IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 20, 2020, accepted November 24, 2020, date of publication December 7, 2020,
date of current version December 28, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3043123

A Preliminary Study: Towards Parallel Garbage Collection for NAND Flash-Based SSDs

GUANGYU ZHU[1], JAEHYUN HAN[2], AND YONGSEOK SON[2]
[1]Department of Computer Science and Engineering, CAU Institute of Innovative Talent of Big Data, Chung-Ang University, Seoul 06974, South Korea
[2]School of Computer Science and Engineering, Chung-Ang University, Seoul 06974, South Korea

Corresponding author: Yongseok Son (sysganda@cau.ac.kr)

This research was supported in part by the Chung-Ang University Young Scientist Scholarship in 2020, and in part by the National Research Foundation (NRF) funded by the Ministry of Education of Korea through the BK21 Plus Program.

ABSTRACT NAND Flash-based solid-state drives (SSDs) have been widely used as secondary storage devices due to their faster access speed, lower power consumption, and higher reliability compared with hard disk drives. However, application I/O performance can be significantly affected by garbage collection (GC) inside SSDs. For example, a GC operation usually moves valid pages from a victim block into a clean block in a serialized manner. Thus, it increases the GC time and affects the application I/O performance. To address this issue, this article presents a preliminary study on a parallel GC scheme for flash-based SSDs. In our scheme, we parallelize valid page migrations during a GC operation to reduce the total GC time. To do this, we first propose a new flash chip architecture that enables valid page migrations in parallel. Second, we collect information such as new addresses for the migration of valid pages by considering the restriction of SSD operations. Finally, we employ multiple worker threads to migrate the valid pages using the collected information in a parallel manner. We implement and evaluate our scheme using Disksim with Microsoft SSD extension. The experimental result shows that the proposed scheme reduces the overall GC time by 70% and 57% on average compared with the existing scheme and a state-of-the-art GC scheme IPPBE, respectively.

INDEX TERMS Garbage collection, NAND flash memory, parallelism, solid-state drive.

I. INTRODUCTION

Today, NAND flash-based solid state drives (SSDs) have become an important part of storage devices because of their better shock resistance, higher I/O throughput, and lower latency compared with hard disk drives (HDDs) [24], [48]. As a different feature, due to an erase-before-write characteristic of NAND flash memory [4], [47], SSDs perform out-of-place updates. For example, when there is a need to update data in a page, SSDs write new data to a new page and mark the old page invalid. Thus, SSDs perform a garbage collection procedure to reclaim these invalid pages to prepare free space for incoming I/O requests.

Although GC is necessary to reclaim the invalid space, it can lead to performance issues. For example, the response time of some I/O requests can reach up to tens of milliseconds [12] when GC takes place in certain address spaces. This is because it blocks incoming I/O requests to those address spaces which increases the I/O response time.

The associate editor coordinating the review of this manuscript and approving it for publication was Muhammad Zakarya.

To reduce the overhead of the GC procedure, previous studies have proposed different schemes. CA-SSD [27] and BPLRU [16] manage buffers to reduce the amount of data written to flash chips, thereby reducing the overhead of GC. PGC [34] presents a semi-preemptible GC scheme that preempts ongoing GC operations and merges them with pending I/O requests whenever possible. Thus, incoming I/O requests are served along with GC I/O instead of waiting for the completion of the GC procedure. I/O-parallelized GC [9] uses multi-plane advanced commands to handle GC and normal I/O requests in parallel which increases plane utilization. IPPBE [15] is a new advanced command that leverages the structure of NAND flash memory to parallelize block erase operations. Thus, IPPBE can reclaim more invalid pages in a given erase latency. Our study is in line with these approaches [9], [15], [16], [27], [34] in terms of reducing the negative impact of SSD GC. In contrast, we focus on parallelizing page movements in a GC operation by reading valid pages in a victim block and writing them into a new block simultaneously.

In this article, as a preliminary study, we propose a parallel garbage collection scheme for flash-based SSDs, which

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

reduces the amount of time required for a single GC operation. In our scheme, we enable valid page migrations in parallel during a GC operation to reduce total GC time. To do this, we first propose a new flash chip architecture that is organized with additional registers for each die and plane. This enables page read/write operations in parallel within a single plane where a GC operation occurs. Second, we collect the information of valid pages to be migrated by considering the restriction of SSD operations. Finally, we employ multiple workers (threads) to migrate the valid pages using the collected information in a parallel manner.

We implement and evaluate our scheme using the Microsoft Research (MSR) SSD extension [1] with Disksim [6]. The experimental results show that our proposed scheme reduces the overall GC time by up to 70% and 57% compared with the existing GC scheme and a state-of-the-art scheme (IPPBE [15]), respectively, under real-world workloads when using four migration workers.

In summary, the existing GC scheme moves valid pages from a victim block into a clean block in a serialized manner which increases the GC time and affects the SSD performance. To alleviate this performance degradation, we propose a parallel GC scheme to reduce the amount of time required for a single GC operation. The contributions of our work are as follows:

- We propose a flash chip architecture that enables a parallel GC procedure inside SSDs by adding registers for each die.
- We design and implement a parallel garbage collection scheme for flash-based SSDs to reduce the total GC time.
- The experimental results show that our scheme can reduce the total GC time by 70% on average compared with the existing scheme under real-world workloads. And our scheme outperforms a state-of-the-art GC scheme IPPBE by 57%.

The rest of this article is organized as follows: Section II discusses the related work. Section III describes the background and motivation. Section IV presents the design and implementation of the proposed scheme. Section V shows the experimental results. Section VI concludes this article.

II. RELATED WORK

A. LEVERAGING DATA LOCALITY

CA-SSD [16] and BPLRU [27] are buffer management schemes that use data locality to reduce the access number to flash chips. VBBMS [13] is a virtual-block based buffer management algorithm that uses of both temporal and spatial localities by putting random requests and sequential requests into different buffer regions. Our study is in line with these works [13], [16], [27] in terms of reducing the performance degradation by GC. In contrast, we focus on the GC operation instead of optimizing the buffer management.

B. HOT/COLD DATA SEPARATION

Lee and Kim [33] perform an empirical study that shows the hot/cold data separation policies which can reduce the GC overhead. Stoica and Ailamaki [46] proposed an update frequency-based data placement algorithm. This algorithm reduces the number of valid pages within a victim block, which reduces the GC overhead. Thermo-GC [54] is a cost-effective page hotness identifier that leverages the valid page information in a GC victim block to estimate page hotness and to reduce the lifetime variance of pages in a block. Isolation [56] separates hot/cold data by treating the valid pages in GC victim blocks as cold data. It manages one more free block per plane and puts valid pages in victim blocks and user write request data into different free blocks. LFGC [52] improves durability of flash memory by tracking the number of erase operations of each dirty block and selects blocks with less erasures as victim blocks. It separates cold and hot valid pages during GC, which reduces the page copy overhead. Our study is in line with these works [33], [46], [52], [54], [56] in terms of improving SSD performance. In contrast, we focus on improving the internal GC procedure.

C. USING PREDICTION

EIGC [35] introduces a grey prediction model to determine the number of victim blocks that should be selected for next GC. This strategy provides better separation of cold and hot data which reduces the cleaning cost and the number of GC. DYGC [36] is a dynamic GC scheme that uses exponential weighted moving average to estimate the hotness of each valid page in victim blocks. This helps to better separate cold pages from hot pages. Yang *et al.* [55] proposed a scheme to predict the future temperature of data by using LSTM. This scheme uses K-Means to dispatch data to different blocks according to the temperature, thereby reducing the data copy overhead in GC. Our study is in line with these works [35], [36], [55] in terms of reducing valid page migration overhead. In contrast, we focus on accelerating page migration process itself.

D. SEPARATING/SUSPENDING GC

PGC [34] splits the GC procedure into distinct operations according to preemption points. At each point, the context of GC can be saved so that incoming requests can be served first. HIOS [22] is a host interface I/O scheduler that redistributes the GC overheads across non-critical I/O requests. The GC procedure of HIOS is divided into multiple steps and are processed between I/O requests. Kim *et al.* [29] proposed two erase suspension mechanisms that leverage the iterative erase mechanism used in modern SSDs. These mechanisms allow a read operation to interrupt an ongoing erase operation, thereby reducing read tail latency during GC. Our study is in line with these approaches [22], [29], [34] in terms of improving GC operation. In contrast, we focus on paralleling read and write operations of the GC procedure.

E. NEW SSD INTERNAL TECHNOLOGY

TTFlash [53] resolves the tail latency problem caused by GC by using a redundant array of independent NAND (RAIN) and a powerful controller with a large capacitor-backed RAM. Read requests to chips that under the GC process will not be blocked by taking advantage of the redundant data. Write requests can put data into capacitor-backed RAM and then respond to the host instead of waiting for GC to finish. Our study is in line with TTFlash [53] in terms of improving GC process. In contrast, we focus on improving the GC procedure by paralleling the read and write operations.

