IEICE Electronics Express, Vol.*, No.*, 1–12

Boosting Performance of SSD with Chip-level RAID by Deferring Garbage Collection

Jie Liang, Yongkun Li[a)], **Hao Chen and Yinlong Xu**
School of Computer Science and Technology, University of Science and Technology of China
No 96, Jinzhai Road Baohe District, Hefei City, Anhui, 230026, P.R.China
a) *ykli@ustc.edu.cn*

Abstract: Garbage Collection (GC) degrades SSDs' performance notably, especially for SSDs deployed with chip-level RAID. To address this issue, we propose a *deferring garbage collection (DGC)* scheme to improve the I/O performance. DGC first predicts whether GC will be triggered on a chip by monitoring its amount of awaiting write requests and the available free pages, and then redirects some pending writes to other "idle" chips so as to defer the GC on busy chips and mitigate the interference between GC and writes. We implement DGC atop a trace-driven simulator. Compared with traditional GC schemes of SSD deployed with chip-level RAID-5, DGC can reduce the average response time by 5.8% - 46.7%, and the 99-th percentile response time by 25.3% - 77.6%, under different workloads.

Keywords: RAID, SSD, Garbage Collection, performance
Classification: Circuits and modules for storage

References

[1] S. Yan, *et al.*: "Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs," ACM Transactions on Storage. Volume 13 Issue 3.(2017) (DOI:10.1145/3121133).
[2] N. Li, *et al.*: "A page lifetime-aware scrubbing scheme for improving reliability of Flash-based SSD," IEICE Electron. Express **14** (2017) 20170831 (DOI: 10.1587/elex.14.20170831).
[3] Y. Du, *et al.*: "CD-RAIS: Constrained dynamic striping in redundant array of independent SSDs," IEEE CLUSTER(2014) 212. (DOI: 10.1109/CLUSTER.2014.6968742).
[4] M. Jung, *et al.*: "HIOS: A host interface I/O scheduler for Solid State Disks," ACM/IEEE, ISCA(2014) 289. (DOI: 10.1109/ISCA.2014.6853216).
[5] N. Agrawal, *et al.*: "Design Tradeoffs for SSD Performance," USENIX, ATC (2008) 57.
[6] J. Kim, *et al.*: "Chip-level RAID with flexible stripe size and parity placement for enhanced SSD reliability," IEEE Transactions on Computers. Volume 65 Issue 4, (2016) (DOI: 10.1109/TC.2014.2375179).

DOI: 10.1587/elex.15.20180407
Received April 20, 2018
Accepted May 15, 2018
Publicized May 28, 2018

[7] N. Shahidi, *et al.*:"Exploring the Potentials of Parallel Garbage Collection in SSDs for Enterprise Storage Systems," IEEE SC (2017) 561. (DOI: 10.1109/SC.2016.47).
[8] G. Ganger, *et al.*:"The DiskSim Simulation Environment(v4.0)," Parallel Data Lab, http://www.pdl.cmu.edu/DiskSim/ Online-document, (2009).
[9] SNIA. IOTTA repository. http://iotta.snia.org/tracetypes/3.
[10] Y. Li, *et al.*:"Stochastic Modeling of Large-Scale Solid-State Storage Systems: Analysis, Design Tradeoffs and Optimization," ACM SIGMETRICS (2013) 179. (DOI:10.1145/2465529.2465546).
[11] D. Park, *et al.*:"Hot data identification for flash-based storage systems using multiple bloom filters," IEEE MSST (2011) 191. (DOI:10.1109/msst.2011.5937216).
[12] D. Patterson, *et al.*:"A Case for Redundant Arrays of Inexpensive Disks (RAID)," ACM SIGMOD (1988) 688. (DOI:10.1145/971701.50214).
[13] J. Liang, *et al.*:"Improving Read Performance of SSDs via Balanced Redirected Read," IEEE NAS (2016). (DOI:10.1109/NAS.2016.7549406).
[14] C. Petersen, *et al.*:"Solving Latency Challenges with NVM Express SSDs at Scale," https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2017/20170809_SIT6_Petersen.pdf.
[15] J. Kang, *et al.*:"The Multi-streamed Solid-State Drive," USENIX HotStorage (2014).

