Accelerating Garbage Collection of 3D Flash Memory via Exploiting Inter-Channel Parallelism

Hongbin Gong*, Zhirong Shen*‡, Jiwu Shu*†,
*Xiamen University, †Tsinghua University,
‡Innovation Laboratory for Sciences and Technologies of Energy Materials of Fujian Province (IKKEM)
23020201153744@stu.xmu.edu.cn, {shenzr,jwshu}@xmu.edu.cn

***Abstract*—3D flash memory relaxes the scaling constraints but unfortunately arouses the tedious *garbage collection* (GC) problem. We identify via an in-depth analysis that the root cause is the intra-channel relocation in conventional GC operations. We present ParaGC, an approach that strives to speed up the GC process via fully exploiting the inter-channel parallelism of 3D flash memory. ParaGC builds on two elementary designs: (i) relocation arrangement, which carefully determines the relocation traffic that each channel affords; and (ii) page assignment, which dispatches pages based on the access characteristic and the channel busyness. We implement ParaGC and conduct extensive experiments with ten real-world traces, showing that ParaGC can reduce 20.2-41.3% of the read latency, 24.3-38.8% of the write latency, and 51.5-75.8% of the GC latency.**

I. Introduction

Today's planar (or 2D) flash memory is heavily plagued by the scaling constraints, since the conventional method of simply squeezing more bits into a flash cell is hard to compensate the growth of bit errors and access interference [3], [13], [20]. By stacking flash layers (e.g., 24-96 layers [22], [23]) along the *vertical* direction within a single chip, *3D flash memory* can naturally increase the storage density without scaling the process technology, hence opening up an opportunity to break the scaling limits [22].

While achieving larger storage capacity, the 3D flash memory also comes with an intractable *garbage collection* (GC) problem. The reason is that the flash memory updates data following an "out-of-place" manner; that is, it writes the newly updated data to another clean *flash page*, while at the same time marking the old data and the corresponding resided flash page *invalid* [27]. Hence, as the number of clean pages gradually declines, the flash memory is forced to periodically invoke GC operations, which can supply clean pages by choosing and erasing a victim *flash block* (composed of multiple pages). Nevertheless, the GC operation amplifies the writes in return, since it requires relocating the residual *valid pages* (i.e., the pages still containing valid data) beforehand. As a block in the 3D flash memory is enlarged by stacking more layers [8], the *relocation traffic* (i.e., the amount of data relocated during the GC operation) explosively grows, making the GC latency prolonged.

In view of this, lots of research studies have been conducted to accelerate the GC operation for 3D flash memory, which can be roughly classified into the following branches: (i) suppressing the relocation traffic [4], [7], [8], [11], [13], [18], [21], [25], [30], and (ii) shortening the relocation latency [6], [15], [19], [26], [28], [29]. However, we carefully identify that *most of them still neglect the inter-channel access parallelism and serialize the GC operations within a single channel, making the GC operation become time-consuming.* How to exploit the inter-channel access parallelism to shorten the GC operation is unfortunately largely unexplored.

Our observation is that the lengthy GC operation of the 3D flash memory is demonstrated to impose severe impact on the foreground requests, which stems mainly from the tedious data relocation (see Section II-B). We hence propose ParaGC, an approach that seeks to accelerate the data relocation of the GC operation through exploiting the inherent inter-channel parallelism. ParaGC first carefully determines the number of valid pages to be relocated across channels, with the aim of alleviating the affection between the data relocation and the foreground requests. ParaGC further selects the valid pages for each channel based on the access hotness (of pages) and the busyness (of channels), so as to speculatively balance future accesses across channels. To the best of our knowledge, ParaGC is the *first* work that proposes to speed up the data relocation via exploiting the inter-channel parallelism of the 3D flash memory. Furthermore, ParaGC complements previous studies [4], [7], [8], [11], [13], [18], [19], [21], [30] on the GC acceleration and thus can work together with them for more performance gains. To summarize, this paper makes the following **key contributions**.

- We conduct a trace-driven analysis to assess the interference induced by the GC operation and identify the root cause (Section II-B).
- We propose ParaGC that exploits the inter-channel parallelism to accelerate the data relocation in the GC operation. ParaGC carefully arranges the data relocation based on access characteristics and channel busyness, hence alleviating the interference between the foreground requests and the data relocation (Section III).
- We conduct extensive experiments, showing that ParaGC can reduce 20.2-41.3% of the read latency, 24.3-38.8%

Corresponding author: Zhirong Shen (shenzr@xmu.edu.cn). This work was supported by the National Key R&D Program of China (2021YFF0704001), Natural Science Foundation of China (No. 62072381), Science and Technology Projects of Innovation Laboratory for Sciences and Technologies of Energy Materials of Fujian Province (IKKEM) HRTP-[2022]-1, CCF-Huawei Innovation Research Plan (CCF2021-admin-270-202102), and Xiamen Youth Innovation Fund (3502Z20206052).

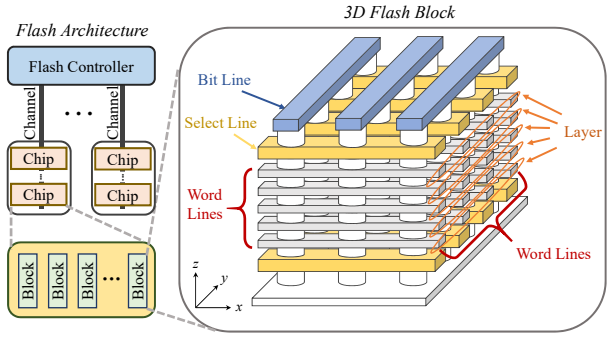


Fig. 1. The bird's eye view of a 3D flash block.

of the write latency, and 51.5-75.8% of the GC latency (Section IV).

The rest of this paper proceeds as follows. Section II provides the fundamental knowledge about the flash memory. We elaborate on the design of ParaGC in Section III and evaluate its performance in Section IV. We finally review existing studies in Section V and conclude this paper in Section VI.

