20], and erase suspension [21], were proposed to raise the priority of user I/O requests. Meanwhile, as the number of pages per block in 3-D NAND flash increases, data migration during GCs has become the main factor of GC overhead, and GCs seriously impede I/O requests by consuming channel resources. Yan et al. [22] proposed TTFLASH to alleviate the tiny tail latency caused by data migration during GCs, which eliminates channel blocking through parity-based RAIN and Copyback. Shahidi et al. [23] proposed an active parallel GC strategy based on plane-level parallelism to improve read and write efficiency. Moreover, Mao et al. [24] devised a new I/O scheduler—Amphibian (AMP)—for SSDs, where the impact of GCs on requests is averted: request response time is shortened by adjusting request priorities. Regrettably, the above studies cannot fundamentally curb the number of GCs, and these existing techniques overlook conflicts between data parallel distribution and GCs. Unlike the aforementioned schemes, our RDA curtails the redundant parallel distribution of consecutive pages by the size of read requests, thereby mitigating data fragmentation and the detrimental impact of

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:22:49 UTC from IEEE Xplore. Restrictions apply.

GCs on reads. Our experimental evaluations show that RDA maintains read parallelism for contiguous data and slashes the probability of requests being blocked by GCs.

C. Data Allocation

Data allocation, as a key technology affecting SSD performance, has been studied in depth by numerous researchers. A wide range of works have delved into dynamic and static allocation strategies [25] to fully utilize the multichannel advantage of SSDs. For example, Hu et al. [26] discussed advantages and disadvantages of the dynamic and static allocation schemes. Tavakkol et al. evaluated the performance improvement of SSDs with dynamic allocation schemes, which can well balance I/O requests. Furthermore, to address the new challenges introduced by 3-D NAND flash, some studies adopt data allocation strategies to slash read and write disturb and mitigate the negative impact of GC [27], [28], [29]. For instance, Uchigaito et al. [28] proposed a new allocation scheme for getting rid of GC during big-graphs analysis, and the scheme enables high-speed analysis of big graphs in NAND flash memory. Chang et al. developed a data allocation strategy to program logical pages to word-lines with no programmed word-lines around [30], thus effectively alleviating program disturb. These aforementioned allocation schemes, however, fall short on balancing data locality and parallelism—being unable to efficiently utilize parallel resources. In contrast to the existing data allocation schemes, our RDA—taking into account the parallelism requirements of I/O requests—enhances SSD lifetime by optimizing data locality without affecting parallelism.

In addition, Zhang et al. [31] proposed an SSD parallelism management and data placement scheme PLAN to improve the efficiency of GCs. Specifically, PLAN utilizes big superblocks and small superblocks to serve write requests with different sizes. Among them, a big superblock is a collection of blocks of all chips in a SSD that are written together, and a small superblock only collects blocks of 1/4 chips. Moreover, PLAN also places data with different lifetimes in different superblocks by evaluating the data lifetimes. Unfortunately, PLAN has only two superblocks (small superblock or big superblock) selected by the size of write requests and ignores the parallel requirements of read request. Unlike PLAN, our RDA scheme has a larger adjustable range of parallelism (from two to max parallelism) and can flexibly adjust the write parallelism of write requests by the size of read requests. Therefore, RDA reduces redundant data parallel distribution while ensuring the parallelism of read requests, which effectively improves GC efficiency and alleviates the impact of GCs on read requests.

III. Background

A. Flash-Based SSD Organization

A modern NAND flash-based SSD, as the mainstream storage drive, consists of multiple components. Fig. 1 sketches the internal structure of an SSD, which is composed of four components: 1) a host interface; 2) an SSD controller; 3) a DRAM cache; and 4) flash chips [32]. The host interface is responsible for the communication interaction between the host and the SSD. The SSD controller manages the flash chips through various functions (such as data allocation, GC, wear leveling, etc.), and DRAM stores the FTL mapping table and cached data. Flash chips are data storage elements, which are distributed on multiple channels to maintain channel-level parallelism in the SSD.

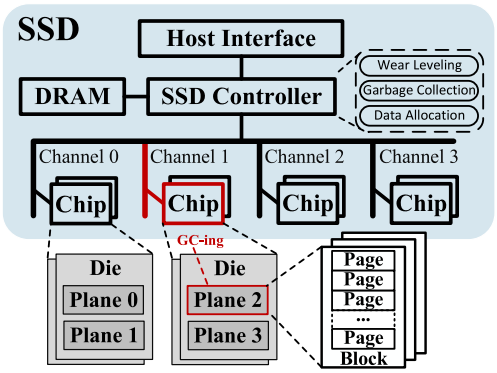

Fig. 1. SSD architecture and NAND flash chip.

Fig. 1 illustrates that a flash chip has multiple dies and each die consists of several planes, each of which contains thousands of blocks. Also, a block is made up of numerous pages. NAND flash memory has three operations: 1) read; 2) write; and 3) erase. Among them, read and write operations are in a unit of page, whereas an erase operation is performed in a unit of block. Furthermore, compared with 2-D NAND flash, the number of pages per block in 3-D NAND flash can reach up to 576, and the size of a single page is increased to 16 kB, thereby hiking up the capacity of a chip. For example, Samsung researchers developed a 128 Gb Vertical NAND flash memory in 2015 [33] and a 256 Gb 3b/Cell V-NAND flash memory in 2017 [34]. In the above two flash memory, the number of pages per block increased from 384 to 576, and the page size increased from 8 to 16 kB. Currently, pages per block in 3-D NAND flash are still increasing.

B. Parallelism and Garbage Collection

Parallelism: Fig. 1 reveals that the storage components of the SSD have multilevel storage units, which support multiple levels of parallelism, namely, the channel, chip, die, and plane levels. Specifically, different channels, chips, and dies can simultaneously serve reads and writes in SSDs [23]. In addition, multiplane operations developed by flash manufacturers provide plane-level parallelism for flash memory, and perform read/write operations on multiple planes in parallel [7]. In doing so, a plane becomes the finest-grained parallel unit executable in SSDs. Thus, the planes in different channels or chips simultaneously perform read/write operations to maximize read/write parallelism. Taking the simulated SSD in Section VI-A as an example, eight planes in the SSD can read/write eight pages in one read/write latency.

GC: When free pages are insufficient, GC is triggered by flash memory. A GC operation involves the following three steps: 1) find a victim block with the most invalid pages; 2) migrate the valid pages in the victim block to a free block; and 3) erase the victim block [11]. Among these

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:22:49 UTC from IEEE Xplore. Restrictions apply.

steps, step 2) consumes channel resources and blocks all requests on the GC-ing channel (see the red mark in Fig. 1). Especially, modern 3-D NAND flash-based SSDs suffer from "Big Block" problems [35]. More specifically, the number of valid pages per block grows sharply as the density of 3-D NAND flash increases. Consequently, when 3-D NAND flash memory invokes GC operations, the number of migrated valid pages is much larger than that of 2-D NAND. Additionally, because the dynamic allocation scheme allocates consecutive pages to different parallel units, the data fragmentation problem in the victim block becomes extremely severe, and step 2) will take up a lot of time on the GC-ing channel. Fortunately, the advanced command *Copyback* developed by the flash manufacturer compresses the data migration of step 2) in planes to release channel resources [8], [36]. However, it is noteworthy that GC-ing planes still impede requests assigned to planes.

In 3-D NAND flash-based SSDs, the high-storage density of 3-D NAND leads to a larger capacity per chip. Compared with 2-D NAND, 3-D NAND flash-based SSDs have fewer flash chips and fewer parallel units for the same storage capacity. Therefore, an important problem for 3-D NAND flash-based SSDs is how to fully utilize their limited parallel resources. Our research mainly focuses on the efficient management of limited parallel resources in 3-D NAND flash-based SSDs. The proposed RDA scheme improves the parallel resource utilization of 3-D NAND flash SSDs by reducing redundant data parallel distribution.

IV. Motivation

Modern NAND flash SSDs usually have two mainstream data allocation strategies: 1) *locality-friendly strategy* and 2) *parallel-friendly strategy*. These two strategies have their own advantages and disadvantages. Specifically, locality-friendly strategies place consecutive logical pages in the same block, which effectively avoid data fragmentation, thereby improving GCs and SSD lifetime. However, the locality-friendly strategies, due to extremely poor parallel performance, cannot support parallel reads of consecutive pages. Currently, numerous studies are focused on improving SSD parallelism through allocation strategies. These parallel-friendly strategies allocate consecutive logical pages to different parallel units, resulting in enhanced read/write parallelism and throughput. Unfortunately, the parallel-friendly strategies exacerbate GC and lifetime overhead. In particular, we reckon that redundant parallel distribution will increase the possibility of requests being blocked by GCs, which in turn downgrade the latency performance of SSDs.