F. LEVERAGING PARALLELISM INSIDE SSDs

VBP-FAST [17] virtually makes a logical block span several channels. Therefore, data in a virtual block is actually placed on multiple physical blocks belonging to different channels, dies, and planes. This allows VBP-FAST to use the internal parallelism of SSDs across different planes to improve performance. I/O-parallelized GC [9] focuses on a resource wasting issue that occurs during GC, called plane under-utilization problem. During GC, except for the plane that reclaims invalid pages, other planes in the same die are idle. I/O-parallelized GC uses the multi-plane advanced command to serve GC and normal I/O requests in parallel which increases the plane utilization during GC. IPPBE [15] is a new advanced command that leverages the structure of NAND flash to parallelize the block erase operation. The triple-well NAND flash plane architecture allows multiple blocks to be selected during the erase operation. And erasing more than one block does not incur additional time spent performing the operation. By erasing multiple blocks simultaneously, more space can be reclaimed during an erase operation. Our study is in line with VBP-FAST [17], I/O-parallelized GC [9], and IPPBE [15] in terms of leveraging parallelism inside SSDs. In contrast, we focus on utilizing the page-level parallelism.

Even though the previous studies handle the performance degradation problem induced by GC procedure, they do not focus on exploiting page-level parallelism to reduce the amount of time required for a single GC procedure. In contrast, our study exploits the page-level parallelism inside a flash plane to accelerate valid page migration of GC, which improves the single GC procedure.

III. BACKGROUND AND MOTIVATION

A. SSD INTERNALS

This section describes an overview architecture of modern NAND flash SSDs [9], [49], [53] as shown in Fig. 1. Fig. 1a shows main components of SSDs. In general, SSDs have four main components:

- **Host interface logic (HIL).** HIL connects the host and the SSD together. It receives I/O requests from a host, passes them to the SSD, and sends responses to the host. Common interfaces are SATA and PCIe.
- **Controller.** This is a core component that processes I/O requests and manages flash translation layer (FTL), bad blocks, error-correcting code (ECC), etc. The FTL is a software layer that is responsible for mapping logical block addresses to physical flash addresses, wear-leveling, and handling garbage collection.
- **DRAM.** It is attached to the controller and provides space for the address mapping table. It is also used as read/write buffers.
- **Flash chips.** These are the storage medium. There are single-level cell (SLC) flash chip and multi-level cell (MLC, TLC, and QLC) flash chip. The multi-level cell can store more bits per cell which leads to higher storage density, whereas has much longer read/write latency.

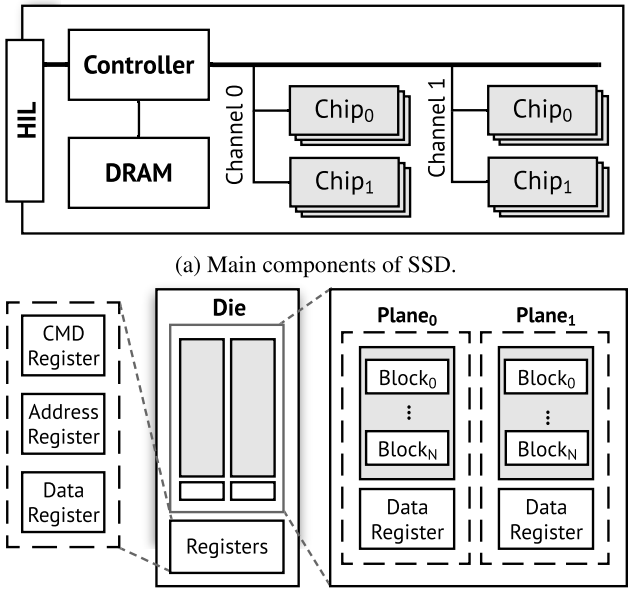


FIGURE 1. Overview architecture of modern NAND flash-based SSDs.

To increase storage density, flash manufactures connect several flash chips together through a shared I/O bus to consist a channel. Fig. 1b shows the internal architecture of a flash chip. As shown in the figure, each chip contains one or more dies and has its own set of ready/busy signals [19]. Each die is composed of multiple planes (typically two planes ($\texttt{Plane}_0$, $\texttt{Plane}_1$)) and has its own address, command, and data registers, which allows dies to operate independently [9]. In other words, each die independently processes normal I/O and GC operations. A plane is divided into thousands of blocks ($\texttt{Block}_0$ to $\texttt{Block}_N$) and there are one or two data registers used as I/O buffers for each plane. A block typically consists of 64 or 128 pages. The size of a page is usually between $4KB$ to $16KB$.

B. GARBAGE COLLECTION

Due to the erase-before-write characteristic of SSDs, a flash page can be written only if the page is erased (clean). To avoid the need to erase old pages during write operations, which increases latency, SSDs adopt an out-of-place write policy. When a logical page is updated, its data is written to a

Algorithm 1 Existing GC Algorithm (Valid Pages in a Victim Block Are Read and Written From/to the Destination Block in Serialized Manner)

```
Input: GC victim block (blk)
1: procedure GARBAGE COLLECTION(blk)
2:     for V ← each valid page in blk do
3:         issue a single read (V);
4:         issue a single write (V);
5:     end for
6:     issue block erase (blk);
7: end procedure
```

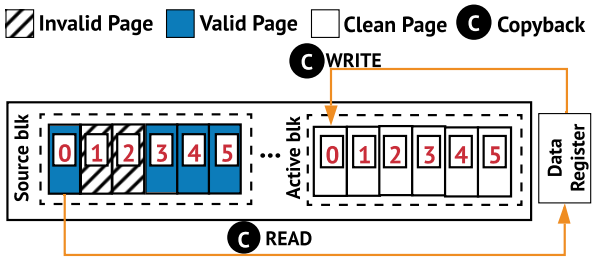


FIGURE 2. A procedure of copyback command in a single plane.

different physical page. To do this, a Flash Translation Layer (FTL) remaps logical pages to new physical pages while the original physical pages are marked as invalid. This enables SSDs to accept the write requests without immediately performing erase operations.

The FTL keeps tracking the number of free pages within a plane and maintains a free (active) block for incoming requests. Once the number of free pages drops below a given threshold, GC takes place to erase some victim blocks for reclaiming the invalid pages as described in Algorithm 1. First, a victim block is selected for GC. Once a GC victim block is selected, each valid page in it is read and written to the free page of the destination block in serial. Finally, the selected victim block is erased.

When valid pages are read or written during GC, the data must be transferred to the data register at first. Then, the data is transferred in/out via shared buses. During the transmission, the shared buses are blocked to avoid data corruption. The blocking of shared buses delays other requests from being serviced along these paths. Since SSDs always need to guarantee an amount of free space for correct operations, GC has the highest priority and incoming normal I/O requests can be blocked during GC operations.

Generally, there are two types of approaches which are global and local approaches [9]. In the global approach, valid pages in a victim block are moved to an active block in different chips/dies/planes via read/write commands. In the local approach, valid pages in a victim block are moved to an active block in the same plane using the *copyback* advanced command.

Fig. 2 shows the procedure of execution of a copyback command in a single plane when a data register is used. As shown in the figure, there are six pages (i.e., $\texttt{Page}_0$ to $\texttt{Page}_5$) in a block (i.e., `Source blk`). The copyback operation reads $\texttt{Page}_0$ from `Source blk` to the data register. Then, it writes the data from the data register to $\texttt{Page}_0$ in an active block (i.e., `Active blk`) in the same plane. Since copyback only works inside a plane, it does not block the shared buses connecting other dies and channels during GC operations. In this article, we will present our scheme based on the local GC first, then we discuss how our scheme can work based on the global GC.

C. PARALLELISM INSIDE SSDs

There are four levels of parallelism in SSDs: channel-level, chip-level, die-level, and plane-level. Exploiting parallelism at each level can increase SSD performance. For example, as shown in Fig. 1a, when two channels (i.e., $\texttt{Channel}_0$ and $\texttt{Channel}_1$) can serve a request which includes more than one page simultaneously, we use channel-level parallelism. When a request can be served by $\texttt{Chip}_0$ and $\texttt{Chip}_1$ of $\texttt{Channel}_0$ simultaneously, we benefit from chip-level parallelism [19]. In addition, inside a flash chip, each die has its own address, command, and data registers, which enables die-level parallelism. Within a die, all planes share the same address and command registers, but each plane has its own data register. To exploit parallelism at the plane-level, we need to use multi-plane advanced commands [50].

In summary, a single GC operation in the existing scheme moves valid pages from victim blocks to other blocks in the serialized manner, which cannot exploit the page-level parallelism and has a relatively long GC time. This decreases the performance of the SSD because the GC operation blocks incoming normal I/O requests. In this article, we enable page-level parallelism inside each plane by devising a new flash architecture for an efficient and parallel GC procedure. Our GC scheme leverages page-level parallelism by migrating valid pages via multiple read and write operations in parallel for a GC operation. Thus, this scheme can accelerate overall GC operations.

IV. DESIGN AND IMPLEMENTATION

A. OVERVIEW AND ASSUMPTION

This section describes an overview and assumption for our preliminary study. As we mentioned, because of the limitation of current SSD internal architecture, the existing GC scheme can only read/write one valid page from/into a single plane. Even though previous state-of-the-art techniques are effective in reducing the negative impact of GC, the improvement of individual GC processes is still necessary.