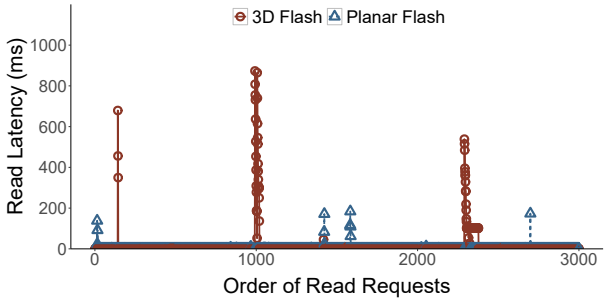
II. Background

In this section, we start with the necessary basics on 3D flash memory (Section II-A) and then elaborate on the observations learned from the trace-driven analysis (Section II-B).

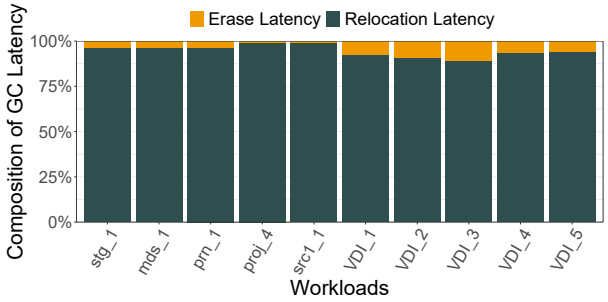
A. 3D Flash Memory

Figure 1 first depicts the architecture of the 3D flash memory. In the 3D flash memory, an *SSD controller* connects multiple (e.g., 2-10) *channels*, each of which is shared by a number of *chips*. A chip is composed of multiple *flash blocks*, which comprise thousands of *flash pages*. Figure 1 further zooms in the architecture of a flash block (at the right side), which stacks five *flash layers* (in orange). A flash layer has four *word lines*, each of which comprises multiple flash cells for data storage.

Basically, flash memory represents binary bits through controlling the number of electrons trapped in *flash cells* (i.e., the basic storage units) [16], [22]. It operates data through three basic operations, namely write, read, and erase. Specifically, the flash memory adopts the *out-of-place update*: when a write operation arrives, the controller will select a clean *flash page* (often made up of multiple cells) and inject a certain number of electrons to represent the updated data, while the outdated data will be *invalidated*. Hence, the read and write operations in flash memory are performed in units of pages. When the clean flash pages are used up, the controller will select a *flash block* (called "victim block") based on the footprints of the invalid pages. It then performs a GC operation to recycle the space occupied by the invalid pages in the selected victim block through the following steps: (i) relocating the valid pages to another block *within the same channel*, and (ii) erasing the entire victim block to replenish clean pages. Hence, the relocation latency and the erase latency collectively constitute the GC latency.



(a) **Ob#1**: GC process in the 3D flash memory is more severe



(b) **Ob#2**: The relocation latency predominates the GC latency

Fig. 2. Observations from trace-driven analysis.

TABLE I
Abbreviations of Selected VDI Traces [17]

Trace names	20160220 08-LUN0	20160218 10-LUN2	20160217 12-LUN3	20160219 09-LUN6	20160219 18-LUN3
Abbreviated names	VDI_1	VDI_2	VDI_3	VDI_4	VDI_5

B. Observations

We conduct an in-depth trace-driven analysis to understand how severely the lengthy GC process in the 3D flash memory affects the foreground requests and try to catch the root cause.

Methodology: We select one typical real-world trace `proj_4` from MSR Cambridge Traces [24], which includes 6,369,774 access requests. We warm up the 3D flash memory by repeatedly writing random data, so as to run short of clean pages and trigger the GC operation for evaluation. We then monitor one particular channel, replay 3,000 read requests to this channel, and measure the resulting read latencies. We also measure the composition of the actual GC latency by replaying some traces from MSR Cambridge Traces [24] and the state-of-the-art production *virtual desktop infrastructure* (VDI) [17]. For clear presentation, we change the name of the traces selected from the VDI repository and give their abbreviations in Table I. The 3D flash and planar flash have the same capacity, with the difference that the 3D flash has a larger block size to keep up with the capacity growth. Figure 2 shows the results and conveys the following observations.

Observation 1 (More severe interference). The GC operation in the 3D flash memory imposes more severe interference to the foreground requests (Figure 2(a)), where the induced latency spike is 3.3x taller than that in the planar flash memory

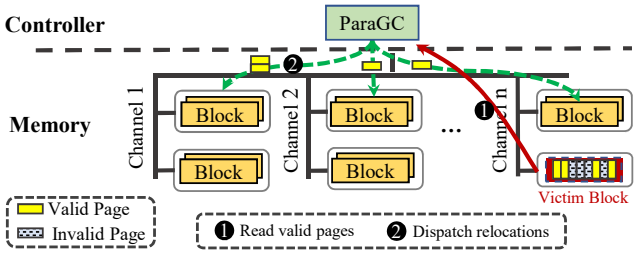


Fig. 3. Workflow of ParaGC.

on average. We identify the increased relocation traffic coming from the victim block is the major cause, as the migration will incur more drastic competition for storage I/O resources.

Observation 2 (Relocation latency dominates). The relocation latency takes up 94.4% of the GC latency on average (Figure 2(b)). The reason is that the GC latency is composed of the erase latency and the relocation latency (Section II-A), where the former is nearly constant (e.g., 10 ms [12]) while the latter increases with the number of valid pages; in the data relocation, the time of writing a page (e.g., 3 ms [12]) is serval orders of magnitude higher than that of reading a page(e.g., 66 μs [12]).