In this section, we analyze and experimentally examine the impact of different parallel distributions on GC-blocked requests. Then, we investigate the size distribution of read requests in numerous workloads, thereby rationally adjusting the parallelism of consecutive pages in a write request.

A. Correlation Between Parallel Distribution and GC

To maximize the utilization of parallel units, dynamic allocation schemes assign consecutive pages to different planes through high-write parallelism. Such high-parallel distribution furnishes high-read parallelism for servicing subsequent read requests. Taking Fig. 1 as an example, there are a total of four planes across channels 0 and 1, and the four consecutive pages of a write request are placed on the four planes, respectively. When a read request accesses the four pages, the pages can be simultaneously accessed through parallel read operations. In this case, the parallel distribution of the pages significantly shortens the latency of the read request. Unfortunately, the parallel distribution exacerbates the probability of requests being blocked by GCs. Specifically, when a GC is provoked by plane 2, the requests of accessing plane 2 will be hindered by the GC. When accessing the four pages in the above example, the page on plane 2 only is accessed after completing the GC, and thus seriously weakens the read latency. More importantly, the four pages distributed on different planes have a high probability of being blocked by GCs, which is four times higher than those distributed on the same plane.

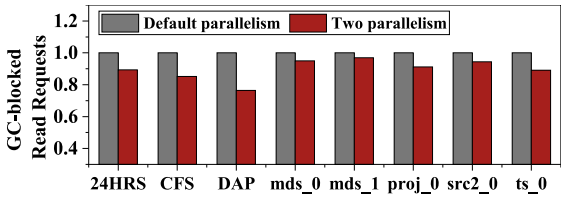

Fig. 2. GC-blocked read requests in different parallel distributions.

To gain insight into this issue, we conduct two experiments to navigate the impact of GCs on reads in different parallel distributions. Specifically, we adopt two SSDs with different settings in the experiments: 1) two parallelism and 2) default parallelism. The two parallelism setup means that the maximum parallel distribution of consecutive pages provided by the SSD is two, and the default parallelism case represents that a write request is written to the parallel units of an SSD by the highest-write parallelism. It worth noting that when the highest-write parallelism exceeds the maximum parallelism supported by the SSD, the request is written to the parallel units with the maximum parallelism. For example, when a write request with twelve pages is written to an SSD with eight parallel units (planes), the two parallelism allocates the pages to two planes, whereas the default parallelism assigns the pages on eight planes, respectively. We conduct experimental evaluations on the SSDsim [26] simulator with eight real workloads, and the experimental parameters are listed in Table II.

Fig. 2 shows the number of GC-blocked reads for the two parallelism and the default parallelism. To intuitively and clearly illustrate the experimental results, we normalize the experimental results of the two parallelism to the default parallelism. Fig. 2 unveils that compared with the default parallelism, the two parallelism significantly slashes the number of GC-blocked read requests. Also, although the default parallelism has a high-parallel distribution of consecutive pages and an excellent read parallelism, it exacerbates the possibility of read requests being blocked by GCs. Conversely, the low-parallel distribution (like the tow parallelism) alleviates the impact of GCs on read requests, but it cannot serve read requests with high parallelism. Therefore, we can dynamically

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:22:49 UTC from IEEE Xplore. Restrictions apply.

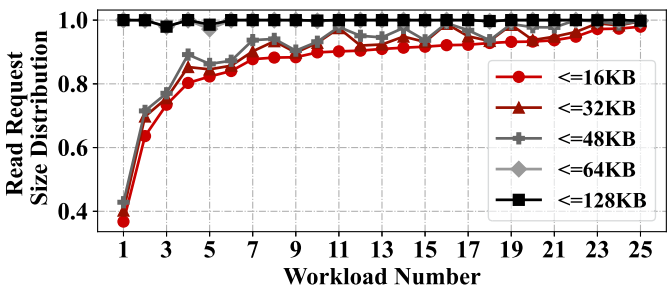

Fig. 3. Read request size distribution on different workloads.

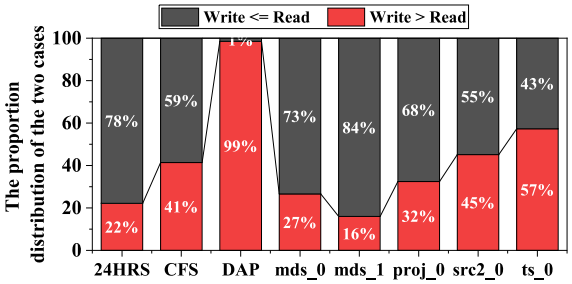

Fig. 4. Proportion distribution of the two cases on different workloads.

adjust the parallel distribution of continuous data by controlling the parallelism of a request: this approach strikes a balance between the impact of data parallel distribution and GCs on read latency.

B. Read Request Size

Recall that data parallel distribution can significantly improve the read parallelism of consecutive pages, but excessive parallel distribution aggravates the influence of GCs on requests. Furthermore, the parallel distribution of partial consecutive pages is redundant. The redundant parallel distribution is mainly caused by the inconsistent read and write sizes of consecutive pages. In some cases, consecutive pages are written to flash memory by high-write parallelism. However, only part of the pages is accessed when subsequent read requests are arrived. At this point, the parallel distribution of the remaining pages is redundant. The parallel distribution of the remaining pages does not improve the parallelism of read requests, but rather exacerbates data fragmentation. For example, the N consecutive pages of a large write request are placed on N parallel units, respectively. When a small read request accesses M ($M <= N$) pages among the consecutive pages, only M parallel units are required to provide parallel operations for the read request. At this point, the parallel distribution of the remaining $N - M$ pages on the other parallel units is redundant. It can be seen from the above example that the effective parallel distribution of consecutive pages is affected by the size of read requests. The high-parallel distribution of consecutive pages has a marginal performance enhancement for small reads.

To offer insights into read request sizes in real-world workloads, we investigate the proportion of read requests with various sizes in twenty-five real-world workloads. The real-world workloads examined in our experiments are obtained from the MSR trace [37] and Microsoft Production Server [38]. In particular, according to the request size, we divide read requests in the workloads into five cases: 1) <=16 kB; 2) <=32 kB; 3) <=48 kB; 4) <=64 kB; and 5) <=128 kB. Fig. 3 reveals the distribution ratio of the five cases under the twenty-five workloads. On the x-axis, numerical nodes, each of which represents a workload, are sorted by an increasing order of their percentage of requests (<=16 kB). Fig. 3 shows that more than 80% of read requests are smaller than 16 kB in 88% of the tested workloads, and almost all read requests are smaller than 128 kB. Such findings indicate that consecutive pages written by write requests are usually retrieved by small reads. In this case, maximizing parallel distribution for consecutive pages is unnecessary. Thus, we ought to pay heed to the parallelism required by read requests, thereby ensuring read parallelism and alleviating adverse repercussion of GCs on read operations.

Furthermore, we also experimentally investigate the size of each read request and the write size of the accessed data, thereby evaluating the access status of data written by write requests. Specifically, according to the read request size and the write size of the accessed data, we divide the relationship between read requests and write sizes into two cases: 1) Write > Read and 2) Write <= Read. Write > Read indicates that a portion of data written by a write request is accessed by a small read request. Write <= Read signifies that the data of a write request is fully accessed by a large read request.

Fig. 4 illustrates the proportion distribution of the two cases under the eight real-world workloads. We can observe from Fig. 4 that there are a large number of the case Write > Read on the eight workloads. In particular, the proportion of Write > Read reaches the astonishing 99% on DAP. On average, 42.38% of read requests in each workload only access a portion of data written by a write request. The above experiments reveal that massive read requests only access a portion of data written by a write request. Therefore, we can flexibly adjust the parallelism of write requests through the size of read requests, thereby reducing redundant data parallel distribution while ensuring the read parallelism of SSDs.

Furthermore, placing the pages of write requests on different parallel units can fully utilize the parallel units to reduce the write latency and improve write throughput of SSDs. However, the parallelism of write requests does not affect the write parallelism and write throughput of SSDs. Specifically, since DRAM of an SSD caches the pages of write requests, the cached pages of different write requests can be allocated to different planes simultaneously, thus ensuring the write parallelism of the SSD. Therefore, adjusting the parallelism of write requests does not affect the write parallelism and write throughput of SSDs.

V. RDA Scheme

To benefit from limited parallel resources in 3-D NAND flash-based SSDs, we propose a novel read-request driven adaptive allocation scheme - RDA - to curb unnecessary data

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:22:49 UTC from IEEE Xplore. Restrictions apply.