In terms of the number of valid pages in a victim block, a previous study [41] has shown the occupancy rate of valid pages in victim blocks can reach more than 40% under high disk utilization in the real-world workloads. This makes migrating valid pages in victim blocks a huge burden in the GC process. Furthermore, when using multi-bit per cell flash chips (e.g., TLC and QLC) that have longer read/write

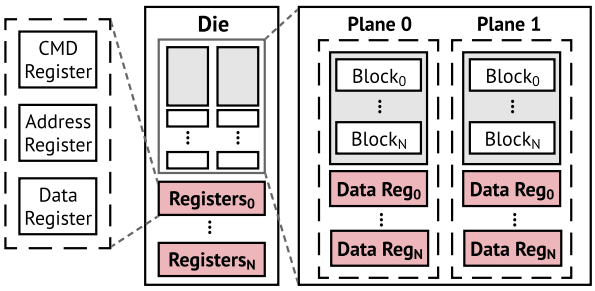


FIGURE 3. New flash chip architecture (Each die has *N* set of registers and each plane has *N* data registers).

time compared with single-bit per cell (SLC) chips, the large number of valid pages further increases the time required for GC operations.

Recently, commercial SSDs have been equipped with fairly powerful multi-core controllers [40], [43]. Such multi-core SSD controllers are likely to become more commonplace. Also, host-managed flash [5], which moves the FTL to the host operating system, enables the use of much more powerful host side processors and more flexibility to control SSDs.

In order to utilize high-performance multi-core processors more efficiently and address the performance degradation introduced by GC, in this article, we propose a page-level parallel garbage collection scheme to reduce the time required for GC. First, we propose a new flash architecture that enables page-level parallelism in a single plane. Second, to parallelize the GC works properly, we collect the information such as new addresses for valid pages in victim blocks by considering the restriction of SSD. Finally, we leverage page-level parallelism by enabling multiple worker (threads) to migrate valid pages from victim blocks to other active blocks. In the following subsections, we describe our scheme in detail.

B. OVERALL ARCHITECTURE

To make it possible to achieve parallelism inside the plane (the page-level parallelism), we devise a new flash architecture as shown in Fig. 3. In this architecture, we add register sets for each die and data registers for each plane to enable multiple workers to migrate multiple valid pages. The number of set of registers and data registers are identical. Also, the number of workers can be configured as much as the number of set of registers and data registers. Thus, this architecture enables page-level parallelism inside a single plane.

For example, as shown in the figure, in the die, there are N sets of registers ($\texttt{Register}_0$ to $\texttt{Register}_N$). A set of registers consists of one command register, one address register, and one data register. For each plane ($\texttt{Plane}_0$, $\texttt{Plane}_1$), there are N data registers ($\texttt{Data Reg}_0$ to $\texttt{Data Reg}_N$). In this case, up to N workers can perform valid page migration in parallel. With this modification, the commands, addresses, and data of multiple operations can be sent to the same die. They are interleaved because the dies in the same chip share the internal bus of the chip. Thanks to the high speed of NV-DDR3 interface which can reach up to $1600MT/s$ [2], [50], the interleaving time is negligible.

We note that the number of registers should be carefully chosen by considering the internal resources of SSDs and workloads. Since the computing power and bandwidth of the interconnect bus are limited resources shared by all processes within an SSD, unbalanced resource allocation for migration workers may reduce the GC time but stall normal I/O requests. Thus, rather the total performance of SSD can be decreased. In our article, the evaluation section shows the efficiency according to the number of workers and workloads.

C. FEASIBILITY AND COST OF THE NEW FLASH ARCHITECTURE

1) FEASIBILITY OF ADDING REGISTERS

Real products have already used additional registers to improve SSD performance. For example, many flash memory chips like [38] already have two registers (one data register, one cache register) per plane to implement a cache programming/reading mechanism. Regarding the feasibility of adding registers in each die, there are some flash chips like [26] which use several data registers for reducing read errors. Based on previous studies [26], [38], adding registers to plane and die can be feasible.

2) THE COST OF ADDING NEW HARDWARE

Although there is no physical prototype, we can roughly estimate the cost of adding registers. In terms of space, if we add registers by using SRAM to a 3D TLC NAND flash chip, the space cost can be estimated as follows. A study [26] presents a 3D TLC NAND memory flash chip with a high areal density up to $3.98GB/mm^2$. The chip has one die, and the die size is $128mm^2$. There are two planes in the die, and each plane has 2874 blocks. One block consists of 768 pages of which size is 16 KB. The density of SRAM is about $4.75MB/mm^2$ [45]. Thus, the size of a data register would be $3.4 \times 10^{-3}\ mm^2$. To enable four workers running in the chip, we need to add 9 data registers, 3 command registers, and 3 address registers. The size of command and address registers are not specified in the standard [50]. If we assume all three kinds of registers have the same size, the total size of added registers is around $5.1 \times 10^{-2}\ mm^2$. And, the size of added registers accounts for 0.4% of the die.

The monetary cost caused by the new hardware is mainly composed of two parts, one is the cost of added register, and the other is the cost of peripheral circuits linking these registers with other components. In terms of the cost of the added registers, we can estimate it from the price of SRAM. Because price of SRAM is about 500$ per GB [7], when we use SRAM to build the register, the cost of one 4KB register is about 0.002$. To enable four workers on each die, we need to add at most 12 data registers. This costs about us 0.024$ per die. The cost of peripheral circuits that link registers with other components is hard to estimate because it is related

to the specific layout of the architecture. Thus, we leave to evaluate this cost as future work.

3) MULTI-PAGE READ/PROGRAMMING WITHIN A BLOCK

New data can only be written after the last data written in a block used to be a well-known limitation of NAND flash. However, there are some existing flash architectures [18], [26], [28], [30] that can support multi-page read/programming simultaneously in a single block. Especially, [18] presents a multiple pages programming scheme like our scheme by adding data register in the plane. They present an architecture that comprises a bit line select gate bank. The bit line select gate bank contains multiple bit line select gates. The bit line select gates allow a page buffer to be coupled to multiple bit lines, which allows multiple pages to be programmed and read at the same time. Thus, this enables multi-page programming/read in a single block by using the multiple bit lines simultaneously. Based on previous studies, we believe it is possible to read and write several pages in parallel at the same block. Consequently, our scheme can be feasible and applied to flash architecture.

D. PARALLEL GARBAGE COLLECTION

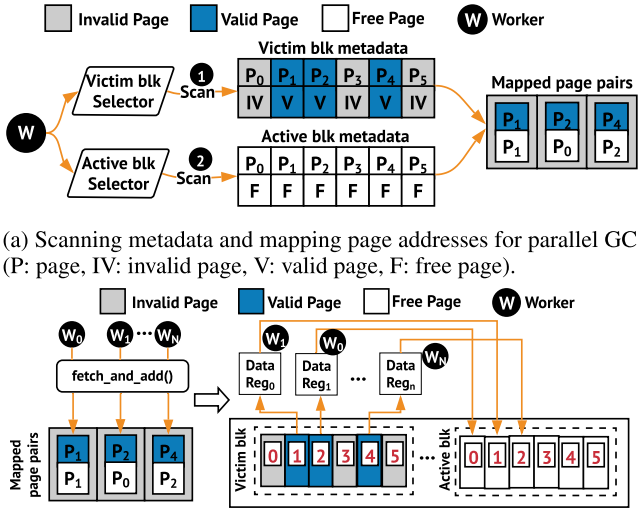
The existing GC procedure reads and writes the valid pages in a victim block to a new block one by one sequentially. To improve the existing procedure, we enable parallel garbage collection by leveraging page-level parallelism based on the proposed SSD architecture. To do this, we first collect the addresses of valid pages and free pages by scanning the metadata of the victim block and the active block. Then, we match valid page addresses and their corresponding free page addresses by considering the restriction of SSDs. This allows our parallel GC procedure to be performed properly. Second, based on the collected address information, we adopt multiple workers to read valid pages and write them into free page addresses in parallel. This reduces the total time required for the GC procedure. In this article, we introduce our idea based on a page-mapping FTL. We note that the same idea can also be used on hybrid-mapping FTLs because our scheme focuses on accelerating the valid page migration on physical flash blocks, which has no effect on other parts of the FTL. In this section, we describe how our scheme works based on the local GC[1] as follows (We will discuss how our scheme can work with global GC[2] in the discussion subsection).

1) COLLECT INFORMATION CONSIDERING THE RESTRICTION OF SSDs

During our GC process, we select a victim block depending on a victim block select policy. (we use a greedy strategy [8] in this article). After the victim block is selected, we collect information (e.g., addresses of valid pages and free pages) in advance to enable the copyback advanced command to

[1]The victim block and the destination block are in the same plane and pages are migrated by copyback commands.

[2]The victim block and the destination block are in different chips, dies, and planes and the pages are migrated by read/write commands.



(a) Scanning metadata and mapping page addresses for parallel GC (P: page, IV: invalid page, V: valid page, F: free page).

(b) Copyback command that runs in parallel on a multi-register plane.

FIGURE 4. Parallel garbage collection.

read/write pages within a single plane in parallel. During this procedure, there is a restriction that we need to handle. In some SSDs, the copyback command has an odd/even page restriction. This means the contents need to be written into an odd page when reading from an odd page [9], [19], [42], [44], [50].