Given the severity of the GC operation and the composition of the GC latency, we hence pose the following question: *Can we exploit the inter-channel parallelism to accelerating GC operation by reducing the relocation latency?*

III. ParaGC Design

We present ParaGC, an approach that seeks to exploit the inter-channel access parallelism for accelerating the GC process. The *main idea* of ParaGC is to relocate valid pages *across channels* (as opposed to serializing them within a single channel), so as to parallelize the data relocation and hence shorten the GC latency. Figure 3 depicts the workflow of ParaGC. It first reads the valid pages to be relocated into the RAM (represented by the red solid arrow, Step ❶), and then dispatches these pages across channels for load balancing (represented by the green dotted arrow, Step ❷).

Overview: ParaGC is built atop the following design primitives: (i) *a relocation arrangement algorithm*, which monitors the *service rate* (i.e., the size of the read requests served per time unit) of each channel and accordingly determines the number of valid pages to be relocated, so as to relieve the interference induced to the foreground requests (Section III-A); and (ii) *a hotness-aware page assignment algorithm*, which carefully determines the exact valid pages to be relocated for the channels, by taking both the service rates (of channels) and the access characteristics (of valid pages) into consideration, so as to speculatively balance future accesses after the relocation (Section III-B).

A. Relocation Arrangement

Real-world storage loads usually exhibit highly skew access properties [1], [2], [5], and hence the numbers of read requests loaded may vary across channels. Randomly dispersing the valid pages to parallelize the data relocation is not a good option, as it may bring back the unbalanced requests. In view of this, ParaGC proposes to monitor the read requests constantly, where each channel will maintain a *ring* structure (with r slots) that records the amount of read data (denoted by d) during the last r time units [1]. By doing this, we can calculate the *service rate* of each channel, given by $\frac{d}{r}$. Hence, a larger service rate indicates that the corresponding channel is much busier for serving the foreground read requests, and hence should be assigned with less relocation traffic. For instance, if a channel transfers 500 MB of data in total to serve the foreground reads during the last five seconds, then its service rate is 100 MB/s.

Assumptions: ParaGC makes the following assumptions. First, we assume that the service rate of each channel keeps stable during the data relocation. Second, we assume that the valid pages to be relocated are given higher priorities to be executed, so as to shorten the GC operation. Third, we pay special attention to the interference between the data relocation and the foreground reads, as the writes can be served by choosing a channel that is not occupied by the GC operations.

Problem formulation: We first formulate the relocation arrangement problem. Suppose that there are n channels (denoted by $\mathcal{C} = \{C_1, C_2, \cdots, C_n\}$) and their service rates are $\{s_1, s_2, \cdots, s_n\}$, respectively (where $s_i \geq 0$ for $1 \leq i \leq n$). Without loss of generality, we assume that one block of the last channel (i.e., C_n) is selected as the victim block for launching a GC operation, which introduces the relocation of v valid pages (where $v \geq 0$). Let $\{v_1, v_2, \cdots, v_n\}$ be the numbers of valid pages to be relocated across the n channels. Hence, we have

$$v_1 + v_2 + \cdots v_n = v, \quad \text{where } v_i \geq 0 \text{ for } 1 \leq i \leq n \qquad (1)$$

Suppose that T is the average time for a channel to write a valid page. Hence, the time for each channel C_i to write the assigned valid pages can be computed as $t_i = v_i \cdot T$ (where $1 \leq i \leq n$). Besides, to guarantee data reliability for accidental failures (e.g., power outage), we require the last channel to finally erase the victim block after ensuring that all the valid pages have been successfully settled down, and hence the stall time of C_n is $\max\{t_i\}_{i=1}^{n}$.

Based on the above analysis, we can compute *the accumulated amount of data (denoted by D) that are requested by foreground reads but suspended during the data relocation*, given by

$$D = s_1 \cdot t_1 + s_2 \cdot t_2 + \cdots + s_{n-1} \cdot t_{n-1} + s_n \cdot \max\{t_i\}_{i=1}^{n} \qquad (2)$$

Clearly, D is not a constant value but depends on the relocation arrangement (i.e., the arrangement of $\{v_i\}_{i=1}^{n}$ across the n channels). Hence, we can establish the following optimization problem as follows, which aims to minish the requested data that are interfered during the data relocation (i.e., objective),

[1]The number of slots and the time unit can be configured according to system requirements.

Algorithm 1 Relocation Arrangement

Input: v (number of valid pages to be relocated)
$\{s_1, s_2, \cdots, s_n\}$ (service rates of n channels)
λ (iteration limit)
Output: $\{v_1^*, v_2^*, \cdots, v_n^*\}$ (arrangement of valid pages across n channels)

1: Generate an arrangement $\{v_1, v_2, \cdots, v_n\}$ and compute D
2: **repeat**
3: `// Try each pair for tentative adjustment`
4: **for** each channel $C_i \in \mathcal{C}$ **do**
5: **for** each channel $C_j \in \mathcal{C}$ and $C_j \neq C_i$ **do**
6: Set $v_i' = v_i - 1$, $v_j' = v_j + 1$
7: Compute D'
8: Computer the reduction $u_{i,j} = D - D'$
9: **end for**
10: **end for**
11: `// Perform the adjustment with the most reduction`
12: Set $(i^*, j^*) = \arg\max_{(i,j)}\{u_{i,j} | 1 \leq i \neq j \leq n\}$
13: **if** $u_{i^*,j^*} < 0$ **then**
14: break
15: **end if**
16: Set $v_{i^*} = v_{i^*} - 1$, $v_{j^*} = v_{j^*} + 1$
17: Set $\lambda = \lambda - 1$
18: **until** $\lambda = 0$
19: **return complete**

while at the same time exploring opportunities to parallelize the GC process (i.e., condition).

$$\text{Minimize} \quad D = \sum_{i=1}^{n-1} s_i \cdot t_i + s_n \cdot \max\{t_i\}_{i=1}^{n} \tag{3}$$

subject to

$$\begin{aligned} & t_i = v_i \cdot T \quad \text{for } 1 \leq i \leq n \\ & v_1 + v_2 + \cdots + v_n = v \\ & v_i \geq 0 \quad \text{for } 1 \leq i \leq n \end{aligned} \tag{4}$$