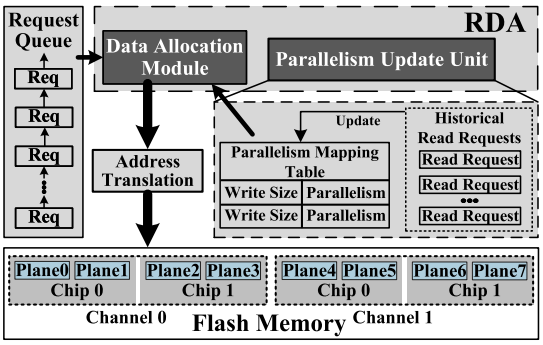

Fig. 5. Overview of RDA for 3-D NAND flash.

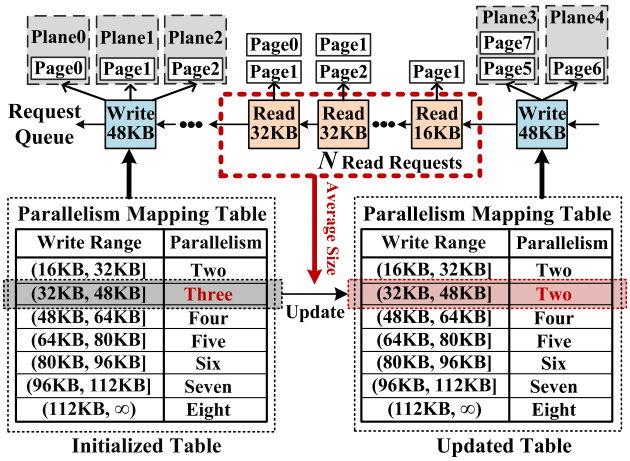

Fig. 6. Parallelism update unit workflow.

parallel distribution and improve data locality, thereby revamping the overall SSD performance. In RDA, the parallelism of write requests is determined by the size of read requests. the goal of RDA is to minimize the redundant data parallel distribution without affecting the parallelism of read requests. RDA adjusts the parallelism of write requests according to the size of read requests, and thus the parallel distribution of the write requests meets the parallel requirements of the read requests.

A. System Overview of RDA

Although 3-D NAND flash-based SSDs pushes up the capacity of a single plane, the parallel units inside the SSD are still extremely limited. Moreover, since the redundant parallel distribution deteriorate data fragmentation and the impact of GCs, SSDs suffer from an excessive waste of inadequate parallel resources. To improve the overall performance of 3-D NAND flash-based SSDs, we develop a read-request driven adaptive allocation scheme RDA, which curtails redundant parallel distribution of consecutive pages. Unlike the existing techniques, RDA enhances data locality while promoting read/write parallelism.

RDA strives to optimize the I/O performance of SSDs by adjusting data allocation during writing logical pages to flash chips. Specifically, RDA implements a parallelism mapping table based on historical read requests, and thus assigns the specified parallelism for write requests with different sizes. Upon the arrival of a write in flash chips, RDA allocates the pages of the write to multiple planes according to the specified parallelism and the pages' logical distance, which reduces the redundant parallel distribution and assures read parallelism.

Fig. 5 depicts the overall system design of the proposed RDA, which is deployed in the SSD controller orchestrating data allocation. The overarching goal of RDA is to optimize read performance through proper data allocation scheduling. To this end, RDA employs historical read requests to adjust the parallel distribution of consecutive pages. Specifically, the RDA scheme is implemented by the two core modules, namely, the parallelism update unit and the data allocation module. The parallelism update unit is responsible for updating the parallelism mapping table by the size of historical read requests, and allocating the specified parallelism for write requests with different sizes. The data allocation module is in charge of assigning the pages of write requests to different planes by the specified parallelism and page distance. Below, we elaborate on the parallelism update unit followed by the data allocation module in Sections V-B and V-C, respectively.

B. Parallelism Update Unit

The parallelism update unit, as the core piece of our RDA, dynamically updates the parallelism of write requests with the continuous access of read requests. RDA implements a parallel mapping table in the parallelism update unit to record the parallelism of write requests. Before introducing the parallel update unit, we first establish the parallel mapping table—a data structure dedicated to storing the parallelism corresponding to writes with various sizes. More prosaically, the parallel mapping table divides a write request into $P-1$ write ranges according to various parallelisms (form two to P) supported by an SSD, where P is the maximum number of parallel units (planes) in an SSD. Write requests per range have the same maximum parallelism. When initializing the parallelism mapping table, the parallelism of a range is set to the maximum parallelism of the range. In RDA, we do not adopt to assign a corresponding parallelism to each write request at a fine-grained level because the fine-grained mapping tends to consume considerable storage space. Therefore, to pursue the generalization and low overhead of the scheme, our RDA utilizes the average size of read requests to adjust the parallelism of a specific write range. Since the parallelism mapping table only stores the parallelism of $P-1$ ranges, and the value of P is extremely limited, the space overhead of the mapping table is negligible.

Fig. 6 illustrates a concrete example, where the mapping table is implemented in an SSD with eight planes ($P = 8$), and the page size of the SSD is 16 kB. The parallelism mapping table registers the parallelism corresponding to the seven ranges of writes: 1) (16 kB, 32 kB]; 2) (32 kB, 48 kB]; 3) (48 kB, 64 kB]; 4) (64 kB, 80 kB]; 5) (80 kB, 96 kB]; 6) (96 kB, 112 kB]; and 7) (112 kB, ∞). It is noteworthy that the upper limit of the parallelism is the total number of planes in SSDs. For instance, in the initialized table plotted in Fig. 6,

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:22:49 UTC from IEEE Xplore. Restrictions apply.

the initial parallelism of (32 kB, 48 kB] and (112 kB, ∞) are the maximum parallelisms of three and eight, respectively.

The parallelism update unit amends the parallelism of different ranges by historical read requests, aiming to mitigate redundant parallel distribution. In the parallelism update unit, the pages of a write request also correspond to the request's range. The parallelism update unit separately tracks reads that access the pages of diverse ranges. When N read requests access the pages of a range, the unit calculates the average size of the read requests, then updates the parallelism of the range in the parallelism mapping table by the average size. In order to match the parallelism requirements of reads, the unit only logs the size of recent N read requests in each range. When the number of reads for a range is less than N, the unit stops updating its parallelism. According to experimental evaluation, RDA has the most stable performance improvement when N is configured to 10 (see Section VI-B). As a result, we set the value of N to 10 in this study.

Fig. 6 shows the workflow of the parallelism update unit. First, according to the initialized parallelism mapping table, a 48-kB write request, respectively, places three pages 1, 2, and 3 on three planes because the parallelism of (32 kB, 48 kB] is three. Then, N read requests pay visit to pages 0, 1, and 2, and the average size is anywhere between 16 and 32 kB. The average size means that these read requests generally require two parallel units to provide parallel read operations, and the parallelism of three is redundant for the write requests of (32 kB, 48 kB]. Thus, the parallel mapping table updates the parallelism of (32 kB, 48 kB] to two. The subsequent 48-kB write request allots pages to two planes by the updated table. At this point, the 48-kB write request cuts down a redundant parallel unit.

Furthermore, if read requests with large size access the data of a range, the unit will increase the parallelism of the range and thus effectively ensures the high-parallelism requirements of the large read requests. In particular, when the average size of read requests equals or exceeds its corresponding write range, RDA provides the maximum parallelism for write requests in the range. Taking the updated table in Fig. 6 as an example, since the page size is 16 kB, the maximum parallelism of (32 kB, 48 kB] is three. If the 48-kB write data is accessed by subsequent read requests with an average size of >=48 kB, the unit will change the parallelism of (32 kB, 48 kB] from two to three. In short, the parallelism update unit dynamically increases or decreases the parallelism of a specified write range by the average size of reads. Therefore, the parallelism update unit effectively trims unnecessary parallel distribution while maintaining the parallelism of read requests.

C. Data Allocation Module

The parallelism update unit in RDA employs the parallelism mapping table to manipulate the parallelism of writes with various sizes. Then, the data allocation module designates plane addresses for the pages of a write by the request's parallelism, and writes the pages to planes, thereby warranting the parallel distribution of consecutive pages. The implementation details of the data allocation module in RDA are articulated as follows.

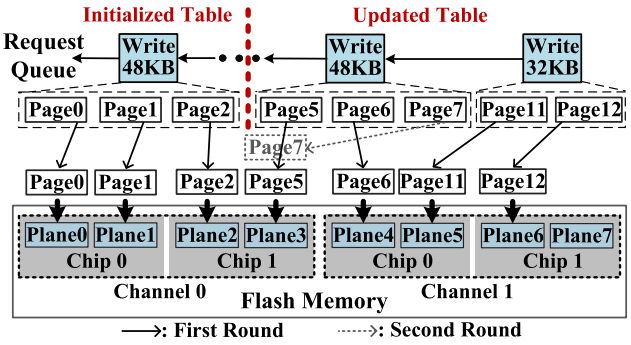

Fig. 7. Data allocation module workflow.