1 Introduction

Due to the lower power consumption, higher shock resistance, and better random access performance, solid-state drives (SSDs) have progressively replaced hard-disk drives (HDDs), and been extensively used in personal computers and large-scale storage servers.

However, SSDs suffer from various kinds of performance issues, for example, *garbage collection(GC)* [7]. Because of erase-before-write feature and limited endurance, SSDs conduct out-of-place write which writes new data into free pages and marks original old pages as invalid, and as a result, valid pages and invalid pages are mixed in the same block. When the number of free blocks is insufficient, GC would be invoked to reclaim free blocks. During GC, SSDs must incur lots of unexpected reads and writes due to valid page movement [5], after that they need an erase operation which also costs a large amount of time. This results in long waiting time for requests that access the same chips. There exist some recent works [1, 4] on optimizing the performance of user requests which are affected by GC, and they achieve good results. There is a new trend that SSDs are deployed with chip-level RAID-5 [1, 3, 6], to ensure fault tolerance, as MLC/TLC flash memory technology increases SSDs' capacity but sacrifices their reliability [2, 6]. However, in this setting, the interference between GC and user requests becomes much more severe due to the fact that RAID-5 incurs more writes when updating parity chunks. As a result, the above optimization techniques for normal SSDs do not work well for SSDs deployed with chip-level RAID-5. This paper aims to mitigate the interference between GC and user requests in such settings.

Traditional approaches of GC have focused on reducing GC execution time [11], exploring various kinds of victim block selection algorithm [10], or hiding GC latency by performing it at non-critical request [4] to realize optimization of GC workflow. However, almost all works require lots of modifications of system.

On the contrary, in this paper, we study the GC issue of SSD deployed with chip-level RAID-5 via a light-weight modification. Compared with previous methods, we only defer GC to boost performance of I/O requests. We propose a *deferring garbage collection (DGC)* scheme which first predicts whether GC will be triggered on a chip by monitoring its amount of awaiting write requests and the available free pages, and then redirects some pending writes to other "idle" chips so as to defer the GC on busy chips and mitigate the interference between GC and writes. Extensive experiments demonstrate that DGC decreases average response time of I/O requests significantly and reduces the 99-th percentile response time observably over existing schemes with limited storage overhead.

2 Related work

GC causes variability in performance and increases worst-case response time. To address this issue, Jung et al. [4] proposes a host interface I/O scheduler that is both GC-aware and QoS-aware. The scheduler re-distributes GC overheads across non-critical requests to reduce worst-case latency in SSDs. N. Shahidi et al. [7] study one of the least understood problems caused by GC on modern SSDs: plane level resource underutilization. It proactively runs GC on the remaining planes of a flash chip whenever any planes need to execute on-demand GC. In this paper, our proposed DGC scheme does not modify GC operation at any time, instead, it leverages idle chips and idle time to defer GC so as to boost performance of SSD.

To offer consistent read latency, the I/O determinism scheme divides an SSD into multiple NVM sets, each of which is QoS isolated region, and ensures that requests performed against disjoint regions will not interfere[14]. With the support of the isolated NVM sets, I/O determinism scheme maintains two windows, namely, deterministic window and non-deterministic window, in which the order of reads and writes can or cannot be controlled. Using this scheme requires a significant amount of effort of understanding when reads can be promoted before writes. In contrast to this solution, DGC avoids the interference introduced by the background GC to user requests with no need to understand and predict application behaviors. Furthermore, DGC can be used in I/O determinism to mitigate the interference between GC and writes in the non-deterministic window.

Multi-streamed SSDs improve the GC performance by maintaining multiple streams, each of which stores a set of data with the same lifetime expectancy, and sending writes to those streams when the specified lifetime of the corresponding data meets the expectancy of that stream [15]. Unlike this approach, instead of optimizing the GC mechanism, DGC aims to defer GC

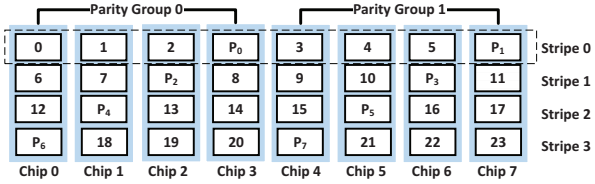

Fig. 1. An SSD deployed with chip-level RAID-5.

until the chip idle time so that GC will not significantly impact the execution of user requests. It is possible to combine DGC and multi-streamed SSDs so that the system performance can be further improved.