To handle this restriction, when creating pairs of valid and destination page addresses, we map the destination page address based on the valid page address to satisfy the restriction. Also, the destination page with the lowest address is selected first.

Fig. 4a shows the procedure of metadata scanning for parallel GC. As shown in the figure, there are three valid pages ($page_1$, $page_2$, and $page_4$) and three invalid pages ($page_0$, $page_3$, and $page_5$) in a victim block (*victim blk metadata*). In this case, since $page_1$ (a valid page) is an odd page in the victim block, we map the page to $page_1$ (a free page) in an active block as its destination page although $page_0$ in the active block is still free. Similarly, we pair $page_2$ and $page_4$ (valid pages) in the victim block with $page_0$ and $page_2$ (free pages) in the active block, respectively. Finally, we get three pairs of addresses ($< 1, 1 >$, $< 2, 0 >$, and $< 4, 2 >$). Here, we only use one worker to scan the metadata because the number of pages in a block is typically between 64 and 512. The time required to scan the metadata of two blocks (victim block and active block) is trivial.

Scanning valid pages in the victim block and mapping destination pages for valid pages are required for both existing and our proposed GC schemes. The existing scheme performs address mapping every time a valid page is moved. Meanwhile, our scheme first performs all address mappings for valid pages in a victim block, then moves valid pages. Therefore, the overhead introduced by the scanning and mapping in our scheme is similar to the existing scheme. The difference for the mapping operations in the two schemes only lies in the time point of doing page mapping.

2) PARALLEL GC PROCESS

As described above, we scan all pages in a block to find valid pages and match new destination pages for them. After this point, we create multiple workers to migrate the valid pages in a parallel manner. Each worker fetches a pair of addresses and moves the valid pages to the selected free pages.

Fig. 4b shows multiple workers which performs copyback commands in parallel. As shown in the figure, N workers are created for valid page migration (the number of workers to be created depends on the processor and flash architecture as mentioned in section IV-B). The workers (W_0, W_1, and W_N) first fetch address pairs to indicate the direction of migration. These address pairs are stored in a shared array. To allocate an address pair for each worker properly and concurrently, we use an atomic instruction (`fetch_and_add()` [14], [21]) that increments the index of the array without a locking mechanism. This ensures that each worker gets a different address pair and no address pair is missed.

For example, a worker (W_0) fetches an address pair ($< 1, 1 >$) and increases the index of the array using `fetch_and_add()` instruction, then calls a copyback command to read a page (P_1) in the victim block into a data register (`Data Reg`$_1$). The worker then writes the content to a page (P_1) in the active block in the same plane. Simultaneously, another worker (W_1) fetches address pair ($< 2, 0 >$), increases the index, and calls a copyback command to begin its page migration task. The remaining workers obtain pairs of addresses in the same manner and complete the transfer of valid pages. After all valid pages in the victim block are transferred, we can erase the block and complete the GC procedure. Consequently, this parallel GC process can accelerate the migration of valid pages via multiple workers to read and write the pages in parallel.

E. IMPLEMENTATION

We implement our scheme by modifying the GC procedure of the MS SSD extension [1] to enable multiple workers to migrate the valid pages in parallel. Algorithm 2 shows a procedure of the proposed parallel GC. As shown in the algorithm, similar to the existing GC procedure, we first fetch the metadata of a selected victim block (`victim_blk`) for the plane that need to perform GC (Algorithm 2, line 2). The metadata includes the page addresses and states of pages in the victim block. After then, we begin to move the valid pages and erase victim blocks via `CLEAN_BLOCK()` function with the selected victim block (Algorithm 2, line 3).

In the block cleaning process, we first change the state of the die to busy. Thus, other requests to the same die will be blocked (Algorithm 2, line 7). Then, we check the metadata of the victim block to know whether there is any valid page or not (Algorithm 2, line 8). If there is no valid page in the victim block, the block will be erased directly, the metadata will be updated, and the GC procedure will be completed (Algorithm 2, lines 9-13). Otherwise, if there is any valid page, we start to prepare migration of the valid pages. We first create an array (`address_pairs`) to store the pairs of valid page and the destination page address (Algorithm 2, line 15). Then, we create an array (`free_pages_info`) to collect the addresses of free pages in the active block (Algorithm 2, line 16). After then, we find and scan an active block in the plane and store the addresses of the free pages in the array (`free_pages_info`) (Algorithm 2, 17). For each valid page, we fetch its address, map its corresponding destination page addresses, and store the pairs of valid and free page addresses into the array `address_pairs` (Algorithm 2, lines 18-24).

Algorithm 3 shows how we map a destination page for each valid page in a victim block. As shown in the algorithm, we match the odd/even free pages starting from the lowest address to valid pages according to the addresses of the valid and free pages (Algorithm 3, lines 2-6). After getting the valid pages and corresponding destination pages, we create

Algorithm 2 Simplified Pseudo-Code of Parallel Garbage Collection

```
1:  function GARBAGE_COLLECTION(plane_num, chip_num)
2:      victim_blk = SELECT_VICTIM_BLK( );
3:      CLEAN_BLOCK(victim_blk, plane_num, chip_num);
4:  end function
5:
6:  function CLEAN_BLOCK(victim_blk, plane_num, chip_num)
7:      change the state of corresponding die from idle to
    busy;
8:      no_valid_pages = CHECK_INFO(victim_blk);
9:      if (no_valid_pages) then
10:         ERASE_BLOCK( );
11:         update metadata;
12:         return
13:     end if
14:
15:     address_pairs[# of pages per block];
16:     free_pages_info[# of pages per block];
17:     SCAN_ACTIVE_BLOCK_INFO(free_pages_info,
    plane_num);
18:     for (i = 0; i < pages_per_block; i++) do
19:         page_addr = victim_blk.page[i];
20:         if (page_addr is valid) then
21:             dest_addr =
    MAP_FREE_PAGE(free_pages_info, page_addr);
22:             Put page_addr, dest_addr into
    address_pairs;
23:         end if
24:     end for
25:     for (i = 0; i < NUM_WORKERS; i++) do
26:         WORKER_CREATE(MOVE_PAGE_PER_
    WORKER, &address_pairs);
27:     end for
28:     Waiting for workers to finish;
29:     ERASE_BLOCK( );
30:     update metadata;
31: end function
```

Algorithm 3 Simplified Pseudo-Code of Address Mapping Process

```
1: function MAP_FREE_PAGE(pages_info, source_page)
2:     if source_page.address % 2 == 1 then
3:         return FIRST_FREE_PAGE(pages_info, odd);
4:     else
5:         return FIRST_FREE_PAGE(pages_info, even);
6:     end if
7: end function
```

Algorithm 4 Simplified Pseudo-Code of per Worker

```
1: function MOVE_PAGE_PER_WORKER(address_pairs)
2:     while (true) do
3:         local_index = fetch_and_add(curr_index, 1);
4:         if (local_index ≥ number of valid pages) then
5:             break;
6:         end if
7:         address_pair = address_pairs[local_index];
8:         MOVE_ONE_PAGE(address_pair);
9:     end while
10:    worker_exit();
11: end function
12:
13: function MOVE_ONE_PAGE(address_pair)
14:    COPYBACK(address_pair);
15:    update metadata;
16: end function
```

workers to perform migration in parallel (Algorithm 2, lines 25-27). The number of workers created depends on the number of data registers for each plane.

Algorithm 4 shows how each worker performs its page migration task. As shown in the algorithm, we use a global variable (`curr_index`) that indicates the current address pair and a local variable (`local_index`) that indicates the address pair of each worker. To ensure that each worker receives a different address pair concurrently, we use the `fetch_and_add()` atomic operation when each worker accesses the address pairs in the array `address_pair` (Algorithm 4, line 3). Thus, each worker increments `curr_index` concurrently and obtains its own pair index. Workers keep performing the tasks until all address pairs in the array `address_pairs` are processed (Algorithm 4, lines 4-6).

Each worker accesses the array (`address_pairs`) to fetch an address pair (`address_pair`) using its own local index (`local_index`) (Algorithm 4, line 7). Then, each worker calls the `COPYBACK` command to copy the data from a valid page to its new address (Algorithm 4, line 14) and updates the metadata according to the results of migration (Algorithm 4, line 15). After all workers finished their operation, the victim block is erased (Algorithm 2, line 29) and the metadata is updated according to the erased block (Algorithm 2, line 30). At this point, the entire GC procedure is completed.

F. DISCUSSIONS

1) COMPLEXITY OF PROPOSED GC SCHEME

Our idea is based on utilizing parallel processing for improving GC performance. Our scheme is relatively simple and efficient, but there are some challenges. The main challenge is to perform parallel operations correctly while improving performance. For example, our scheme collects the information for the parallel operations in advance and properly schedules parallel GC workers with little additional cost. The required information consists of three parts: the number of valid pages in the victim block, the addresses of valid pages, and the addresses of free pages (destination). This information is collected by iterating page metadata in the victim and the free blocks. The complexity of iterating page metadata is $O(n)$, where n is the number of pages in a block. In addition, our scheme considers some limitations of certain NAND flash memory chips, such as the odd/even page restriction of copyback command. Given the odd/even page restriction of copyback command in some flash chips, we determine the odd/even parity of an address when matching valid pages with destination pages as shown in Algorithm 3. When parallel GC workers moving the valid pages into a block, the complexity is $\frac{0(n)}{k}$, where n is the number of valid pages in a victim block and k is the number of workers.