The data allocation module utilizes a loop pointer to determine the planes serving write requests. Specifically, when initializing an SSD, the pointer points to the first plane of the first channel in the SSD and obtains its plane address. Then, when a page is written to the plane referenced by the pointer, the loop pointer switches to the next plane. After the pointer has traversed all plane addresses, it redirects to the first plane address and starts a new round of loops. With the loop pointer in place, RDA simultaneously writes multiple pages to different parallel units of an SSD, and write throughput is naturally bolstered. Additionally, the data allocation module allocates plane addresses to logical pages and then writes the pages to the specified planes according to the loop pointer. It is noteworthy that the module does not limit the plane address of the first page of a write request. More specifically, the first page of a write request does not allocate the specified plane address; rather, the page is directly written to the plane referenced by the pointer. After confirming the plane address of the first page, the module adopts (1) to assign the plane addresses of the other pages by the request's parallelism

$$X_{\text{plane}} = F_{\text{plane}} + (X_{\text{lpn}} - F_{\text{lpn}}) \mod P \tag{1}$$

where P is the parallelism of the request. X_{plane} and X_{lpn}, respectively, represent the plane address and logical page number of page X in the request, and F_{plane} and F_{lpn} are the plane address and logical page number of the first page in the request, respectively. $X_{\text{lpn}} - F_{\text{lpn}}$ is the distance (the number of logical pages) between page X and the first page. According to (1), the data allocation module guarantees that among pages written by the request, any P consecutive pages can be read in parallel. When the pointer directs to a plane, the first page of a request or the page with the plane's address is written to the plane. The time complexity of (1) is $O(1)$: the time overhead of the data allocation module in RDA is imperceptible. Moreover, as SSDs employ DRAM to cache the pages of write requests, the module adjusts the write order of pages and writing back the pages are prioritized corresponding to the plane indicated by the loop pointer.

Let us describe the data allocation module of RDA with an example illustrated in Fig. 7, where an SSD with eight planes serves three write requests. The parallelism of the first request

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:22:49 UTC from IEEE Xplore. Restrictions apply.

and the last two requests are regulated by the initialized table and the updated table in Fig. 6, respectively. In the first round of writing pages, the loop pointer starts from plane 0 and points to eight planes in turn. Pages 0, 1, and 2 of the first request are first written to planes 0, 1, and 2, respectively, (parallelism = three). Next, page 5 of the second request is written to plane 3, and pages 6 and 7 are, respectively, allocated to planes 4 and 3 (parallelism = two). To this end, since the loop pointer is shifted to the next plane after writing a page, page 6 is written to plane 4, whereas the writing of page 7 is deferred to the second round. Then, pages 11 and 12 of the third request are located on planes 5 and 6, respectively. After writing eight pages to the eight planes, the SSD starts the second round of writing pages, and the pointer restarts from plane 0. In the second round, when the pointer points to plane 3, page 7 will be written to the plane. In this case, any two consecutive pages of the second request can be read in parallel to warrant read parallelism.

In summary, RDA adjusts the parallelism of write requests through historical read requests, and the parallelism is incorporated to allot the plane address of pages. In doing so, RDA effectively curbs the redundant parallel distribution of consecutive pages and the SSD performance is overhauled by alleviating read performance degradation incurred by the redundant parallel distribution.

D. Discussion

1) Space Overhead: In RDA, the parallelism mapping table stores the parallelism for requests within a range rather than a certain request or page. Taking the SSD with eight parallel units in Fig. 7 as an example, the mapping table records parallelism information for seven write ranges. Obviously, the space overhead of the mapping table is negligible. In addition, when a write request writes logical pages to the flash memory, RDA assigns the logical pages to different physical pages by the parallelism mapping table and records the write range corresponding to the write request in the metadata of the physical pages. It is worth noting that RDA only records the write range information of the write request corresponding to the pages. Subsequently, when RDA accesses the data of the physical pages, it can simultaneously read the logical address and write range corresponding to the physical page. Therefore, RDA stores the write range information in flash memory instead of FTL on DRAM. The main reason for this storage method is to minimize the impact of the space overhead on SSD performance. Since the DRAM space is extremely limited and is important for the performance improvement of SSDs, recording more information in FTL may affect SSD performance. In contrast, flash memory has a larger storage space, and the impact of adding the range information on the total storage space of flash memory is negligible. For instance, a 1 TB SSD contains eight parallel units (i.e., eight planes), and the page size is 16 kB [34]. In RDA, the eight parallel units means that physical pages have eight different write range types, and each page requires 3 bits of space to store its range type. The SSD contains 67108864 physical pages, and thus it requires 201326592 bits (24 MB) of space to store the write range information for all pages. Obviously, compared with a 1 TB SSD, the space overhead of 24 MB is negligible.

2) Impact of Write Cache: On a DRAM-embedded SSD, write-requested pages are cached in the write cache of DRAM and written back to flash chips sequentially. However, except for the first page of a write request, the remaining pages are allocated by RDA to the specified plane addresses. Therefore, when writing a cached page to flash memory, the page's plane addresses allocated by RDA may not be the plane pointed to by the pointer. At this point, RDA will search for a suitable cached page in the write cache and write it to the plane pointed by the pointer. The suitable cached page is the first page of a write request whose plane address is not restricted, or the page that is allocated to the plane. Therefore, the write cache in DRAM plays an important role in RDA. Fortunately, compared with the limited parallel units, the write cache holds massive pages for RDA to allocate, which effectively guarantees the RDA performance. Taking an SSD with eight planes and a 128-MB write cache as an example, the write cache can store 8192 pages, and the pages will be allocated to the eight planes. Therefore, each plane has sufficient candidate cached pages to find a suitable cached page. It worth nothing that the impact of cache size on optimization schemes involving caching is usually unavoidable, and a larger cache can improve the performance of most schemes. Fortunately, with the increasing capacity of mainstream SSDs, the DRAM embedded in the SSDs also has greater storage capacity. Since the large-capacity cache can provide more cached pages, we believe that RDA will have a promising performance in mainstream SSDs with larger DRAM capacity.

3) Preemptive GC: Preemptive GC is common solution in SSDs to alleviate the impact of GCs on requests. Specifically, preemptive GC prioritizes serving read/write requests by interrupting GC operations, and then resumes the interrupted GC operations after completing the requests. However, it cannot completely eliminate the impact of GCs on read/write requests and seriously affects the write performance of SSDs. Specifically, SSDs with preemptive GC have the following problems. First, preempting GC interrupts data migration operations, while erase operations during GCs still block read/write requests. Second, if a GC is continuously interrupted by incoming requests, the GC-interrupted flash chip cannot erase invalid data and generate free space through the GC. In this case, write requests will be postponed indefinitely until the chip has free space after completing the GC. Third, to prevent the above case, SSDs usually maintain two GC thresholds: 1) a soft GC threshold and 2) a hard GC threshold (soft > hard). If the remaining space of a flash chip is less than the soft threshold, the flash chip triggers a preemptive GC operation. When the remaining space is less than the hard threshold, the flash chip will perform an uninterruptible GC to prevent write requests from being postponed indefinitely. At this point, requests accessing the chip are blocked during the uninterrupted GC. In short, preemptive GC cannot completely eliminate the impact of GCs on read requests. Fortunately, our RDA strategy does not suffer from the above problems. RDA mitigates the impact of GCs on read requests by reducing

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:22:49 UTC from IEEE Xplore. Restrictions apply.

redundant data parallel distribution. Moreover, since the purpose of RDA is to reduce the probability of read requests being blocked by GCs, RDA does not conflict with preemptive GC, and even can avoid the impact of uninterruptible GCs on read requests. Therefore, RDA cannot be replaced by preemptive GC and can effectively improve the performance of SSDs.

4) Multiple Read Requests: A modern SSD can fetch and schedule I/O requests from multi I/O command queues, and inserts the requests into a device-level queue of the SSD. Then, the requests in the device-level queue are serviced by DRAM. When a read request in the device-level queue misses DRAM, SSDs will divide the read request into multiple subread requests based on the physical page size. Unlike write requests, flash chips accessed by the read request are determined, and each subread requests access the specified flash chip. A GC-blocked chip will inevitably block read requests accessing data within the chip. While a subread request accesses a GC-blocked chip, it will seriously affect the completion time of its corresponding read request. Fortunately, RDA curtails the possibility of read requests being blocked by GCs through degrading the parallel allocation of consecutive pages, which effectively reduces the overall read latency of read requests.