3 Background and motivation

We set the traditional log-structured FTL of SSD deployed with chip-level RAID-5 as baseline, and propose DGC to alleviate interference problem between GC and write requests in such settings. In this section, we first introduce the SSD deployed with chip-level RAID-5, then explain GC and its impact on user writes, and motivate our work at last.

3.1 SSD deployed with chip-level RAID-5

We first illustrate the SSD deployed with chip-level RAID-5 via Fig. 1. Chip-level RAID within SSD provides tolerance against chip-level fault as many previous works [1, 6] presented. As shown in Fig. 1, an SSD is divided into stripes which consist of multiple chunks. Each stripe should be across all chips and in practice subdivided into different parity groups. This can shorten recovery cost and strengthen the robustness of the flash chips array. In Fig. 1, the chips are divided into parity group 0 and parity group 1, respectively. There is one parity chunk generated from the data chunks in the same parity group of a stripe. Parity chunks are distributed among all chips in round-robin for load balance. When one data chunk is lost, we can use the remaining data chunks and the corresponding parity chunk in the parity group to recover the lost one. When a data chunk is updated, the corresponding parity chunk would be modified accordingly with either read-modify-write (RMW) or read-construct-write (RCW). RMW first reads the original data chunk and the original corresponding parity chunk, then performs a XOR operation to compute a new parity chunk using these two original chunks and the new data chunk, and finally writes the new data chunk and the new parity chunk to the SSD. As a result, RMW needs four I/O operations and one XOR operation for updating a data chunk. Unlike RWM, RCW needs to read all other chunks which will not be updated in the same parity group, then performs one XOR operation against all chunks including the updated chunk(s) to produce a new parity chunk, and finally writes the updated data chunks and the new parity chunk to the SSD. RAID systems will choose either of the two techniques which leads to fewer I/O operations.

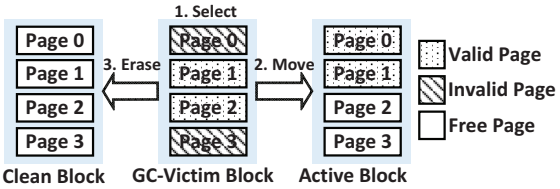

Fig. 2. A schematic of GC.

3.2 Garbage collection

As shown in Fig. 2, the GC operation has three steps. First, FTL would select a candidate victim block to execute GC operation. Second, FTL moves valid pages in the candidate block to a clean block. Meanwhile, entries of the valid pages in the mapping table are updated. At last, the erase operation turns the selected victim block into a clean block. During GC, besides erase operation, many unexpected reads and writes increase rapidly due to valid page movement. This causes sharp degradation of SSDs' performance [4].

3.3 Motivation

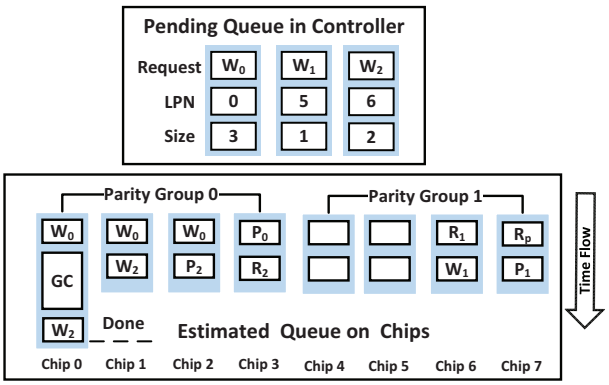

Fig. 3. An example of traditional SSD during GC.