Solution: Finding the optimal solution requires $\binom{v+n-1}{n-1}$ attempts [2], which is extremely time-consuming. ParaGC puts forward a greedy algorithm to search the solution that can approach the optimal one with polynomial time complexity. Algorithm 1 elaborates on the procedure of the relocation arrangement. It first generates an initial arrangement and computes D (i.e., the amount of requested data affected by the data relocation, see Equation (2)). We then seek to adjust the arrangement to achieve the most reduction by performing the following steps: (i) we try each pair of channels (i.e., C_i and C_j, where $1 \leq i \neq j \leq n$) and calculate the reduction of D, once we tentatively migrate one page (which is supposed to be assigned to C_i) from C_i to C_j (Lines 4-10); and (ii) we finally choose the pair (i.e., C_{i^*} and C_{j^*}) that can lead to the most reduction and perform the adjustment (Lines 12-17). We repeat the adjustment until either the adjustment is useless (Lines 13-15) or the iteration reaches the limit (Line 18). We can readily deduce that the computational complexity of Algorithm 1 is $O(\lambda n^2)$, where λ is a pre-given iteration limit.

[2]This problem can be converted to putting v balls into n boxes that allows some boxes to be empty. So there are $\binom{v+n-1}{n-1}$ possible combinations.

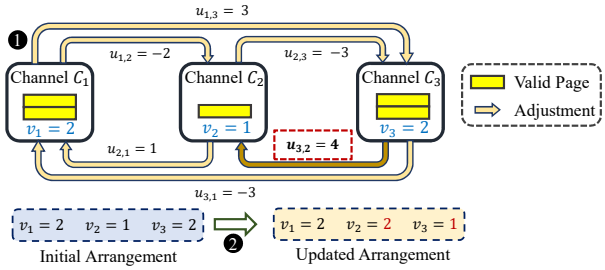


Fig. 4. Example of the relocation arrangement with five valid pages and three channels (i.e., $v = 5$ and $n = 3$).

Example: In Figure 4, we suppose that the initial arrangement is $\{v_1 = 2, v_2 = 1, v_3 = 2\}$. We scan each channel and try all six tentative adjustments (Step ❶). We choose the adjustment that migrates one more page (which is supposed to be assigned to C_3) to C_2 can achieve the largest gain (i.e., $u_{3,2} = 4$). We then update the arrangement to $\{v_1 = 2, v_2 = 2, v_3 = 1\}$ (Step ❷) and go to the next round of the adjustment.

B. Hotness-Aware Page Assignment

ParaGC further takes the data access characteristics into account, so as to speculatively mitigate the read hotspots after relocation. To keep track of the read frequencies of the flash pages, ParaGC designs a memory-efficient `hotness identifier`, which employs a *count-min sketch* [9] to record the number of accesses to each page. The count-min sketch uses multiple independent hash functions to map the read pages to the *counters* of a hash table (whose values are set to zero at the very beginning). It can estimate the access frequencies that falls within a certain distance with the true frequencies with a certain probability [10]. To capture the hotness of a flash page, the hotness identifier first defines m *hotness thresholds* (i.e., $\{k_1, k_2, \cdots, k_m\}$, where $k_1 < k_2 < \cdots < k_m$), in order to classify the flash pages into m groups (denoted by $\{\mathcal{G}_1, \mathcal{G}_2, \cdots, \mathcal{G}_m\}$) based on their access frequencies. Hence, when a read request arrives, ParaGC first pinpoints the corresponding counters by feeding the accessed logical page number to the hash functions, increases the values of the associated counters by one, and uses the smallest value of these counters (denoted by e) to serve as the estimated access frequency; ParaGC then establishes a hotness threshold k_i ($1 \leq i \leq m$), such that either of the following three conditions establishes: (i) $e < k_1$ (i.e., the page is not a hotly accessed page); (ii) $k_i \leq e < k_{i+1}$ where $1 \leq i \leq m-1$; and (iii) $k_m \leq e$. Hence, we organize the page to the group $\mathcal{G}_i$ (if $k_i \leq e < k_{i+1}$, where $1 \leq i \leq m-1$) or the group $\mathcal{G}_m$ (if $k_m \leq e$). Hence, a page is considered to be hotter if the identify of the group to which it belongs is larger. To downgrade the impact of the historical accesses, we decay all counter values by half after performing a certain number of read requests. We measured that the hotness identifier consumes only 238.4 KB of RAM space for a 288 GB SSD (Section IV-A).

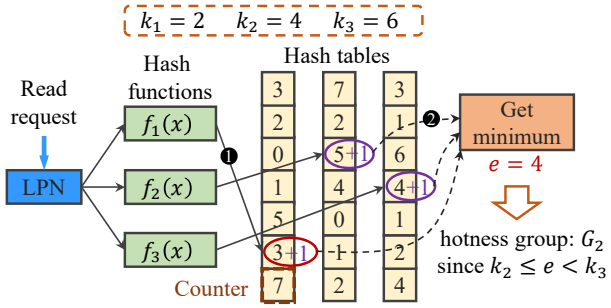


Fig. 5. Example of the hotness identifier that determines the hotness group to which the page belongs.

Algorithm 2 Hotness-Aware Page Assignment

Input: v (number of valid pages to be relocated)
$\{P_1, P_2, \cdots, P_v\}$ (v pages to be relocated)
$\{C_1, C_2, \cdots, C_n\}$ (n channels)
Output: $\{\mathcal{A}_1, \mathcal{A}_2, \cdots, \mathcal{A}_n\}$ (sets of pages relocated to n channels)

1: Sort v pages based on their hotness groups in descending order and get $\mathcal{P} = \{P_1, P_2, \cdots, P_v\}$
2: Sort n channels based on their service rates in ascending order and get $\mathcal{C} = \{C_1, C_2, \cdots, C_n\}$
3: **for** each channel $C_i \in \mathcal{C}$ **do**
4: Establish $\mathcal{A}_i$, which comprises the hottest v_i^* pages of $\mathcal{P}$
5: Set $\mathcal{P} = \mathcal{P} - \mathcal{A}_i$
6: **end for**
7: **return complete**

Example: Figure 5 shows an example of the hotness identifier, where we assume $k_1 = 2$, $k_2 = 4$, and $k_3 = 6$. For a flash page accessed by a read request, the hotness identifier first pinpoints the three counters steered by the hash functions and increases the corresponding values by one (Step ❶). It then uses the smallest value of the three counters (i.e., 4) to estimate its access frequency and learns that the flash page is in the hotness group $\mathcal{G}_2$, since $k_2 \le e = 4 < k_3$ (Step ❷).