2) PARALLEL GC IMPACT ON NORMAL I/O PERFORMANCE

Our parallel GC scheme reduces the total GC time for SSDs, therefore, it can improve the overall I/O throughput during GC. However, the overall performance of normal I/O requests can depend on the number of GC workers. As described in Section III-A, both normal I/O and GC I/O requests can exist in an SSD simultaneously. Because computing power and shared bus bandwidth are limited resources, when we allocate excessive GC workers, there can be a performance degradation for normal I/O. Therefore, the number of GC workers should be carefully considered according to the resources of specific SSDs. Unfortunately, it is difficult to determine exactly how the number of GC workers affects the performance of normal I/O since our scheme is implemented based on an SSD simulator, which does not simulate the computing resources. Thus, we cannot provide a model for SSD resource allocation to help determine how many workers can optimize the SSD performance. However, as shown in evaluation section, even if only one worker is added to help the task of valid page migration, the total GC time can be reduced significantly.

3) ENABLING PARALLEL GLOBAL GC

Our parallel local GC mechanism can also enable parallel global GC. In the global GC, valid pages in a victim block can be moved to different units such as chips, dies, and planes. Instead of using copyback commands in local GC, the global GC transfers valid pages using normal flash page read/write commands. When transferring the valid pages, the global GC first reads valid pages from a flash chip to the controller, and then dispatches the data to another target SSD unit. With this

global GC, our scheme can perform the read/write commands for the valid pages in parallel. For example, similar to our local GC, we collect information of the target units (e.g., address pairs) in advance to prepare the parallel global GC. Then, each worker executes read commands to move valid pages to controller and executes write commands to write valid pages back to the flash chip according to the collected address pairs.

4) COMBINATION WITH OTHER STUDIES

Our proposed parallel GC is effective in reducing the time required for a GC operation and it can be easily combined with other schemes to achieve better performance. For example, the IPPBE [15] command erases multiple blocks simultaneously. Our scheme can help IPPBE to reduce the time required for copying valid pages inside a block since our scheme is based on page-level parallelism. Thus, combining our scheme with IPPBE can further reduce the total GC time using both block-level and page-level parallelism. I/O-parallelized GC [9] uses multi-plane commands to serve GC operations and blocked normal I/O requests on the planes of a single die simultaneously. With this scheme, we can allow multiple migration workers in our scheme to leverage the multi-plane commands within a die. Therefore, our scheme can accelerate I/O-parallelized GC by reducing total GC time to further improve the performance of SSDs.

V. EVALUATION

In this section, we evaluate our proposed scheme and compare it with existing and recent GC schemes. In subsection V-B, we compare our proposed scheme with an existing GC scheme. In subsection V-C, we compare our scheme with a state-of-the-art GC scheme (IPPBE). In subsection V-D, we discuss how command latency and the number of valid pages can affect the performance of our proposed scheme.

A. EXPERIMENTAL SETUP

To evaluate the performance of our scheme, we implemented our parallel GC using the Microsoft Research's SSD extension [1] with Disksim [6]. This is a sophisticated and trace-driven simulator that is widely used in many studies [9], [34], [51]. In this simulator environment, the experiment results are statically generated based on the SSD organization settings and predefined static time values, such as the time of a read/write command, and the time to transfer data between the processor and flash memory. Thus, the experimental results are always identical if the given SSD settings and workloads are unchanged.

1) SIMULATOR SETTINGS

Table 1 lists the SSD parameters that we used in the simulator. The page read latency is the time that a page read command reads data from a flash memory page to the data register in each plane. Similarly, the page write latency is the time that a page write command writes data from the data register in a plane to a flash memory page. The block erase latency is the time that a erase command erases a flash memory block. We used two sets of parameters. The parameters of the SLC chip were from the MS SSD extension [1] which is collected from a SAMSUNG SSD. The timing information and organization setting for MLC chip was from Micron [38].

TABLE 1. SSD simulator parameters. ("←" indicates the same setting as the SLC chip).

Parameter	SLC chip	MLC chip
Overprovisioned blocks	15%	←
# of channels	4	←
Chips per channel	4	←
Dies per chip	2	←
Planes per die	2	←
Blocks per plane	2048	←
Pages per block	64	256
Page Size	$4KB$	←
Page Read latency	$25\mu s$	$75\mu s$
Page Write latency	$200\mu s$	$1300\mu s$
Block Erase latency	$1.5ms$	$3.8ms$
Cleaning policy	Greedy	←

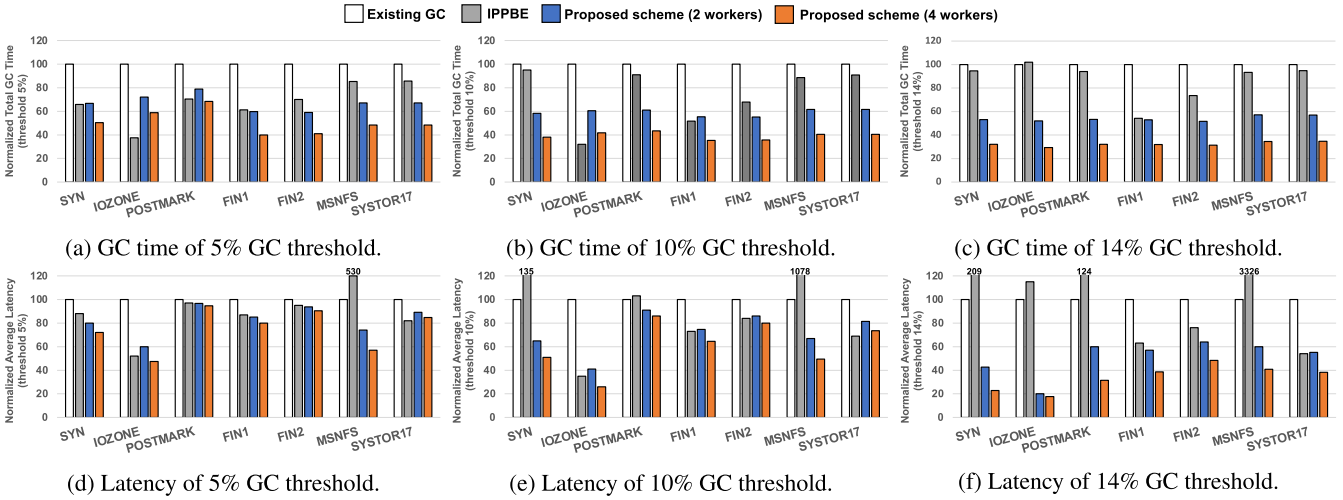
For each simulated SSD, there are four channels and each channel has four chips. There are two dies per chip, two planes per die, and 2048 blocks per plane. For SLC chip, there are 64 pages per block; for MLC chip, there are 256 pages per block. Therefore, the size of SLC SSD is 32GB, and the size of MLC SSD is 128GB. The GC's victim block selection strategy is the greedy algorithm that chooses the block with the least number of valid pages in a plane as the victim block [8]. Thus, this policy can minimize the number of migrations for valid pages during GC. The simulator uses a page-mapping FTL. The FTL scheduler gives GC the highest priority, reads have the second highest priority, and writes have the lowest priority.

The GC threshold determines when GC procedure starts. Thus, the activation of GC procedure depends on the remaining number of free blocks according to the threshold. For example, if we set the GC threshold to 10%, the GC will start running when the number of free blocks available is less than 10%. In our study, we set the value of the GC threshold as 5% and 10% by referencing the previous studies [1], [20], [23], [37], [44]. We set the greatest threshold to 14% because the overprovisioning value in our simulator is set to 15% and the threshold cannot bigger than the overprovisioning value. In summary, we use three GC thresholds to show the performance according to the different thresholds.

We note that the average proportion of valid pages in a victim block can reach more than 40% under high disk utilization in the read-world workloads as shown in a previous study [41]. So, we choose main performance criteria when a GC threshold is 10%. It is because our evaluation shows that the average proportion of valid pages in a victim block is approximately 40% at this GC threshold for more realistic evaluation.

2) WORKLOADS

We use both synthetic and real-world traces to analyze the efficiency of the proposed scheme as shown in Table 2. There



(a) GC time of 5% GC threshold. (b) GC time of 10% GC threshold. (c) GC time of 14% GC threshold.

(d) Latency of 5% GC threshold. (e) Latency of 10% GC threshold. (f) Latency of 14% GC threshold.

FIGURE 5. Performances of existing GC, IPPBE, and our scheme on SLC chip with different GC thresholds.

TABLE 2. Workload parameters.