Fig. 3 shows an example of traditional SSD during GC. In the example, there are 8 chips in the SSD and 2 parity groups in a stripe. Each parity group is composed of 4 chunks. There are 3 write requests (W_0, W_1, W_2) in the pending queue of the controller. As indicated in previous study [5], data chunks are stored on all chips in round-robin by modulo operation. Logical page number (LPN) means the initial address of requests. Therefore, we can get the targeted chips of the request according to its LPN and size. W_0 has 3 sub_requests to access chip 0 to chip 2 which implies there is no un-updated chunks in the parity group. Therefore, W_0 has 4 write operations on chip 0 to chip 3 (including write parity chunk P_0) as shown in Fig. 3. The array system chooses RCW to compose the new parity group since RCW requires fewer I/Os. While for W_1, array system would choose RMW to compose its parity group according to default setting when I/O operations are equal. Therefore, W_1 has 2 read and 2 write operations on chip 6 and chip 7 as

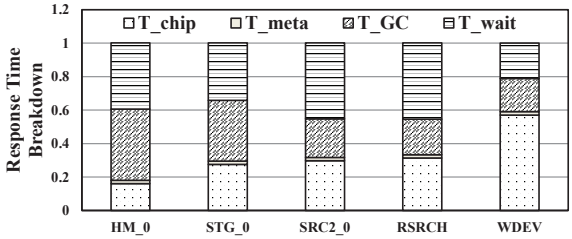

Fig. 4. Response time breakdown.

shown in Fig. 3. As analyzed for W_0, W_2 also chooses *RCW* to compose new parity groups and it has 1 read and 3 write operations on chip 0 to chip 3. Note that, W_2 on chip 0 would invoke GC, and the response time to W_2 would get a huge increase [4, 7] due to the erase operation and valid page movement. Although there are idle chips (chip 4 and chip 5) for W_2, RAID-5 waits for chip 0 to serve it.

To further reveal the impact of GC, we run five traces on an SSD of 64 chips with chip-level RAID-5 to show the breakdown of response time in Fig. 4. The traces are selected from real-world servers by MSR Cambridge [9]. The other experiment settings and characteristics of workloads are described as in Section 5. In Fig. 4, T_chip means latency on chip which only depends on hardware property. T_meta denotes latency of searching or updating metadata of a data chunk. T_GC represents the operating and waiting time of a GC operation. While invoking GC, all the requests behind GC would be added by a large waiting time, as write request 2 shown in Fig. 3. At last, T_wait stands for the normal waiting time (not including waiting time due to GC). We can easily observe that T_GC and T_wait are main factors for the total response time to a request except for WDEV workload.

Based on these results, we see that GC operation of SSD deployed with chip-level RAID-5 degrades the performance notably. To address this issue, we propose a DGC scheme to boost the performance of SSD.

4 Design of DGC scheme

In this section, we first illustrate the main idea of DGC via a simple example, and then describe the system architecture of DGC. At last, we show the design details and system overheads of DGC.

4.1 Main idea

Fig. 5 illustrates how our proposed Deferring Garbage Collection (DGC) works. We still take the scenario shown in Fig. 3 as an example. DGC keeps detecting pending queue and status of chips. When DGC finds that the count of free pages in a chip is less than the number of pending write requests of the same chip, it will send an warning signal to the write request so as to indicate that GC may be triggered soon. In the example, sub_request of W_0 on chip 0 is warned. The DGC will select candidate chip from other parity groups to redirect warned requests. The candidate chip should be relatively idle.

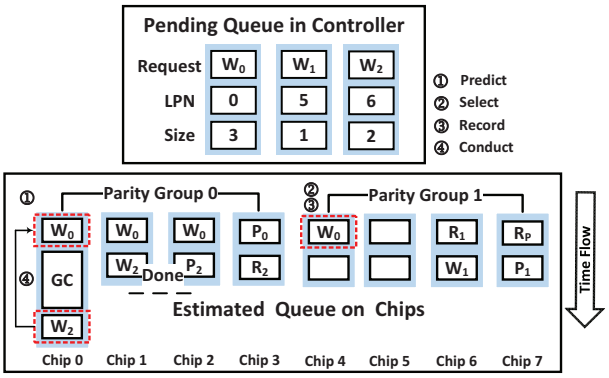

Fig. 5. A schematic of DGC.

In the example, chip 4 and chip 5 are idle for current pending queue, and they are both in parity group 1 while chip 0 is in parity group 0. Therefore, DGC chooses chip 4 to redirect warned W_0. Then DGC records targeted chip number and other metadata information on W_0's mapping entry. Since W_0 has been redirected to chip 4, chip 0 can serve W_2 before invoking GC. The total three pending requests cost two time slots to be served, and this improves the SSD's performance greatly compared to the SSD in Fig. 3. At last, after serving W_2, GC is conducted on chip 0 to reclaim enough pages to serve other requests in pending queue.