For example, in an SSD with four chips 0, 1, 2, and 3 (each chip has only one plane), there are two four-page write requests W1 and W2, and two read requests R1 and R2 access the pages of W1 and W2, respectively. In the traditional allocation scheme, W1 writes its four pages to the four chips 0, 1, 2, and 3 with maximum parallelism, respectively. Likewise, W2 also writes to the four chips. Then, R1 and R2 access the pages of W1 and W2 on the four chips. At this time, all chips are accessed by subread requests from R1 and R2, and each chip has two subread requests. When chip 0 is blocked by a GC, the subread requests of R1 and R2 on chip 0 are blocked. R1 and R2 cannot be finished until the blocked subrequests are completed. Thus, in the traditional allocation scheme, both R1 and R2 are blocked, and the completion time of the two requests is significantly increased. Conversely, RDA can reduce read requests affected by GCs. It is assumed that RDA adjusts the parallelism corresponding to four-page write requests to two. In RDA, W1 writes four pages to chips 0 and 1 according to the parallelism two, and each chip stores two pages. Similarly, W2 is written to chips 2 and 3. Then, R1 accesses the pages of W1 on chips 0 and 1, while R2 accesses chips 2 and 3. At this point, each chip has two subread requests, and all chips are also accessed by R1 and R2. When chip 0 is blocked, the subread requests of R1 on chip 0 are blocked. Fortunately, R2 accessing chips 2 and 3 will not be blocked by the GC. Therefore, unlike the traditional allocation schemes, RDA only blocks one read request in the above example, thus effectively improving read performance.

5) Write Ranges and Average Size: RDA distributes write requests to different write ranges by the request size and counts the size of read requests accessing all written data in each range. Thus, RDA can fully fetch the parallel requirement of all read requests of a range. Additionally, RDA adopts the average size of read requests to reasonably adjust the parallelism of a range, which can comprehensively consider the parallelism requirements of all requests within the range. In a nutshell, using the size-averaging method, RDA can balance the read parallel requirements of different written data in a range as much as possible, thereby slightly reducing the parallelism of write requests while significantly improving read performance.

TABLE I
SPACE OVERHEAD OF FINE-GRAINED REQUEST MAPPING

Trace	Total Write	Total write addresses	Space Overhead
24HRS	4293405	2903065	11612KB
CFS	1173051	755227	3021KB
DAP	476079	92938	372KB
mds_0	1067061	33141	133KB
mds_1	116676	34248	137KB
proj_0	3697143	216268	865KB
src2_0	1381085	56062	224KB
ts_0	1485042	55669	223KB

Furthermore, there are two main reasons why RDA adjusts write parallelism through different write ranges rather than different write requests. First, compared with recording write requests, the space overhead generated by recording write ranges and their parallelism is extremely limited. In contrast, recording the parallelism of different write data will generate tremendous space overhead. Second, the cost-effectiveness of recording write requests is extremely low. Since most of the data in an SSD is cold data that will not be written repeatedly, it is useless for the mapping table to record cold write requests. Therefore, RDA employs write ranges to adjust the parallelism of write requests, which can directly obtain the parallelism requirements of read requests, thereby diminishing redundant data parallel distribution. Below, we analyze the SSD space overhead in coarse-grained range mapping and fine-grained request mapping, thereby revealing the relationship between granularity size, overhead, and performance.

In RDA, the parallelism mapping table can be stored in the form of an array in terms of data structure, which is usually represented by an array of P-1 elements. In the parallelism mapping table with the coarse-grained range mapping, P is the number of parallel units in an SSD. P-1 is the number of write ranges, and each element records the write parallelism corresponding to each range. Taking a SSD with eight parallel units as an example (see Fig. 6), the parallelism mapping table records the parallelism of the seven write ranges through an int array *Array*[7] with seven elements. *Array*[1] represents the parallelism corresponding to the write range (32 kB, 48 kB]. At this time, RDA with the coarse-grained range mapping generates 28 Byte (7×4 Byte) of space overhead.

However, if RDA adopts the fine-grained request mapping, the parallelism mapping table of RDA must record the first logical address of a write request and its corresponding parallelism. At this point, P-1 is the number of write requests with different write logical addresses. Table I shows the space overhead of the fine-grained request mapping for different workloads. In Table I, total writes are the total number of write requests in a workload; Total write addresses is P-1, which is the number of write requests with different

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:22:49 UTC from IEEE Xplore. Restrictions apply.

TABLE II
SIMULATION PARAMETERS

SSD parameters	
(SSD Capacity, Channels)	(128GB, 2)
(Chips, DRAM capacity)	(4, 128MB)
(Die, Planes, Blocks, Pages)	(1, 2, 1888, 576)
(Page size, Cell density)	(16KB, TLC)
(Over provision, GC trigger)	(25%, 5%)
Latencies	
Page Program	900μs
Page read (LSB, CSB, MSB)	(90μs, 120μs, 180μs)
Block earse	10000μs

TABLE III
I/O CHARACTERISTICS OF EIGHT WORKLOADS

Trace	Total I/Os	Write Ratio	Avg_Write_Size	Avg_Read_Size
24HRS	5250996	0.817	23.32	212.22
CFS	4477510	0.262	25.24	17.45
DAP	1086729	0.438	194.26	124.23
mds_0	1211034	0.881	14.47	47.42
mds_1	1637711	0.071	27.68	119.15
proj_0	4224524	0.875	81.41	35.67
src2_0	1557814	0.887	14.19	16.21
ts_0	1801734	0.824	16.01	27.36

write logical addresses, and is also the number of mapping relationships in the mapping table; Space overhead is the space overhead generated by the parallelism mapping table with the fine-grained request mapping. It can be seen from Table I that the overhead of the fine-grained range mapping is much higher than that of the coarse-grained range mapping. More importantly, the space overhead of fine-grained request mapping will generate large additional data writes, because the mapping table occupies the storage space that holds the write request data. The additional data writes will trigger GC operations, thus seriously affecting SSD performance.

VI. EVALUATION

A. Experimental Setups

To evaluate the effectiveness of our RDA, we implement RDA in a well-known SSD simulator, SSDsim [26]. SSDsim, as a trace-based public simulator, seamlessly integrates distinctly defined structures, such as a buffer management, a request allocation layer, an FTL layer, and a hardware module layer. Meanwhile, a wide adoption of SSDsim is found in a pile of studies investigating the performance of SSD-based storage systems. The detailed SSD parameters adopted by our implementation are tabulated in Table II. Specifically, we simulate a 128 GB capacity SSD, and the SSD has four TLC 3-D NAND chips. The configuration parameters of our experiments are mainly derived from [5], in which the alternative allocation schemes are delineated.

Furthermore, our simulated SSD employs two-step programming to write data. We must clarify that RDA is not affected by programming methods. Currently, two-step programming and one-shot programming are currently the mainstream programming methods, which are applied to floating gate flash memory [39] and charge trap flash memory [4], respectively. The difference between the two programming methods lies in how the plane writes the data of logical pages to physical pages. However, as a data allocation scheme, the main role of RDA is to distribute logical pages to different planes. Therefore, regardless of the programming methods, RDA can still use (1) to place consecutive pages on different parallel units according to the parallel requirements of read requests, thus fully utilizing the parallel resources of SSDs.

Workloads: We evaluate the performance of RDA driven by the eight realistic workloads. Table III indicates that the workloads have various write ratios, ranging from 7.1% to 88.7%. Among the workloads, workloads 24*HRS*, CFS, and DAP are obtained from Microsoft Production Server [38]. The other five workloads come from the MSR [37] repository. In Table III, *Avg_Write_Size* and *Avg_Read_Size* signify the average size of write and read requests by the number of logical sectors, respectively. Additionally, among these eight workloads, 24*HRS*, *mds*_0, *proj*_0, *src*2_0, and *ts*_0 are write-intensive workloads, while the other three workloads exhibit read-intensive behaviors. Because RDA optimizes read performance by allocating consecutive data to multiple parallel units, the write-intensive workloads facilitate coordinated support for evaluating the improvement of read performance through the data allocation of RDA.

Comparison Schemes: We compare RDA with four state-of-art solutions: 1) *dynamic allocation (Baseline)*; 2) AMP [24]; 3) SOML [5]; and 4) *Preemptive GC* [40]. Among the alternatives, the dynamic allocation scheme is serving as a baseline solution. For fair comparison purposes, we normalize the experimental data of all the tested schemes against the baseline scheme. Since the eight workloads have different access characteristics, the performance improvement of the above schemes may not be of the same order of magnitude under the workloads. To clearly demonstrate the performance improvement of different schemes, we adopt the normalization approach to comparing the schemes against the baseline. The four alternative solutions are summarized as follows.