Workload	Type	%Write	Average Request size
SYN	Synthetic	100%	$6KB$
IOzone	Synthetic	100%	$360KB$
Postmark	Synthetic	16.8%	$222KB$
Financial 1	Real-world	78.3%	$5.2KB$
Financial 2	Real-world	18.6%	$4.8KB$
MSN file server	Real-world	23.6%	$8.7KB$
Systor17	Real-world	19.1%	$40KB$

are random write traces provided by Disksim (SYN) and IOzone (IOZONE) [39]. Postmark (POSTMARK) [25] is a synthetic trace which simulates the behavior of mail servers. Financial 1 (FIN1) and financial 2 (FIN2) are real-world traces that were collected from two large financial institutions [3]. MSN file server (MSNFS) [11] is a real-world trace that was collected from MSN storage server. SYSTOR17 [31] is also a real-world trace that were collected from virtual desktop infrastructure [32]. The workloads consist of write-intensive and read-intensive traces. The workloads of SYN and IOzone are 100% random writes. Financial 1 is also a write-intensive trace that has 78.3% write requests. Postmark, Financial 2, MSN file server, and Systor17 are read-intensive workloads that have 83.2%, 81.4%, 76.4%, and 80.9% read requests, respectively.

3) PRE-CONDITIONING

To evaluate the effect of GC, we need to perform pre-conditioning on SSDs before evaluations. For example, before evaluating each workload, we run a real-world workload which we use in the evaluation to fill the SSD. However, the amount of writes of many traces of real workloads is less than 60 GB [49]. To make it possible to use real-world traces to perform pre-conditioning, we use a method from the previous study [10], [49] to generate pre-conditioning traces. More specifically, we first use the information of read requests (e.g., LBAs, request sizes) to generate write requests to fill the SSD. We use LBAs and request sizes in read requests because the write operation should be performed before reading the data. If the read requests are not enough for steady-state, we then use the write requests of the workloads. If both read and write requests are not enough for pre-conditioning the flash storage space, we shift the LBAs of workloads to generate new write requests to fill the remaining space. This approach makes SSDs into steady-state before evaluation and also keeps the data placement characteristics of each workload.

B. COMPARING EXISTING SCHEME WITH PROPOSED SCHEME

In this section, we compare the performance between the existing GC scheme and our proposed scheme. In this GC scheme, when a victim block is selected, valid pages in the victim block are moved to a clean block one by one in sequence. We present the performance results in terms of the total GC time and the average latency (normal I/O) which are normalized to the existing GC scheme as shown in Fig. 5.

1) TOTAL GC TIME

Fig. 5a, 5b, and 5c show the total GC time under different GC threshold settings. In the case of 5% GC threshold, our scheme reduces the total GC time by 33% and 49% on average, compared with the existing scheme, when using two and four workers, respectively. It is because our scheme performs the GC procedure in parallel instead of a serialized manner, and the performance gap increases as the number of workers increases. With 10% GC threshold, our scheme reduces the total GC time by 43% and 60% on average in the case of two and four workers, respectively. In the case of 14% GC threshold, our scheme further reduces the total GC time by 46% and 70% on average when using two and four workers, respectively. Especially, in this case, four workers reduce the GC time by up to 44% compared with that for two workers since the number of valid pages to be migrated is relatively large.

TABLE 3. The average number of valid pages and its proportion in a victim block under different GC thresholds.

Workloads	threshold: 5%		threshold: 10%		threshold: 14%	
SYN	12.8	20.0%	25.3	30.5%	51.3	80.1%
IOzone	8.0	13.5%	21.6	33.7%	51.8	80.1%
Postmark	4.6	7.2%	19.0	29.7%	49.3	77.0%
FIN 1	21.4	33.4%	33.4	52.2%	49.3	77.0%
FIN 2	20.1	31.4%	31.2	48.8%	53.7	83.9%
MSNFS	17.1	26.8%	28.6	44.7%	52.0	81.3%
SYSTOR17	17.2	26.9%	28.8	45%	51.9	81.3%

This result shows the performance difference according to the GC threshold since the threshold affects the number of valid pages in victim blocks. Table 3 shows the average number of valid pages in a victim block for each workload. As shown in Fig. 5 and Table 3, the total GC time gap between existing and proposed schemes increases as the number of valid pages in victim blocks increases. It is because more migrated valid pages significantly increase the migration time among the total GC time. When the number of valid pages are similar, the performance trend is similar in the both cases of write-intensive and read-intensive workloads.

As the number of workers increases, more valid pages are moved in the same time period, so the performance is also improved. However, the relative proportion of performance improvement decreases as the number of workers increases. When using two workers under 10% threshold, the GC time is reduced by 40% on average. When using four workers under 10% threshold, GC time is reduced by 60% on average. Therefore, the number of workers should be chosen carefully according to the workload and computational power of the target SSDs.

2) AVERAGE LATENCY

Fig 5d, Fig. 5e, and Fig. 5f show average normal I/O latencies under different GC thresholds. In the case of 5% GC threshold, our scheme reduces the latency by up to 40% and 53% compared with the existing scheme when using two and four workers, respectively. In the case of 10% GC threshold, our scheme reduces the latency by up to 59% and 74% when using two and four workers, respectively. With 14% GC threshold, our scheme reduces the latency by up to 80%.

Unlike total GC time, the change of latency is related to the characteristics of workloads. As shown in Fig. 5e, when a workload is write-intensive, the latency is further reduced compared with that of read-intensive workloads. As expected, it is because the write-intensive workload frequently triggers the GC operations.

Reduction of the average I/O latency is also related to the size of the request. Table 2 shows the category and average request sizes for each workload. SYN, IOzone, and Financial 1 are all write-intensive workloads, but the latency reduction of IOzone is the largest among them since the request size of IOzone is much larger than those of the other workloads. Similarly, even though MSNFS is a read-intensive workload, its latency has been well reduced. This is because MSNFS contains some larger requests compared to other traces. Because of the parallelism within SSDs, a write request with a large size will be serviced by multiple dies. This makes the write request with a large size issues several GC procedures in different dies. Therefore, in a given time period, more victim blocks will be reclaimed simultaneously.

C. COMPARING STATE-OF-THE-ART SCHEME WITH PROPOSED SCHEME

In this section, we compare the performance between IPPBE and our proposed scheme. IPPBE shares a similar idea with our scheme in terms of improving the internal parallelism of the SSD plane. The difference is that IPPBE achieves block-level parallelism by erasing multiple blocks in the same plane at the same time, while our scheme achieves page-level parallelism by moving multiple pages in parallel. We set the number of blocks erased at a time to two like shown in IPPBE. Same as the previous section, the results are presented in terms of the total GC time and the average latency which are normalized to the existing GC scheme.

1) TOTAL GC TIME

Fig. 5a, 5b, and 5c show the total GC time under different GC threshold settings. IPPBE reduces the total GC time by 32%, 26%, and 13% on average compared with the existing scheme in the case of 5%, 10%, and 14% GC threshold, respectively. In the three cases of 5%, 10%, and 14% GC threshold settings, the performance of our scheme outperforms IPPBE by 17%, 34%, and 57%, respectively.

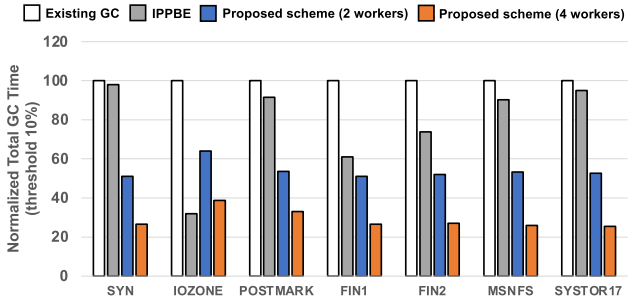
As shown in the figure, the performance of IPPBE under each workload varies greatly. This is because the performance of IPPBE relays on the ability to reduce the number of valid page migrations. Table 4 shows the number of valid page migrations in each workload when using IPPBE, which is normalized to the number of valid page migrations when using the existing scheme. As the table shows, the more the number of valid pages migration is reduced due to the use of IPPBE, the better the performance of IPPBE. Postmark is a special case. In the case of no reduction in the amount of the number valid page migrations, Postmark still has a performance improvement under 5% GC threshold. This is because IPPBE erases two blocks at a time within one block erase latency, and the average number of valid page migrations of Postmark is only 4.5 (not listed in the table). The time to migrate valid pages (1 *ms*) is less than the time required to erase one block (1.5 *ms*), thus reducing the overall time of the GC. In addition, our scheme has more steady performance improvement in all workloads compared with IPPBE. As shown in the results, we demonstrate that the page-level parallelism may increase more performance compared with block-level parallelism in our evaluation case.

2) AVERAGE LATENCY

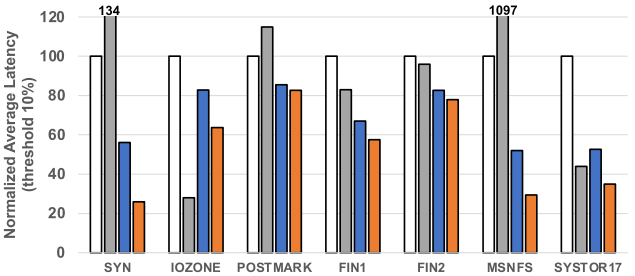
Fig 5d, Fig. 5e, and Fig. 5f show average normal I/O latencies under different GC thresholds. As shown in the figure, IPPBE's improvement in latency is less than its improvement in total GC time. In addition, sometimes the latency becomes

TABLE 4. The number of valid page migrations when using IPPBE (normalized to that of the existing scheme).