4.2 System architecture

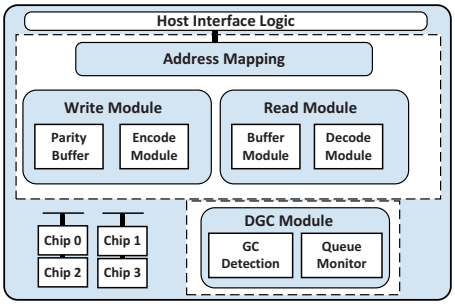

Fig. 6. The design architecture of an SSD with DGC .

We realize DGC by using a trace-driven simulator. The system is composed of three key modules, write module, read module, and DGC module, as depicted in Fig. 6. We do not need to change SSDs' internal module except for mapping bits compared with the SSD deployed with chip-level RAID-5. Write module buffers write requests, encodes data chunks and generates parity chunks. We use a write buffer module to divide incoming write requests into parity groups. Read module consists of a decode module and a buffer module. The decode module serves read or updated requests by locating parity group, and reading corresponding targeted data or parity chunk. The buffer module is designed to limit extra I/Os induced by duplicated reads. DGC module consists of queue monitor and GC detection. DGC can get

all of the chips' queue condition from queue monitor to acquire the count of pending write and read requests. From GC detection module, DGC can get the status of chip to gain the count of free pages. Therefore, DGC can use these information to estimate whether pending write requests would invoke GC operation or not.

4.3 Design details

In this subsection, we describe the design details of DGC to show how DGC predicts GC, how DGC selects targeted chips to redirect write requests, and when and how DGC conducts the deferred GC.

Predict GC. Each SSD's chip has its own queue. DGC can detect the queue conditions to predict the occurrence of GC operation, based on the count of write(C_w) requests in pending queue. DGC uses the information from SSD configuration to record the count of remaining free pages(C_r), and compare it to the C_w, if $C_r < C_w$, then DGC sends a warning to controller. After redirecting requests to other chips, the warned chip's C_w would get decreased. Therefore, the warned chips would stay in a state of $C_r = C_w$. There would be properly enough pages to serve subsequent write requests on the warned chips.

Select targeted chips. After deciding to redirect some writes to other chips, the best chips would be chosen based on following requirements. DGC will select the chips from other parity groups to ensure fault tolerance. This is because that once choosing the chip in the same parity group, and fault occurs on this targeted chip coincidentally, the parity group can not recover it as two data chunks are lost at the same time. DGC will compare estimated completion time (CT) of all other candidate chips. Each CT has been recorded by a global variable. When a request is added in the pending queue, the CT will be updated by adding a read or write unit completion time according to its request type. In addition, the estimated arriving time of requests (AT) is considered by DGC. DGC can get information from pending queue of controller layer to record AT of each chip. DGC should have little influence on targeted chips. Therefore, the targeted chip should meet the requirement that its CT added a write unite should be not more than AT.

Conduct GC. After redirecting requests, the original chips also may be subjected to GC operations. To solve this problem, DGC needs to conduct GC operation on these original chips. Although these chips reach the threshold of over-provisioning space, while there is no requests to invoke GC to reclaim free pages, GC would not be invoked on traditional SSD deployed with chip-level RAID-5. Instead, DGC conducts GC operation during idle time under such a condition. DGC can detect the pending queue condition to recognize idle time, for instance, examining whether pending queue is empty or not.

4.4 Storage overhead

For traditional SSD, FTL has a mapping table to record every logical page number (LPN) to physical page number (PPN) which does not record tar-

Fig. 7. Storage Overhead of DGC with 64 chips.

geted chip number as it can be calculated by a modular operation. Usually, for a 4096 bytes data chunk, it needs 8 bytes mapping entry to implement this function. Whereas for DGC, data chunk needs to change its targeted chip number due to redirection of write operations. There needs to be one extra bit to record whether redirected or not and needs extra $logN$ mapping bits to record targeted chip numbers where N is total number of the chips. For example, supposing that the SSD has 64 chips, DGC only needs to add 7 extra bits to 8 bytes mapping entry for every 4096 bytes data chunk. For a 256GB SSD with 64 chips, there needs 512MB for traditional FTL mapping table, and 56MB extra space for DGC.