After determining the hotness groups, we select the channel for accommodating each relocated page. Algorithm 2 shows the detailed procedure to pinpoint the valid pages to be relocated for each channel, whose *main idea* is to dispatch hotter pages to the channels with smaller service rates, so as to speculatively balance the read requests across channels and relieve the read hotspots.

Details of Algorithm 2: ParaGC first sorts the v valid pages based on the identities of the groups to which they are belonged in descending order, which are represented by $\mathcal{P} = \{P_1, P_2, \cdots, P_v\}$. We also rank the channels based on their service rates in ascending order and assume that the channels after sorting are $\mathcal{C} = \{C_1, C_2, \cdots, C_n\}$, where $s_1 \le s_2 \le \cdots \le s_n$. We then scan each channel sequentially. For a channel C_i (where $1 \le i \le n$), we can get the number of valid pages assigned (i.e., v_i^*, see Section III-A). We then fetch the v_i^* hottest pages from $\mathcal{P}$ and expel them from $\mathcal{P}$ (Lines 3-5). After all the valid pages are assigned to the corresponding channels, the algorithm terminates. We can readily deduce that the computational complexity of Algorithm 2 is $O(n \log n + v \log v)$.

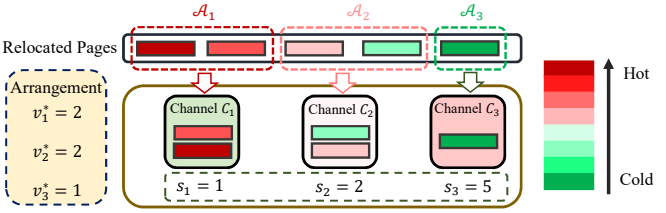


Fig. 6. Example of the hotness-aware page assignment with five valid pages and three channels (i.e., $v = 5$ and $n = 3$).

Example: In Figure 6, we sort the channels in ascending order of the service rates and sort the valid pages in descending order of the identities of the group to which they are assigned. For instance, given the channel C_1, we check the number of valid pages arranged is two (i.e., $v_1^* = 2$) and then assign the first two pages after sorting (i.e., $\mathcal{A}_1$) to C_1. We repeat the assignment for all the channels. We can observe that this assignment can opportunistically balance the access requests across channels.

IV. Evaluation

We conduct extensive experiments to evaluate the performance of ParaGC. We summarize our major findings as below: (i) ParaGC can reduce 25.3-41.3% of the read latency, 24.3-38.8% of the write latency, and 51.1-73.8% of the GC latency for a wide spectrum of real-world traces (Section IV-B); and (ii) ParaGC can retain its effectiveness under sensitivity experiments (Section IV-C).

A. Experimental Setup

Preparation: We implement a ParaGC prototype based on the trace-driven simulator SSDsim [14]. The flash capacity is 288 GB, which includes eight channels, and each channel contains two chips with one die per chip. Each die contains 1,536 blocks, each of which consists of 768 pages. The page size is 16 KB. Besides, the program, read, and erase latencies are configured as 3 ms, 66 μs, and 10 ms, respectively [12]. Besides, we also employ five independent hash functions along with five hash tables in the count-min sketch to record the access frequencies of the accessed pages. A page is considered hot only if the smallest value of the associated counters is not smaller than two. We run experiments on a server with Ubuntu 18.04.1 LTS, which is equipped with Intel Xeon CPU E3-1225 v6 (3.30 GHz) and 16 GB RAM.

Default configurations: We choose the following configurations throughout the evaluation, unless otherwise specified. We set the number of channels as eight and set the *GC threshold* (i.e., the value that the clean pages occupy to start a GC operation) as 20%.

Comparison approaches: We compare ParaGC against another two GC approaches: (i) the Baseline, which relocates all the valid pages of the victim block *within a single channel*. (ii) a GC method that relocates valid pages over multiple

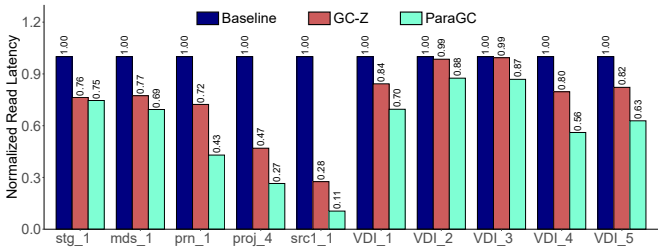


Fig. 7. Exp# 1 (Read latency).

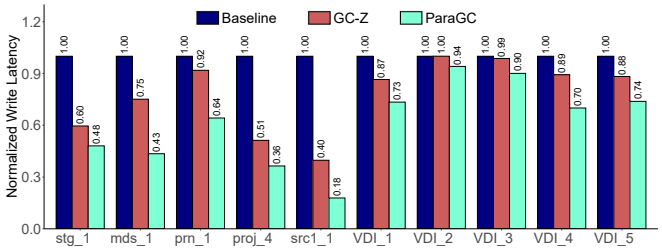


Fig. 8. Exp# 2 (Write latency).

channels, whose distribution follows the Zipf distribution with an $\alpha = 0.95$ (called "GC-Z").

Methodology: We randomly select ten representative traces from both the empirical MSR Cambridge Traces [24] and the state-of-the-art production VDI [17]. Before evaluation, we warm up the flash memory by continuously writing a large amount of random data, making the ratio that remaining clean pages occupy drop below the pre-defined GC threshold. After that, we then dispatch the access requests of the selected traces and measure the latencies. For clear presentation, we normalize the results to those of the Baseline.