Workloads	threshold: 5%	threshold: 10%	threshold: 14%
SYN	75.4%	106.4%	99.6%
IOzone	30.8%	30.3%	104.0%
Postmark	100.1%	104.8%	99.2%
FIN 1	66.5%	54.3%	56.6%
FIN 2	78.1%	72.9%	77.0%
MSNFS	99.1%	97.3%	98.2%
SYSTOR17	99.2%	99.7%	99.7%



(a) Normalized Total GC Time.



(b) Normalized Average Latency.

FIGURE 6. Performances of existing GC, IPPBE, and our scheme on MLC chip under 10% GC thresholds.

longer. This is because IPPBE erases two victim blocks at a time, and then needs to migrate the valid pages in the two blocks to other free blocks. As a result, when the number of valid pages to be migrated is not reduced, the latency of a single GC procedure becomes much longer. Especially for a read-intensive workload with a low request interval, like MSNFS, its average latency can increase by more than 10 times compared with the existing GC scheme. In contrast, the latency of our scheme is always smaller than that of the existing scheme.

D. IMPACT OF COMMAND LATENCY AND THE NUMBER OF VALID PAGES

In order to observe how the latency of read, write, and erase operations affects our scheme, we also evaluate our parallel GC scheme on MLC chips which have high read, write, and erase latencies as shown in Table 1. For example, as shown in Table 1, the page read, page write, and block erase latencies of SLC chip are $25\mu s$, $200\mu s$, and $1.5ms$, respectively; the page read, page write, and block erase latencies of MLC chip are $75\mu s$, $1300\mu s$, and $3.8ms$, respectively. Fig. 6 shows the normalized total GC time and average latency of MLC chip

TABLE 5. The average number of valid pages and its proportion in a victim block in the case of MLC chip with 10% GC threshold.

Workloads	threshold: 10%	
SYN	126.4	49.3%
IOzone	27.0	10.5%
Postmark	18.0	7.1%
FIN 1	68.6	26.8%
FIN 2	59.1	23.1%
MSNFS	134	52.3%
SYSTOR17	141	55.0%

with a 10% GC threshold. As shown in the figure, our scheme reduces the total GC time by 46% and 70% on average when using two and four workers, respectively. The average latency is reduced by up to 44% and 74% when using two and four workers, respectively. This result shows that the overall performance gap between the existing and proposed schemes in the case of MLC increases compared with that of SLC.

The performance gap between SLC and MLC chips depends on the ratio of read/write command latency to erase command latency and the number of valid pages in victim blocks. The GC time consists of two parts. One is the migration time of valid pages, and the other is the time of erasing the victim block. When using the SLC chip setting, the migration time of one valid page is 0.225 *ms* (one read and one write command), and the latency of erase command is $1.5ms$. The ratio of migration time of one page to a erase command latency is 1:6.6. When using the MLC chip setting, the corresponding ratio is 1:2.8. As the ratio of migration time of one page to a erase command latency increases, the proportion of the time required to migrate valid to total GC also increases. Thus, our scheme can show better performance.

Similarly, because the block erase command is issued only once in the GC procedure, as more valid pages in victim blocks, our scheme can more reduce the migration time for valid pages. In addition, the average number of valid pages in victim blocks affects the scalability. Table 5 shows the average number of valid pages in victim blocks in the case of the MLC chip. When using 10% GC threshold, the average number of valid pages in victim blocks is 26.8 and 82 in the SLC and the MLC chip, respectively. In the case of the SLC chip, the performance increment when using two and four workers is 1.7x and 2.5x, respectively. In this situation, there is a huge diminishing marginal benefit as the number of workers increases. This is because when the number of valid pages is relatively small, the time required to erase the block accounts for a large proportion of the overall GC time. However, in case of MLC chip, the performance increment when using two and four workers is 1.74x and 3.4x, respectively. In this situation, the results show an almost ideal performance increase.

When the number of valid pages in victim blocks is relatively large, the time to read/write valid pages can dominate the GC time while the time for erasing block is insignificant. Under this situation, our scheme can show much better scalability like the case in our evaluation. Consequently, as we mentioned in section IV-B, how many computing resources

that we should add can depend on the specific SSD internal resources. In future work, we will try to evaluate our scheme on a hardware emulation platform to perform precise analysis in terms of the scalability and the computing resource utilization.

The performance of IPPBE under MLC and its performance under SLC show similar characteristics. That is, the performance of IPPBE relays on the ability to reduce the number of valid page migrations during GC.

VI. CONCLUSION

SSDs have a performance variability problem due to GC overheads. To handle this issue, we present a parallel garbage collection scheme base on a new multi-register flash chip architecture which provides page-level parallelism inside SSDs. Our scheme reads valid pages from a victim block and writes the valid pages to a target block in a parallel manner. We implement and evaluate our scheme using Disksim with the Microsoft SSD extension. Experiment results demonstrate that our scheme can reduce the total GC time by 70% and 57% on average compared with the existing scheme and a state-of-the-art GC scheme IPPBE, respectively. In future work, we will verify our scheme on a real machine and architecture. Also, we will evaluate whether the volume and energy impacts are acceptable for mobile devices.

REFERENCES

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *Proc. Annu. Tech. Conf.*, vol. 57, 2008, pp. 1–14.

[2] A. Aravindan. (2018). *Flash 101: The NAND Flash Electrical Interface*. [Online]. Available: https://www.embedded.com/flash-101-the-nand-flash-electrical-interface/

[3] K. Bates and B. McNutt. (2003). *Umass Trace Repository*. [Online]. Available: https://traces.cs.umass.edu/index.php/Main/Traces

[4] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti, "Introduction to flash memory," *Proc. IEEE*, vol. 91, no. 4, pp. 489–502, Apr. 2003.

[5] M. Bjørling, J. Gonzalez, and P. Bonnet, "Lightnvm: The linux open-channel SSD subsystem," in *Proc. 15th USENIX Conf. File Storage Technol.*, Santa Clara, CA, USA, Feb. 2017, pp. 359–374. [Online]. Available: https://www.usenix.org/conference/fast17/technical-sessions/presentation/bjorling

[6] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger, "Contributors the disksim simulation environment version 4.0 reference manual," Parallel Data Lab., Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep., 2008.

[7] C. Burch. (Oct. 2011). *The Hierarchy of Memory & Caches*. [Online]. Available: http://www.toves.org/books/cache/

[8] W. Bux and I. Iliadis, "Performance of greedy garbage collection in flash-based solid-state drives," *Perform. Eval.*, vol. 67, no. 11, pp. 1172–1186, Nov. 2010.

[9] W. Choi, M. Jung, M. Kandemir, and C. Das, "Parallelizing garbage collection with I/O to improve flash resource utilization," in *Proc. 27th Int. Symp. High Perform. Parallel Distrib. Comput.*, Tempe, Arizona, 2018, pp. 243–254.

[10] CMU-SAFARI. *Mqsim Perform Precondition*. Accessed: Sep. 2, 2020. [Online]. Available: https://github.com/CMU-SAFARI/MQSim/blob/1d1a2f73f84e47a41a1f62a14f75719d0179d5e7/src/ssd/FTL.cpp

[11] M. Corporation. *Microsfot Production Server Traces*. Accessed: Aug. 15, 2020. [Online]. Available: http://iotta.snia.org/traces/158.

[12] J. Cui, W. Wu, X. Zhang, J. Huang, and Y. Wang, "Exploiting latency variation for access conflict reduction of NAND flash memory," in *Proc. 32nd Symp. Mass Storage Syst. Technol. (MSST)*, 2016, pp. 1–7.

[13] C. Du, Y. Yao, J. Zhou, and X. Xu, "VBBMS: A novel buffer management strategy for NAND flash storage devices," *IEEE Trans. Consum. Electron.*, vol. 65, no. 2, pp. 134–141, May 2019.

[14] *Intel Itanium Processor-Specific Application Binary Interface (ABI)*, FS Foundation, Jaipur, Rajasthan, 2005.

[15] T. Garrett, J. Yang, and Y. Zhang, "Enabling intra-plane parallel block erase in NAND flash to alleviate the impact of garbage collection," in *Proc. Int. Symp. Low Power Electron. Des.*, Jul. 2018, pp. 1–6.

[16] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging value locality in optimizing nand flash-based ssds," in *Proc. FAST*, 2011, pp. 91–103.

[17] D. He, F. Wang, H. Jiang, D. Feng, J. N. Liu, W. Tong, and Z. Zhang, "Improving hybrid FTL by fully exploiting internal SSD parallelism with virtual blocks," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, pp. 1–19, Jan. 2015.

[18] F.-C. Hsu, "Methods and apparatus for NAND flash memory," U.S. Patent 16 849 875, Jul. 30, 2020.

[19] 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. Int. Conf. Supercomput.*, 2011, pp. 96–107.