4.5 Reliability and write amplification

As shown in Fig. 5, DGC will select the chips from other parity groups to redirect the write request, while the encoding relationship of the redirected write request keeps unchanged. For example, W_0 on chip 4 also takes part in the parity group 0 with W_0 on chip 1, W_0 on chip 2, and P_0 on chip 3 to ensure fault tolerance, although it is located in the parity group 1. Once one of the four chunks is lost, we can recover it by decoding the three remaining chunks. Therefore, we conclude that DGC does not sacrifice the reliability of SSD deployed with chip-level RAID-5.

The write amplification of chip-level RAID-5 within SSD has been well analyzed in Kim et al.'s work [6] and Du et al.'s work [3]. The DGC only redirects write requests to other idle chips, and introduces no extra write for SSD deployed with chip-level RAID-5. Therefore, DGC has no extra write amplification compared with SSD deployed with chip-level RAID-5.

5 Performance evaluation

In the following subsections, we first introduce our system configurations and the workloads used in the experiments, then present the experimental results.

5.1 Experimental setup and workloads

Table I. SSD parameters

Parameter	Value
Page Size	4KB
# of pages per block	32
# of blocks per chip	8192
# of chips per SSD	64
Page read latency	0.025ms
Page write latency	0.200ms
Block erase latency	1.500ms

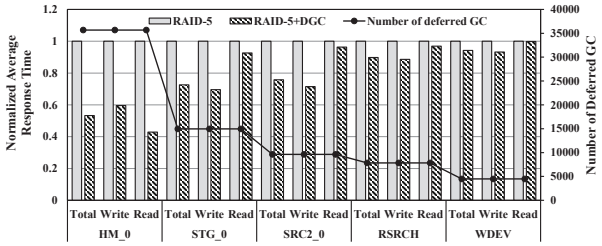

Fig. 8. Average response time of RAID-5 and RAID-5+DGC under the MSR traces.

We use a trace driven simulator, DiskSim with SSD extension [8], which has been widely used in the exploration of SSDs, to evaluate the performance of DGC. We model an SSD based on a common setting as shown in Table I. For data allocation, the widely used round-robin method is applied in page-level address mapping. The over-provisioning ratio is set to 15%. Greedy garbage collection and dynamic wear leveling are implemented in FTL. The parity group size is 8 in following evaluations.

Table II. Statistics of I/O workloads.

Trace	# of Requests	Write Ratio	Working Set	Unique Write Space
HM_0	3993316	0.6451	14.0 GB	4.6 GB
STG_0	2030915	0.8472	10.8 GB	1.3 GB
SRC2_0	1557814	0.8966	15.6 GB	1.2 GB
RSRCH_0	1433655	0.9168	16.9 GB	0.9 GB
WDEV_0	1143261	0.7993	16.9 GB	0.8 GB

The workloads in the experiments are chosen from MSR Cambridge traces [9], which are widely used in previous works to evaluate SSDs' performance. Table II shows the statistics of these traces. The working set means the difference between the maximum page number and the minimum page number. The unique write space indicates unduplicated accessed space for all write requests. Note that the traces are pre-processed to make all requests size be aligned with 4 KB. In Table II, HM_0 trace has the maximum unique write space, and WDEV trace has the minimum unique write space.

5.2 Average response time of requests

Fig. 8 shows the results of the normalized average response time with RAID-5 and RAID-5+DGC. On average, DGC reduces the total response time by 22.9% compared to RAID-5. DGC performs the best under HM_0 workload with a reduction of response time by 46.7%, while performs the worst under WDEV workload, only with a reduction by 5.8%. For write requests, DGC reduces write response time by 23.5% compared to RAID-5 on average. DGC performs the best under HM_0 workload with a reduction of write response time by 40.4%, while performs the worst under WDEV workload, only with a reduction by 6.8%. For read requests, DGC reduces read response time by 14.3% compared to RAID-5 on average. DGC performs the best under HM_0

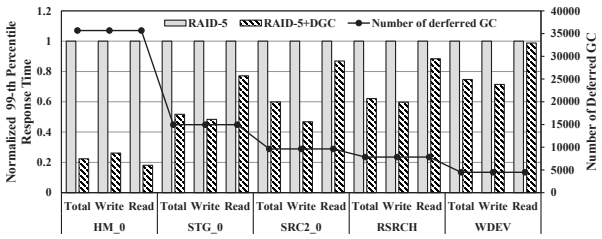

Fig. 9. The 99-th percentile response time of RAID-5 and RAID-5+DGC under the MSR traces.

workload with a reduction of read response time by 57.4%, while performs the worst under WDEV workload, with a reduction by 0.1%. We show the number of deferred GC on the right-side y-axis. It is easily observed that the more GC being deferred, the better performance it gets.