B. Experiments on Property

We first evaluate the properties of ParaGC, in terms of read latency, write latency, and GC latency.

Exp# 1 (Read latency): We first measure the read latencies under different GC approaches. Figure 7 shows that compared to the Baseline and GC-Z, ParaGC can reduce 41.3% and 25.3% of the read latencies on average, respectively. The underlying reason is that the Baseline simply performs the data relocation within the single channel, hence seriously suspending the incoming read requests. GC-Z though disperses the relocated data across multiple channels, it still does not well coordinate the executions of the read requests and the data relocation, thereby introducing significant interference. As a comparison, ParaGC carefully arranges the data relocation based on the service rates of the channels (Section III-A), such that the interference is greatly alleviated.

We also observe that ParaGC dramatically reduces the read latencies for some traces (e.g., `proj_4` and `src1_1`). This is because both of them incur massive relocation traffic, hence parallelizing the data relocation can achieve more performance gains.

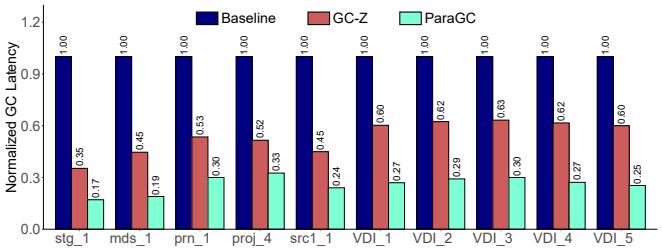


Fig. 9. Exp# 3 (GC latency).

Exp# 2 (Write latency): We then measure the write latencies under different GC approaches. Figure 8 indicates that ParaGC reduces 38.8% and 24.3% of the write latencies on average when compared to the Baseline and GC-Z, respectively. The fundamental reason is that the relocation arrangement in ParaGC (Section III-A) helps it relieve the interference and hence the foreground writes can make full use of the idle I/O resources. In contrast, the Baseline relocates all the valid pages of the victim block within a single channel and the GC-Z does not take into account the load of each channel, thereby resulting in under-utilization of the available I/O resources.

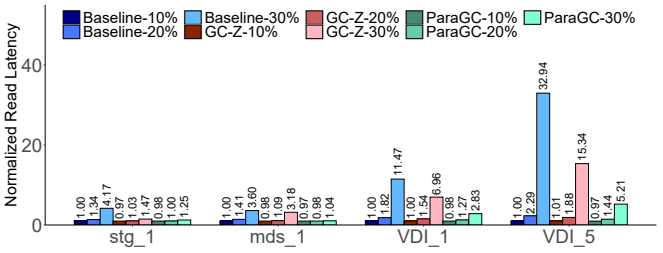
Exp# 3 (GC latency): We also evaluate the GC latency, which comprises the latencies for data relocation and block erase. Figure 9 shows that ParaGC can dramatically reduce the GC latencies by 73.8% and 51.1% when compared to the Baseline and GC-Z, respectively. The major reason is that ParaGC parallelizes the data relocation to utilize the available I/O resources across channels. Besides, the Baseline always introduces the longest GC process since it sequentializes the data relocation and migrates all the relocated pages within the same channel.

C. Experiments on Sensitivity

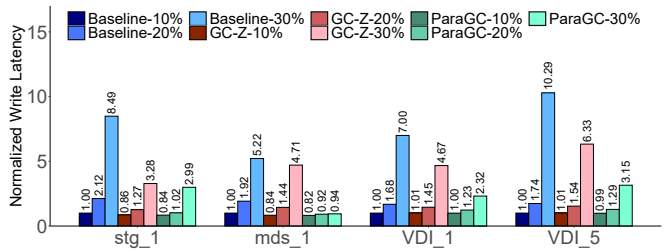
We assess the sensitivity of ParaGC by varying the default configurations. Due to page limits, we select four representative traces (i.e., `stg_1`, `mds_1`, `VDI_1`, and `VDI_5`) for comparison.

Exp# 4 (Impact of GC threshold): We first measure the impact of the GC threshold by varying it from 10% to 30%. Figure 10 shows the read latency, the write latency, and the GC latency under different GC thresholds. We find that the latencies all increase with the GC threshold, since the number of valid pages to be relocated probabilistically grows with the GC threshold, exacerbating the interference between the foreground access and the background relocation. Nevertheless, ParaGC still averagely reduces 29.4% of the read latency, 31.6% of the write latency, and 63.5% of the GC latency under different GC thresholds.

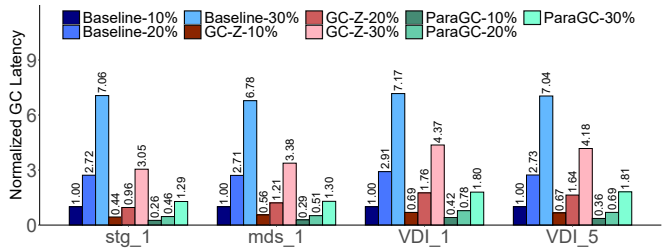
Besides, we also find that ParaGC can gain higher performance improvement when there are more pages to be relocated. Take the trace `stg_1` as an instance, when the GC threshold is 10%, ParaGC can reduce the write latency by 16.0% compared to the Baseline (Figure 10(b)); when the GC



(a) Read latency.

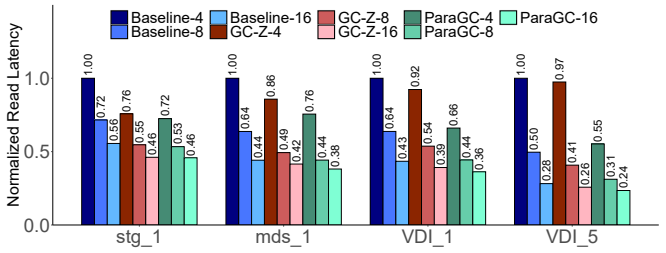


(b) Write latency.