1) *Dynamic Allocation:* The dynamic allocation scheme sequentially writes pages to multiple planes according to the cache queue. The scheme effectively maintains high-read/write parallelism of SSDs, but the conflict between GCs and parallelism is ignored.
2) *AMP:* AMP adjusts the order of a request queue based on the request characteristics. AMP avoids the access conflict between I/O and GCs, shortening the overall latency of SSDs. However, AMP falls short on addressing the adverse repercussion of GCs on parallel operations.
3) *SOML:* SOML is fine-grained parallel read operations within a plane. SOML combines multiple small read requests by adding extra circuits, thus improving read throughput. Unfortunately, the extra circuits introduce additional space and time overhead.
4) *Preemptive GC:* Preemptive GC can prioritize I/O requests from a host by preempting the ongoing GCs, thereby significantly reducing I/O latency. Regrettably,

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:22:49 UTC from IEEE Xplore. Restrictions apply.

Preemptive GC still does not fully address the impact of GCs on requests.

B. Performance Evaluation

We evaluate the performance and durability of RDA with five metrics, namely, 1) GC-blocked read requests; 2) parallel read counts; 3) GC counts; 4) write amplification; and 5) average response time. The first two metrics assess the impact of RDA on read requests, and the latter three metrics evaluate the performance of RDA in terms of SSD lifetime and latency. The rationale behind the selection of these metrics is given as follows.

1) GC-blocked read requests reveal the number of read request blocked by GC operations, thus analyzing the optimization of our RDA on GC-blocked I/Os.
2) Parallel read count is a significant indicator for evaluating the impact of different schemes on read parallelism.
3) In NAND flash memory, each block has a constraint on the maximum number of erase operations. As each GC involves a block erase operation, GC counts intuitively express SSD lifetime.
4) Write amplification indicates that the data actually written to flash memory is greater than the write data issued by a host. Write amplification factor (WAF) represents the ratio of the actual amount of data written on flash memory to the amount of data written by a host. The WAF value reflects the impact of SSD optimization schemes on SSD performance and lifetime.
5) Finally, average response time indicates the average time spent by an NAND flash-based SSD in processing a request: the response time is mainly divided into read response time and write response time.

Furthermore, recognizing that the N value have potential influence on the performance of RDA, we discuss the choice of N value in RDA at the end of this section.

1) GC-Blocked Read Requests: In this section, we pay attention to GC-blocked read requests for evaluating the scheme performance. GC operations will block the access of read requests, which seriously affect read latency. An increasing number of GC-blocked requests exacerbates SSD performance degradation. In our experiment, we compare our RDA with the other three schemes in terms of GC-blocked read requests under the eight workloads. It is worth noting that since preemptive GC interrupts GCs and prioritizes read requests, the interrupted GCs will not block read requests. Therefore, we do not evaluate preemptive GC in this section.

Fig. 8 unveils the number of GC-blocked read requests of different schemes handling the eight workloads. We can observe from Fig. 8 that our RDA has a leading edge over the other schemes regardless of the workload conditions. For example, RDA significantly reduces the GC-blocked read requests of the baseline scheme by 12.1%, 13.1%, 62.1%, 7.4%, 5.1%, 13.1%, 18.5%, and 21% under the eight traces, respectively. Moreover, compared with AMP and SOML, RDA slashes the number of GC-blocked requests by averages of 14.9% and 26.4%, respectively. These results demonstrate that RDA is superior to the other alternative schemes. Although

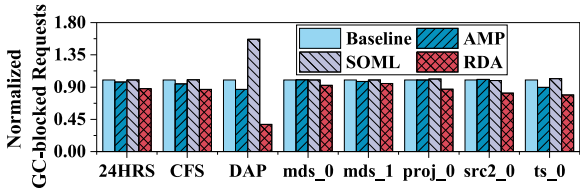

Fig. 8. GC-blocked read requests of four schemes.

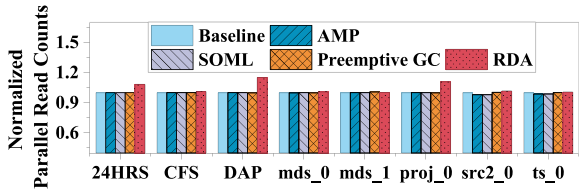

Fig. 9. Parallel read counts of different schemes under eight traces.

AMP alleviates the negative impact of GCs on requests, AMP is unable to fundamentally avert the requests to access GC-ing chips. Unlike the existing schemes, our RDA technique, suppressing the redundant parallel distribution of consecutive pages, is adroit at diminishing the probability of requests being impeded by GCs. RDA has a clear edge over the alternatives in boosting the latency performance of SSDs.

2) Parallel Read: In this group of experiments, we examine parallel read operations to evaluate the performance of SSD optimization schemes. The parallel read operation is a dominant indicator for evaluating the impact of various strategies on read parallelism. A high number of parallel read counts entails the fast performance of an entire storage system.

Fig. 9 unfolds the parallel read counts of the five schemes driven by the eight traces. In Fig. 9, RDA claims a winning position among its peers with respect to parallel read counts. In particular, compared with the baseline scheme, our RDA increases the number of parallel read operations by 8.2%, 15.2%, and 10.9% under 24*HRS*, DAP, and *proj*_0, respectively. Under the other workloads, RDA presents a similar performance to those of the baseline scheme, but RDA noticeably outperforms the other schemes. Since AMP, SOML, and preemptive GC ignore the impact of parallel distribution of consecutive pages on read parallelism, they have similar performance in terms of parallel read counts. Recall that RDA allocates consecutive pages to different planes by the size of read requests, so parallel reads of the pages are maintained: RDA exhibits superb performance in terms of read parallelism.

3) GC Counts: NAND cells have limited erase operations. Unfortunately, each GC triggers an erase operation and reduces the reliability of cells. Thus, GC counts are an effective indicator for evaluating SSD lifetime. For this reason, we investigate the impact of RDA on SSD lifetime by comparing RDA with the other four schemes in terms of GC counts. The experimental results are shown in Fig. 10.

Fig. 10 unveils that compared with the other four schemes, RDA is the most conducive approach to prolonging the life of SSDs (a low-GC count signifies a long SSD lifetime). For instance, RDA slashes the number of GC operations of

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:22:49 UTC from IEEE Xplore. Restrictions apply.

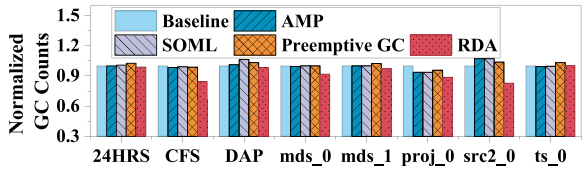

Fig. 10. GC counts of different schemes under eight traces.

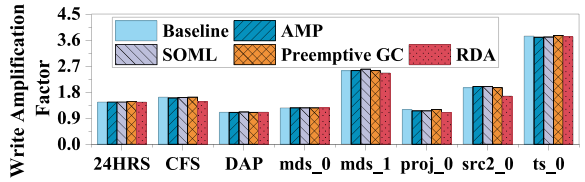

Fig. 11. Write amplification of different schemes with eight traces.

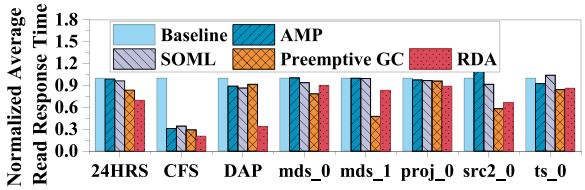

Fig. 12. Average read response time with different schemes under eight traces.

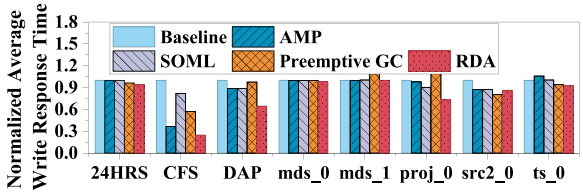

Fig. 13. Average write response time of different schemes under eight traces.

the baseline, AMP, and SOML, preemptive GC by averages of 7.2%, 7.1%, 8.0%, and 8.3%, respectively. Since the comparison schemes ignore the data fragmentation problem exacerbated by the parallel distribution of consecutive pages, these schemes cannot effectively alleviate the surge of GC operations caused by data fragmentation. Conversely, our RDA takes full account of the impact of parallel distribution on GCs while protecting read parallelism. RDA alleviates the data fragmentation problem through pruning the redundant parallel distribution, thereby reducing the number of GC operations.

4) Write Amplification: We evaluate the write amplification of the five alternative schemes on the eight real-world workloads. Write amplification plays an important role in evaluating SSD lifetime: a small write amplification indicates long SSD lifetime.