[20] J.-H. Huang and R.-S. Liu, "DI-SSD: Desymmetrized interconnection architecture and dynamic timing calibration for solid-state drives," in *Proc. 23rd Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2018, pp. 34–39.

[21] *Atomic Builtins Using the Gnu Compiler Collection (GCC)*, Intel, Mountain View, CA, USA, 2001.

[22] M. Jung, W. Choi, S. Srikantaiah, J. Yoo, and M. T. Kandemir, "HIOS: A host interface I/O scheduler for solid state disks," in *Proc. ACM/IEEE 41st Int. Symp. Comput. Archit. (ISCA)*, Jun. 2014, pp. 289–300.

[23] M. Jung, J. Zhang, A. Abulila, M. Kwon, N. Shahidi, J. Shalf, N. S. Kim, and M. Kandemir, "SimpleSSD: Modeling solid state drives for holistic system simulation," *IEEE Comput. Archit. Lett.*, vol. 17, no. 1, pp. 37–41, Jan. 2018.

[24] V. Kasavajhala, "Solid state drive vs. hard disk drive price and performance study," *Proc. Dell Tech. White Paper*, 2011, pp. 8–9.

[25] J. Katcher, "Postmark: A new file system benchmark," Network Appliance, New Delhi, India, Tech. Rep. TR3022, 1997.

[26] C. Kim, "11.4 a 512Gb 3b/cell 64-stacked WL 3D V-NAND flash memory," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2017, pp. 202–203.

[27] H. Kim and S. Ahn, "A buffer management scheme for improving random writes in flash storage," in *Proc. 6th USENIX Conf. File Storage Technol.*, 2018, pp. 239–252.

[28] J.-K. Kim, "Multipage program scheme for flash memory," U.S. Patent 9 484 097, Nov. 1, 2016.

[29] S. Kim, J. Bae, H. Jang, W. Jin, J. Gong, S. Lee, T. J. Ham, and J. W. Lee, "Practical erase suspension for modern low-latency ssds," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 813–820.

[30] Y. Koya, G. B. Bronner, and F. A. Ware, "Multi-page parallel program flash memory," U.S. Patent 8 310 872, Nov. 13, 2012.

[31] C. Lee. *Traces From Understanding Storage Traffic Characteristics on Enterprise Virtual Desktop Infrastructure*. Accessed: Aug. 15, 2020. [Online]. Available: http://iotta.snia.org/traces/4928

[32] C. Lee, T. Kumano, T. Matsuki, H. Endo, N. Fukumoto, and M. Sugawara, "Understanding storage traffic characteristics on enterprise virtual desktop infrastructure," in *Proc. 10th ACM Int. Syst. Storage Conf.*, May 2017, pp. 1–11.

[33] J. Lee and J.-S. Kim, "An empirical study of hot/cold data separation policies in solid state drives (SSDs)," in *Proc. 6th Int. Syst. Storage Conf.*, Haifa, Israel, 2013, p. 1.

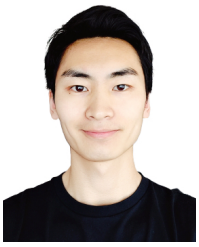
[34] J. Lee, Y. Kim, G. M. Shipman, S. Oral, F. Wang, and J. Kim, "A semi-preemptive garbage collector for solid state drives," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, Apr. 2011, pp. 12–21.

[35] M. Lin and S. Chen, "Efficient and intelligent garbage collection policy for NAND flash-based consumer electronics," *IEEE Trans. Consum. Electron.*, vol. 59, no. 3, pp. 538–543, Aug. 2013.

[36] M. Lin and Z. Yao, "Dynamic garbage collection scheme based on past update times for NAND flash-based consumer electronics," *IEEE Trans. Consum. Electron.*, vol. 61, no. 4, pp. 478–483, Nov. 2015.

[37] W. Liu, L. Zeng, and D. Feng, "CASS: A cooperative hybrid storage system consisting of an SSD and a SMR drive," in *Proc. 6th Int. Conf. Adv. Cloud Big Data (CBD)*, Aug. 2018, pp. 24–29.

[38] *Mt29f32g08cbaca Mt29f32g08cbecb Datasheet*. Accessed: Aug. 1, 2019. [Online]. Available: https://www.micron.com/media/client/global/documents/products/data-sheet/nand-flash/die/l73a_die_32gb_nand.pdf
[39] W. Norcott. (2003). *Iozone Filesystem Benchmark*. [Online]. Available: http://www.iozone.org/
[40] OCZ. *Revodrive Pci-Express SSD Specifications*. [Online]. Available: http://www.ocztechnology.com/ocz-revodrive-pci-express-ssd.html
[41] J. Ou, J. Shu, Y. Lu, L. Yi, and W. Wang, "EDM: An endurance-aware data migration scheme for load balancing in SSD storage clusters," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, May 2014, pp. 787–796.
[42] Samsung. *K9xxg08uxa Datasheet*. Accessed: Apr. 15, 2019. [Online]. Available: http://www.samsung.co.kr/
[43] Samsung. *Pm830 Datasheet*. Accessed: Dec. 5, 2020. [Online]. Available: https://www.digchip.com/datasheets/parts/datasheet/3566/PM830-pdf.php
[44] N. Shahidi, "Flash translation layer design in solid state drives," M.S. thesis, Pennsylvania State Univ. Graduate School, State College, PA, USA, 2017.
[45] T. Song, J. Jung, W. Rim, H. Kim, Y. Kim, C. Park, J. Do, S. Park, S. Cho, H. Jung, B. Kwon, H.-S. Choi, J. Choi, and J. S. Yoon, "A 7nm FinFET SRAM using EUV lithography with dual write-driver-assist circuitry for low-voltage applications," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2018, pp. 198–200.
[46] R. Stoica and A. Ailamaki, "Improving flash write performance by using update frequency," *Proc. VLDB Endowment*, vol. 6, no. 9, pp. 733–744, Jul. 2013.
[47] K.-D. Suh, B.-H. Suh, Y.-H. Lim, J.-K. Kim, Y.-J. Choi, Y.-N. Koh, S.-S. Lee, S.-C. Kwon, B.-S. Choi, J.-S. Yum, J.-H. Choi, J.-R. Kim, and H.-K. Lim, "A 3.3 v 32 mb NAND flash memory with incremental step pulse programming scheme," *IEEE J. Solid-State Circuits*, vol. 30, no. 11, pp. 1149–1156, Dec. 1995.
[48] K. Takeuchi, "Novel co-design of NAND flash memory and NAND flash controller circuits for sub-30 nm low-power high-speed solid-state drives (SSD)," *IEEE J. Solid-State Circuits*, vol. 44, no. 4, pp. 1227–1234, Apr. 2009.
[49] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, S. Ghose, and O. Mutlu, "Mqsim: A framework for enabling realistic studies of modern multi-queue ssd devices," in *16th USENIX Conf. File Storage Technol.*, 2018, pp. 49–66.
[50] "Flash interface specification," Open NAND Flash Interfaces (ONFi), Tech. Rep. Revision 4.1.
[51] S. Wu, Y. Lin, B. Mao, and H. Jiang, "GCaR: Garbage collection aware cache management with improved performance for flash-based SSDs," in *Proc. Int. Conf. Supercomput.*, 2016, pp. 1–12.
[52] G. Xu, Y. Liu, X. Zhang, and M. Lin, "Garbage collection policy to improve durability for flash memory," *IEEE Trans. Consum. Electron.*, vol. 58, no. 4, pp. 1232–1236, Nov. 2012.
[53] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, A. A. Chien, and H. S. Gunawi, "Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in nand ssds," *ACM Trans. Storage*, vol. 13, no. 3, p. 22, 2017.
[54] J. Yang and S. Pei, "Thermo-GC: Reducing write amplification by tagging migrated pages during garbage collection," in *Proc. IEEE Int. Conf. Netw., Archit. Storage (NAS)*, Aug. 2019, pp. 1–8.
[55] 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. 11th USENIX Workshop Hot Topics Storage File Syst.*, 2020, pp. 1–8.
[56] B. Zhou, S. Wan, and C. Xie, "Isolation: Inexpensively separating cold data via garbage collection to improve the lifetime and performance of NAND flash SSDs," *Concurrency Comput., Pract. Exper.*, Jul. 2019, p. 5460.



GUANGYU ZHU received the B.S. degree from the School of Computer Science and Engineering, Chung-Ang University, where he is currently pursuing the M.S. degree. His research interests include operating systems, and file and storage systems.

JAEHYUN HAN received the B.S. degree from the School of Computer Science and Engineering, Chung-Ang University, where he is currently pursuing the M.S. degree. He was an Intern with Bosornd, in 2018 and 2019. His research interests include distributed systems and operating systems.

YONGSEOK SON received the B.S. degree in information and computer engineering from Ajou University, in 2010, and the M.S. and Ph.D. degrees from the Department of Intelligent Convergence Systems and Electronic Engineering and Computer Science, Seoul National University, in 2012 and 2018, respectively. He was a Postdoctoral Research Associate in electrical and computer engineering with the University of Illinois at Urbana–Champaign. He is currently an Assistant Professor with the School of Computer Science and Engineering, Chung-Ang University. His research interest includes operating, distributed, and database systems.