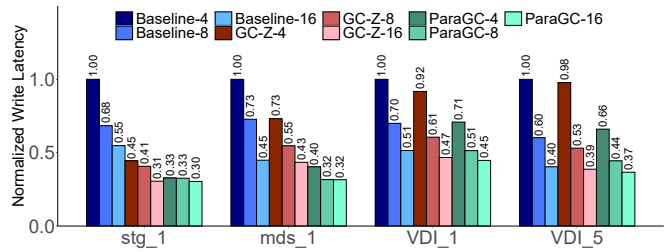


(c) GC latency.

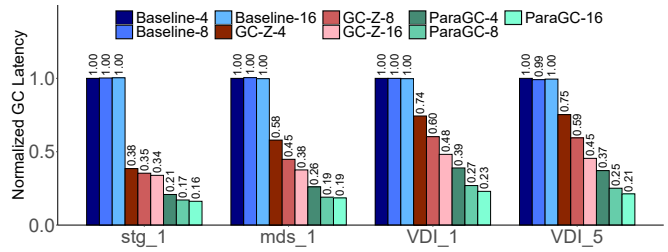
Fig. 10. Exp# 4 (Impact of GC threshold).



(a) Read latency.



(b) Write latency.



(c) GC latency.

Fig. 11. Exp# 5 (Impact of number of channels).

threshold increases to 30%, the reduction of the write latency reaches 64.8%. The underlying reason is that a larger GC threshold will result in a significant increase in the number of the pages to be relocated. Generally, more pages to be relocated will inevitably lead to a longer relocation process, and parallelizing the data relocation across different channels can effectively shorten the relocation time.

Exp# 5 (Impact of number of channels): We further assess the impact of the number of channels, which is varied from 4 to 16. Figure 11 shows the results and we can draw two findings.

First, the read and write latencies all reduce when the number of channels increases (Figure 11(a)(b)). This is because when there are more channels, the controller can dispatch more access requests and parallelize their executions, hence shortening the suspending time. ParaGC still achieves the lowest read and write latencies, and cuts down 20.2% of the read latencies and 29.4% of the write latencies on average compared to the Baseline and GC-Z, respectively.

Second, the GC latency of the Baseline remains unchanged, while those of ParaGC and GC-Z both decline when the number of channels increases (Figure 11(c)). The rationale is that the Baseline serializes the GC operation within a single channel, and hence the GC latency is not affected by the number of channels. Conversely, when there are more channels, ParaGC and GC-Z can leverage the available I/O resources to increase the relocation parallelism, thereby shortening the GC latencies. Overall, ParaGC reduces the GC latencies by 75.8% and 52.5% compared to the Baseline and GC-Z under different numbers of channels, respectively. Besides, as the number of channels increases, the relative benefit of ParaGC decreases. While the read requests load of each channel under the baseline and GC-Z is also relatively reduced, thereby alleviating the blocking problem to some extend. However, considering the actual needs and hardware overhead, the number of channel is generally 4 or 8, so ParaGC can still perform well.

V. Related Work

We classify the closest related work into two categories: (i) suppressing relocation traffic [4], [7], [8], [11], [13], [18], [21], [25], [30], and (ii) accelerating data relocation (without changing the relocation traffic) [6], [15], [19], [26], [28], [29].

Suppressing relocation traffic: Some studies [4], [8], [13], [21], [25] propose to break a block into a number of smaller sub-blocks and perform the GC operation at the sub-block granularity, such that only the valid pages resided in the sub-blocks selected to be erased need to be migrated. However, the sub-block erase needs to amend the circuit, and results in either storage loss [8] or erase interference among sub-blocks [4]. Some studies [7], [13], [30] suggest separating the data with

different access hotness and centralizing the storage of hotly-updated data, so as to opportunistically increase the number of invalid pages and reduce the relocation traffic. ShadowGC [11] and CDM [18] find that reading the cached data directly from the write buffer and NVM can alleviate relocation traffic caused to the flash memory, respectively.

In contrast, ParaGC does not change the relocation traffic but boosts the GC operation through exploiting the inter-channel parallelism. It is orthogonal to these studies.

Accelerating data relocation: Both TTFlash [29] and FastGC [28] depend on intra-plane copyback operation to bypass the SSD controller and migrate data within the same plane, hence requiring no encoding/decoding operations. PaGC [26] further exploits the parallelism at the plane level: it triggers GC in all planes at the same time, ensuring that pages with the same offset can be relocated in parallel. By doing this, PaGC shortens the overall page relocation time of the GC operations. DCD [19] stores data in dual-mode (i.e., SLC-mode and MLC-mode). It prefers reclaiming the SLC block in GC operations and exploits the shorter operation delays of the SLC-mode to accelerate the data relocation. DGCB [15] extends the copyback operation by relocating pages from the source flash chip to another chip within the same channel. Also, it utilizes the shared I/O path and the cache register in the flash controller to hide the data transfer latency during the relocation. By adding a hardware structure to the flash controller, ECB [6] performs the error detection and the data correction in the flash controller, which avoids transferring data to the SSD controller, thereby reducing the data relocation latency.

As a comparison, ParaGC exploits the inter-channel parallelism to accelerate the data relocation.

VI. Conclusion

This paper presents ParaGC, an approach that seeks to accelerate the GC process for 3D flash memory via exploiting the inter-channel parallelism. The main idea behind ParaGC is to relax the constraint of the conventional GC process and carefully disperse the relocation traffic across channels. We implement ParaGC and conduct extensive experiments, showing that ParaGC can greatly cut down the GC latency while improving the read and write performance for a wide range of real-world traces.

References

[1] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of A Large-Scale Key-Value Store. In *Proc. of ACM SIGMETRICS*, 2012.