Fig. 11 shows the WAF of the five different schemes. The experimental results demonstrate that RDA significantly lowers the write amplification of the other schemes under the real-world workloads. Especially, compared with the baseline scheme, RDA reduces write amplification under CFS and *src*2_0 by 9.6% and 15.2%, respectively. Moreover, under the other workloads, RDA also marginally drops the write amplification ratio of the other SSD optimization schemes. AMP, SOML, and preemptive GC present similar performance to the baseline scheme on write amplification because of ignoring the influence of data locality on SSD lifetime. In a nutshell, these experimental results reveal that RDA offers the lowest-write amplification among the five SSD optimization schemes because RDA is adept at improving data locality and suppressing GCs. The reduction of GCs remarkably mitigates write amplification.

5) Average Response Time: Now we are at the position of evaluating the average response time of RDA. Response time, as an important indicator for evaluating SSD performance, is mainly divided into read response time and write response time. As a result, we conduct experiments to compare RDA with the other four schemes in terms of average read/write response time.

Fig. 12, illustrating the average read response time of the five schemes under the eight workloads, unveils that RDA outperforms the other three schemes Baseline, AMP, and SOML under the eight workloads. For instance, compared with the baseline scheme, RDA improves its read response time under the eight traces by 30.3%, 79.4%, 66.5%, 9.8%, 16.6%, 10.8%, 33.7%, and 13.7%, respectively. Also, RDA boosts the read performance of AMP and SOML by an average of 21.9%. Moreover, while preemptive GC exhibits the best performance under *mds*_0, *mds*_1, *src*2_0, and *ts*_0, it does not perform as well as RDA under the other workloads. The reason for the above experimental results is as follows. Under workloads with small write data (e.g., *mds*_0, *mds*_1, etc.), preemptive GC significantly improves read performance by interrupting GC operations. However, when SSDs will trigger uninterruptible GCs under workloads with large write data (e.g., 24*HRS*, CFS, etc.), preemptive GC cannot prioritize read requests. Fortunately, RDA is not affected by uninterruptible GCs, which reduces data parallel distribution to fundamentally alleviate the possibility of read requests being blocked by GCs, thus significantly improving read performance.

Furthermore, compared with SOML, AMP adjusts the access mode of request queues and reduces the data writing of GC-ing flash chips, thus affecting GCs and WAF. Therefore, AMP has better performance than SOML under some workloads in terms of GC-blocked read requests, GC counts and WAF. However, SOML is a read latency optimization scheme that focuses on reducing read latency. SOML merges multiple subpage-level read requests into a page-level read request, and thus reduces read operations and latency. Accordingly, SOML provides better-read performance than AMP.

Fig. 13 shows the average write response time of the five schemes. Unsurprisingly, RDA also has a performance edge over the other schemes. Compared with the other four schemes Baseline, AMP, SOML, and preemptive GC, RDA cuts the write response time by an average of 20.7%, 10.3%, 14.4%, and 17.6%, respectively. These findings are expected: GC is a root cause of extended write latency. It is only after completing GC operations that flash chips can generate free space to store the data of write requests. RDA can effectively lower the impact of data fragmentation on GCs, thereby reducing GC

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:22:49 UTC from IEEE Xplore. Restrictions apply.

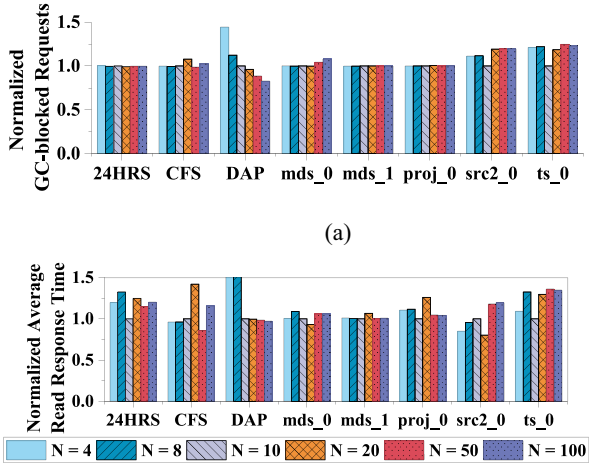

Fig. 14. Performance comparisons between different N values. (a) GC-blocked read requests. (b) Average read response time.

operations and improving write performance. Therefore, RDA immensely shortens read/write latency to reshape the overall SSD performance.

6) Evaluation of N Values: RDA updates the parallelism of the corresponding range by calculating the average size of N read requests. Accordingly, to explore the impact of N values on RDA, we conduct experiments with the six N values: 1) $N = 4$; 2) $N = 8$; 3) $N = 10$; 4) $N = 20$; 5) $N = 50$; and 6) $N = 100$. In the last set of experiments, we quantitatively gauge the RDA performance with different N values in terms of GC-blocked read requests and average read response time.

Fig. 14 (a) and (b) plot the GC-blocked read requests and average read response time of the six N values under the eight workloads. The empirical study demonstrates that due to the diversity of the workload characteristics, RDA exhibits an impressive performance fluctuation with the change of N. Fortunately, $N = 10$ has an outstanding performance exceeding the other N values. Specifically, compared with the five values, N=10 cuts back the number of GC-blocked read requests and read response time by an average of 5.8% and 17.9%, respectively. There are two primary reasons behind these experimental results. First, if the N value is too small, RDA only records a few read requests, which cannot reflect the current I/O access trend and adjust the write parallelism reasonably. Second, since I/O requests have temporal locality, a large N implies a weaker temporal locality. In short, $N = 10$ has an outstanding and stable performance improvement than the other values. Therefore, our RDA adopts the average size of 10 read requests to update the parallelism of a range in the parallel mapping table.

VII. Conclusion

In this article, we proposed a read-request driven adaptive allocation scheme—RDA—to suppress redundant parallel distribution, thereby optimizing the overall performance of SSDs. RDA adjusts the parallelism of write requests with various sizes through historical read requests. Then, RDA allocates the pages of the write requests to multiple parallel units by the specified parallelism. Consequently, RDA is conducive to maintaining read/write parallelism while reducing redundant parallel distribution and data locality in NAND flash-based SSDs is immensely improved.

We conducted experiments to evaluate RDA from the perspective of SSD performance optimization. The experimental results confirm that RDA bolsters the read performance and lifetime of SSDs by reducing redundant parallel distribution. Compared with a baseline dynamic allocation scheme and three state-of-art schemes—AMP, SOML, and preemptive GC—driven by the eight real-world workloads, RDA trims the number of GC-blocked read requests and the read response time by averages of 20.6% and 15.8%, respectively.

References

[1] H. Sun, X. Cheng, C. Zhang, Y. Yue, and X. Qin, "HIPA: A hybrid load balancing method in SSDs for improved parallelism performance," *J. Syst. Archit.*, vol. 131, Oct. 2022, Art. no. 102705.
[2] F. Wen, M. Qin, P. V. Gratz, and A. L. N. Reddy, "Hardware memory management for future mobile hybrid memory systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 11, pp. 3627–3637, Nov. 2020.
[3] C. Kim et al., "A 512-Gb 3-b/cell 64-stacked WL 3-D-NAND flash memory," *IEEE J. Solid-State Circuits*, vol. 53, no. 1, pp. 124–133, Jan. 2018.
[4] F. Wu et al., "Characterizing 3D charge trap NAND flash: Observations, analyses and applications," in *Proc. 36th IEEE Int. Conf. Comput. Design*, 2018, pp. 381–388.
[5] C.-Y. Liu, J. B. Kotra, M. Jung, M. T. Kandemir, and C. R. Das, "SOML read: Rethinking the read operation granularity of 3D NAND SSDs," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2019, pp. 955–969.
[6] M. Kim, W. Jung, H.-J. Lee, and E.-Y. Chung, "A novel NAND flash memory architecture for maximally exploiting plane-level parallelism," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 8, pp. 1957–1961, Aug. 2019.
[7] "3D flash memory toggle DDR2.0 technical data sheet," Data Sheet, Toshiba, Tokyo, Japan, 2016.
[8] "Open NAND flash interface specification revision 5.0." ONFI, 2021. [Online]. Available: https://www.onfi.org/specifications
[9] A. Tavakkol, M. Arjomand, and H. Sarbazi-Azad, "Unleashing the potentials of dynamism for page allocation strategies in SSDs," in *Proc. ACM Int. Conf. Meas. Model. Comput. Syst.*, 2014, pp. 551–552.
[10] S. Wu, C. Du, H. Li, H. Jiang, Z. Shen, and B. Mao, "CAGC: A content-aware garbage collection scheme for ultra-low latency flash-based SSDs," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2021, pp. 162–171.
[11] C.-Y. Liu, J. Kotra, M. Jung, and M. Kandemir, "PEN: Design and evaluation of partial-erase for 3D NAND-based high density SSDs," in *Proc. 16th USENIX Conf. File Storage Technol.*, 2018, pp. 67–82.
[12] W. Wang and T. Xie, "PCFTL: A plane-centric flash translation layer utilizing copy-back operations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 12, pp. 3420–3432, Dec. 2015.
[13] C. Gao, L. Shi, C. J. Xue, C. Ji, J. Yang, and Y. Zhang, "Parallel all the time: Plane level parallelism exploration for high performance SSDs," in *Proc. 35th Symp. Mass Storage Syst. Technol.*, 2019, pp. 172–184.
[14] C. Gao et al., "Exploiting parallelism for access conflict minimization in flash-based solid state drives," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 1, pp. 168–181, Jan. 2018.
[15] C. Gao, L. Shi, K. Liu, C. J. Xue, J. Yang, and Y. Zhang, "Boosting the performance of SSDs via fully exploiting the plane level parallelism," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 9, pp. 2185–2200, Sep. 2020.
[16] Y. J. Seong et al., "Hydra: A block-mapped parallel flash memory solid-state disk architecture," *IEEE Trans. Comput.*, vol. 59, no. 7, pp. 905–921, Jul. 2010.
[17] A. Tavakkol, P. Mehrvarzy, and H. Sarbazi-Azad, "TBM: Twin block management policy to enhance the utilization of plane-level parallelism in SSDs," *IEEE Comput. Archit. Lett.*, vol. 15, no. 2, pp. 121–124, Jul.–Dec. 2016.