The main reason is that GC has more impact on total response time under HM_0 workload, while less under WDEV workload as shown in Fig. 4. The unique write space influences the GC operation, as the conclusion from [10]. The larger space is, the more latency the GC has. This is because that the larger one has more valid pages in victim block compared with the smaller. Therefore, HM_0 performs better than WDEV.

5.3 The 99-th percentile response time of requests

Fig. 9 shows the results of the 99-th percentile response time with RAID-5 and RAID-5+DGC. On average, DGC reduces the 99-th percentile response time by 48.9% compared to RAID-5. DGC performs the best under HM_0 workload with a reduction of the 99-th percentile response time by 77.6%, while performs the worst under WDEV workload, with a reduction by 25.3%. For write requests, DGC reduces the 99-th percentile response time by 49.5% compared to RAID-5 on average. DGC performs the best under HM_0 workload with a reduction of the 99-th percentile response time by 73.9%, while performs the worst under WDEV workload, with a reduction by 28.6%. For read requests, DGC reduces the 99-th percentile response time by 26.1% compared to RAID-5 on average. DGC performs the best under HM_0 workload with a reduction of the 99-th percentile response time by 81.9%, while performs the worst under WDEV workload, with a reduction by 1.2%. The long tail latency has an obvious dependency on waiting time. DGC can alleviate the GC impact on waiting time, therefore the more avoided GC contentions are, the better performance DGC can get.

5.4 Sensitivity to parity group size

Fig. 10 shows the results of average response time of DGC with different parity group size. We can find that as the parity group size increases, the average response time becomes shorter. When the parity group size increases from 4 to 16, the average response time decreases by 17.8%. The reason is that there are fewer counts of parity chunks as parity group size increases. Therefore, a large number of write and read operations would decrease.

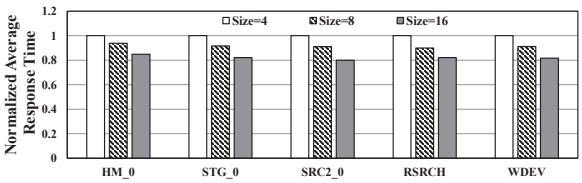

Fig. 10. Average response time of DGC with various parity group size.

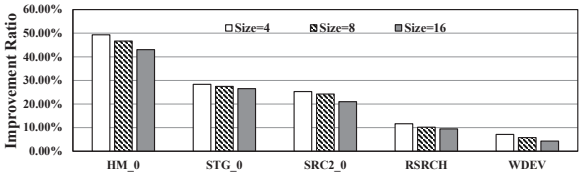

Fig. 11. Improvement ratio of DGC with various parity group size.

Fig. 11 shows the improvement ratio of DGC with different parity group size. We can find that as the parity group size increases, the improvement ratio becomes smaller. When the parity group size increases from 4 to 16, the average improvement ratio of DGC decreases by 3.5%. The reason is that there are fewer counts of parity groups in a stripe as their size rises. This reduces the number of candidate chips for DGC. It becomes more difficult to get relatively idle chips with limited requirements.

6 Conclusion

In this paper, we propose a novel SSD write scheme to improve I/O performance of SSDs with chip-level RAID-5. With DGC scheme, I/O requests are redirected to idle chips for avoiding suffering from large delay of GC operations. When original chips are in idle state, deferred GC would run in background to get enough free space to serve coming requests. For practical deployment, we implement DGC with different enterprise traces atop a widely-used trace driven simulator. Our extensive experiments demonstrate that compared with traditional scheme, DGC decreases average response time of I/O requests significantly and reduces the 99-th percentile response time observably with limited storage overhead.

Acknowledgments

This work was supported by National Nature Science Foundation of China under Grant No. 61772484 and No. 61772486.