[2] P. Bodik, A. Fox, M. Franklin, M. Jordan, and D. Patterson. Characterizing, Modeling, and Generating Workload Spikes for Stateful Services. In *Proc. of ACM SoCC*, 2010.

[3] Y. Cai, S. Ghose, E. Haratsch, Y. Luo, and O. Mutlu. Error Characterization, Mitigation, and Recovery in Flash-Memory-Based Solid-State Drives. *Proceedings of the IEEE*, 105(9):1666–1704, 2017.

[4] H. Chang, C. Ho, Y. Chang, Y. Chang, and T. Kuo. How to Enable Software Isolation and Boost System Performance with Sub-Block Erase over 3D Flash Memory. In *Proc. of CODES+ISSS*, 2016.

[5] L.-P. Chang and T.-W. Kuo. An Efficient Management Scheme for Large-Scale Flash-Memory Storage Systems. In *Proc. of ACM SAC*, 2004.

[6] W. Chang, Y. Lim, and J. Cho. An Efficient Copy-Back Operation Scheme Using Dedicated Flash Memory Controller in Solid-State Disks. *Proc. of the International Journal of Electrical Energy*, 2(1), 2014.

[7] S. Chen, Y. Chen, H. Wei, and W. Shih. Boosting The Performance of 3D Charge Trap NAND Flash with Asymmetric Feature Process Size Characteristic. In *Proc. of DAC*, 2017.

[8] T. Chen, Y. Chang, C. Ho, and S. Chen. Enabling Sub-Blocks Erase Management to Boost The Performance of 3D NAND Flash Memory. In *Proc. of DAC*, 2016.

[9] G. Cormode. Count-Min Sketch., 2009.

[10] G. Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[11] J. Cui, Y. Zhang, J. Huang, W. Wu, and J. Yang. ShadowGC: Cooperative Garbage Collection with Multi-Level Buffer for Performance Improvement in NAND Flash-Based SSDs. In *Proc. of DATE*, 2018.

[12] C. Gao, M. Ye, Q. Li, C. Xue, Y. Zhang, L. Shi, and J. Yang. Constructing Large, Durable and Fast SSD System via Reprogramming 3D TLC Flash Memory. In *Proc. of IEEE/ACM MICRO*, 2019.

[13] H. Gong, Z. Shen, and J. Shu. Accelerating Sub-Block Erase in 3D NAND Flash Memory. In *Proc. of IEEE ICCD*, 2021.

[14] 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. of ACM ICS*, 2011.

[15] J. Jeong and Y. H. Song. A Technique to Improve Garbage Collection Performance for NAND Flash-Based Storage Systems. *IEEE Transactions on Consumer Electronics*, 58(2):470–478, 2012.

[16] J. Kim, A. J. Hong, S. M. Kim, K.-S. Shin, E. B. Song, Y. Hwang, F. Xiu, K. Galatsis, C. O. Chui, R. N. Candler, et al. A Stacked Memory Device on Logic 3D Technology for Ultra-High-Density Data Storage. *Nanotechnology*, 22(25):254006, 2011.

[17] C. Lee, T. Kumano, T. Matsuki, H. Endo, N. Fukumoto, and M. Sugawara. Understanding Storage Traffic Characteristics on Enterprise Virtual Desktop Infrastructure. In *Proc. of ACM SYSTOR*, 2017.

[18] E. Lee, J. Kim, H. Bahn, S. Lee, and S. Noh. Reducing Write Amplification of Flash Storage Through Cooperative Data Management with NVM. *ACM Transactions on Storage*, 13(2):1–13, 2017.

[19] S. Li, W. Tong, J. Liu, B. Wu, and Y. Feng. Accelerating Garbage Collection for 3D MLC Flash Memory with SLC Blocks. In *Proc. of ICCAD*, 2019.

[20] Y. Li and K. Quader. NAND Flash Memory: Challenges and Opportunities. *Computer*, 46(8):23–29, 2013.

[21] C. Liu, J. Kotra, M. Jung, and M. Kandemir. PEN: Design and Evaluation of Partial-Erase for 3D NAND-Based High Density SSDs. In *Proc. of USENIX FAST*, 2018.

[22] Y. Luo, S. Ghose, Y. Cai, E. Haratsch, and O. Mutlu. Improving 3D NAND Flash Memory Lifetime by Tolerating Early Retention Loss and Process Variation. In *Proc. of ACM SIGMETRICS*, 2018.

[23] H. Maejima, K. Kanda, S. Fujimura, T. Takagiwa, S. Ozawa, J. Sato, Y. Shindo, M. Sato, N. Kanagawa, J. Musha, et al. A 512Gb 3b/Cell 3D Flash Memory on A 96-Word-Line-Layer Technology. In *Proc. IEEE ISSCC*, pages 336–338. IEEE, 2018.

[24] D. Narayanan, A. Donnelly, and A. Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. In *Proc. of USENIX FAST*, 2008.

[25] E. C. Oh and J. Kong. Nonvolatile Memory Device and Sub-Block Managing Method thereof, Feb. 24 2015. US Patent 8,964,481.

[26] 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. of IEEE SC*, pages 561–572, 2016.

[27] R. Stoica and A. Ailamaki. Improving Flash Write Performance by Using Update Frequency. *Proc. of the VLDB Endowment*, 6(9):733–744, 2013.

[28] F. Wu, J. Zhou, S. Wang, Y. Du, C. Yang, and C. Xie. FastGC: Accelerate Garbage Collection via An Efficient Copyback-Based Data Migration in SSDs. In *Proc. of DAC*, 2018.

[29] S. Yan, H. Li, M. Hao, M. Tong, S. Sundararaman, A. Chien, and H. Gunawi. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *Proc. of USENIX FAST*, 2017.

[30] P. Yang, N. Xue, Y. Zhang, Y. Zhou, L. Sun, W. Chen, Z. Chen, W. Xia, J. Li, and K. Kwon. Reducing Garbage Collection Overhead in SSD Based on Workload Prediction. In *Proc. of USENIX HotStorage*, 2019.