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:22:49 UTC from IEEE Xplore. Restrictions apply.

[18] Z. Sha et al., "Low I/O intensity-aware partial GC scheduling to reduce long-tail latency in SSDs," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 4, p. 46, 2021.
[19] W. Kang and S. Yoo, "*Q*-value prediction for reinforcement learning assisted garbage collection to reduce long tail latency in SSD," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 10, pp. 2240–2253, Oct. 2020.
[20] L.-P. Chang, T.-W. Kuo, and S.-W. Lo, "Real-time garbage collection for flash-memory storage systems of real-time embedded systems," *ACM Trans. Embedd. Comput. Syst.*, vol. 3, no. 4, pp. 837–863, 2004.
[21] W. Kang, D. Shin, and S. Yoo, "Reinforcement learning-assisted garbage collection to mitigate long-tail latency in SSD," *ACM Trans. Embedd. Comput. Syst.*, vol. 16, no. 5s, pp. 1–20, 2017.
[22] S. Yan, H. Li, M. Hao, and M. H. Tong, "Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs," in *Proc. 15th USENIX Conf. File Storage Technol.*, 2017, pp. 15–28.
[23] N. Shahidi, M. T. Kandemir, M. Arjomand, C. R. Das, M. Jung, and A. Sivasubramaniam, "Exploring the potentials of parallel garbage collection in SSDs for enterprise storage systems," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2016, pp. 561–572.
[24] B. Mao, S. Wu, and L. Duan, "Improving the SSD performance by exploiting request characteristics and internal parallelism," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 2, pp. 472–484, Feb. 2018.
[25] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and C. Ren, "Exploring and exploiting the multilevel parallelism inside SSDs for improved performance and endurance," *IEEE Trans. Comput.*, vol. 62, no. 6, pp. 1141–1155, Jun. 2013.
[26] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity," in *Proc. 25th Int. Conf. Supercomput.*, 2011, pp. 96–107.
[27] C.-Y. Liu, Y. Lee, W. Choi, M. Jung, M. T. Kandemir, and C. Das, "GSSA: A resource allocation scheme customized for 3D NAND SSDs," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit.*, 2021, pp. 426–439.
[28] H. Uchigaito, S. Miura, and T. Nito, "Efficient data-allocation scheme for eliminating garbage collection during analysis of big graphs stored in NAND flash memory," *IEEE Trans. Comput.*, vol. 67, no. 5, pp. 646–657, May 2018.
[29] F. Wu, Z. Lu, Y. Zhou, X. He, Z. Tan, and C. Xie, "OSPADA: One-shot programming aware data allocation policy to improve 3D NAND flash read performance," in *Proc. 36th IEEE Int. Conf. Comput. Design*, 2018, pp. 51–58.
[30] Y.-M. Chang, Y.-H. Chang, T.-W. Kuo, H.-P. Li, and Y.-C. Li, "A disturb-alleviation scheme for 3D flash memory," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, 2013, pp. 421–428.
[31] X. Zhang, S. Pei, J. Choi, and B. S. Kim, "Excessive SSD-internal parallelism considered harmful," in *Proc. 15th ACM Workshop Hot Topics Storage File Syst.*, 2023, pp. 65–72.
[32] I. Fareed, M. Kang, W. Lee, and S. Kim, "Update frequency-directed subpage management for mitigating garbage collection and DRAM overheads," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 40, no. 12, pp. 2467–2480, Dec. 2021.
[33] K.-T. Park et al., "Three-dimensional 128 Gb MLC vertical NAND flash memory with 24-WL stacked layers and 50 MB/s high-speed programming," *IEEE J. Solid-State Circuits*, vol. 50, no. 1, pp. 204–213, Jan. 2015.
[34] D. Kang et al., "256 Gb 3 b/cell V-nand flash memory with 48 stacked WL layers," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 210–217, Jan. 2017.
[35] M.-C. Yang, Y.-M. Chang, C.-W. Tsao, P.-C. Huang, Y.-H. Chang, and T.-W. Kuo, "Garbage collection and wear leveling for flash memory: Past and future," in *Proc. Int. Conf. Smart Comput.*, 2014, pp. 66–73.
[36] "NAND memory toggle DDR1.0 technical data sheet," Data Sheet, Toshiba, Tokyo, Japan, 2013.
[37] "MSR Cambridge traces." SNIA IOTTA, 2007. [Online]. Available: http://iotta.snia.org/traces/block-io/388?only=386
[38] M. Kwon et al., "TraceTracker: Hardware/software co-evaluation for large-scale I/O workload reconstruction," in *Proc. IEEE Int. Symp. Workload Charact.*, 2017, pp. 87–96.
[39] Y. Cai, S. Ghose, Y. Luo, K. Mai, O. Mutlu, and E. F. Haratsch, "Vulnerabilities in MLC NAND flash memory programming: Experimental analysis, exploits, and mitigation techniques," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2017, pp. 49–60.
[40] J. Lee, Y. Kim, G. M. Shipman, S. Oral, F. Wang, and J. Kim, "A semi-preemptive garbage collector for solid state drives," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2011, pp. 12–21.

Shujie Pang received the M.S. degree from the Computer Science Department, Yantai University, Yantai, China, in 2020. He is currently pursuing the Ph.D. degree with the Computer Science Department, Jinan University, Guangzhou, China.

He has published several papers in *ACM Transactions on Storage* and IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS. His current research interests include storage system, computer architecture, and performance evaluation.

Yuhui Deng received the Ph.D. degree in computer science from the Huazhong University of Science and Technology, Wuhan, China, in 2004.

He is a Professor with the Computer Science Department, Jinan University, Guangzhou, China. Before joining Jinan University, he was with EMC Corporation, Shanghai, China, as a Senior Research Scientist from 2008 to 2009. He was a Research Officer with Cranfield University, Wharley End, U.K., from 2005 to 2008. His research interests cover information storage, cloud computing, green computing, computer architecture, and performance evaluation.

Zhaorui Wu received the M.S. degree in computer architecture from the Computer Science Department, Jinan University, Guangzhou, China, in 2021, where he is currently pursuing the Ph.D. degree.

His current research interests include parallel and distributed computing, computer architecture, and performance evaluation.

Genxiong Zhang received the B.E. degree in computer science and technology from South China Agricultural University, Guangzhou, China, in 2020.

He is a research student with the Computer Science Department, Jinan University, Guangzhou. His current research interests include storage system and nonvolatile memory.

Jie Li received the B.E. degree in automation from the Wanjiang University of Technology, Maanshan, Anhui, China, in 2016, and the M.S. degree in computer science and technology from Guangxi University for Nationalities, Nanning, China, in 2019. He is currently pursuing the Ph.D. degree with the Computer Science Department, Jinan University, Guangzhou, China.

His current research interests include parallel and distributed computing, data center architecture, cloud computing, and data replica placement.

Xiao Qin received the B.S. and M.S. degrees in computer science from the Huazhong University of Science and Technology, Wuhan, China, in 1992 and 1999, respectively, and the Ph.D. degree in computer science from the University of Nebraska-Lincoln, Lincoln, NE, USA, in 2004.

He is currently an Alumni Professor and the Director of Graduate Programs with the Department of Computer Science and Software Engineering, Auburn University, Auburn, AL, USA. His research interests include parallel and distributed systems, storage systems, and performance evaluation.

Prof. Qin won an NSF CAREER Award in 2009.

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:22:49 UTC from IEEE Xplore. Restrictions apply.