Parallelizing Garbage Collection with I/O to Improve Flash Resource Utilization

Wonil Choi
Pennsylvania State University
wuc138@cse.psu.edu

Myoungsoo Jung
Yonsei University
m.jung@yonsei.ac.kr

Mahmut Kandemir
Pennsylvania State University
kandemir@cse.psu.edu

Chita Das
Pennsylvania State University
das@cse.psu.edu

ABSTRACT

Garbage Collection (GC) has been a critical optimization target for improving the performance of flash-based Solid State Drives (SSDs); the long-lasting GC process occupies the flash resources, thereby blocking normal I/O requests and increasing response times. This is a well-documented problem, and a wide range of prior works successfully hide the negative impact of GC on the I/O response times. In this paper, however, we unveil another serious side-effect of GC, called the plane under-utilization problem. More specifically, while a plane is busy doing GC, the other plane(s) in the same die remain idle, as all the planes in a die share a single command and address path that is dedicated to the GC. We also note that most of the state-of-the-art proposals attacking the GC impact on I/O response times are not able to resolve the plane under-utilization problem, and in turn, miss a great potential to further improve the SSD performance. Thus, we next propose a scheduling technique, I/O-parallelized GC, which leverages the idle planes during GC to serve the blocked I/O requests. As a result, flash resources (planes) can be active during the most of GC time and the blocked I/O requests can get serviced quickly, and in turn, an improved SSD performance can be achieved. Using simulation-based evaluations over a wide variety of workloads, we show that the proposed I/O-parallelized GC scheme can improve the response times of the GC-affected I/O requests by 83% (reads) and 70% (writes), by increasing the average plane utilization from the (two planes-per-die) baseline 50% to 74.4% during GC. The I/O-parallelized GC is orthogonal to prior proposals that hide GC overheads; so, they can be combined for further SSD performance improvement.

CCS CONCEPTS

• **Information systems** → **Flash memory**; **Storage management**; • **Theory of computation** → **Parallel algorithms**;

ACM Reference Format:
Wonil Choi, Myoungsoo Jung, Mahmut Kandemir, and Chita Das. 2018. Parallelizing Garbage Collection with I/O to Improve Flash Resource Utilization. In *HPDC '18: The 27th International Symposium on High-Performance Parallel and Distributed Computing, June 11–15, 2018, Tempe, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3208040.3208048

HPDC '18, June 11–15, 2018, Tempe, AZ, USA

ACM ISBN 978-1-4503-5785-2/18/06…$15.00
https://doi.org/10.1145/3208040.3208048

1 INTRODUCTION

The performance of storage is crucial for many parallel and distributed computing systems. Solid State Drives (SSDs) have emerged as a popular choice for various computing environments where ultra-fast storage performance is required. Despite their overall low response times against spinning disks, SSDs employed in enterprise and high-performance computing systems do not usually deliver uniform performance, ultimately degrading user experience and impacting revenues negatively. In other words, the response times of a considerable fraction of I/O requests reach up to tens of milliseconds [9], and this is referred to as the "tail latency" problem [15, 27]. Among various sources of this performance degradation in an SSD, garbage collection (GC) is the primary contributor. Specifically, as GC operations have the highest priority and take long times, they block the service of normal I/O requests until they complete; thus, they can significantly increase the I/O response times.

This well-known problem has been investigated in the past decade, and a wide range of prior proposals attacked the negative impact of GC on I/O response times. For example, a representative solution [21] makes the long-lasting GC process preemptible; when I/O requests are submitted during GC, the ongoing GC is paused, then the I/O requests get serviced, after which the GC is resumed. Doing so prevents I/O requests from waiting for GC execution to finish (and in turn, their response times from increasing). Recently, a state-of-the-art proposal [32] shows that it can almost-perfectly eliminate the tail latencies by keeping multiple same data in different locations and by reading a replica from a GC-free location.

In this paper, however, we unveil another negative impact of GC: flash resources are severely under-utilized during GC. In fact, most today's commercially available flash chips provide high levels of intra-chip parallelism – there are multiple dies in a chip with multiple planes in each die. Dies are independent units, and die-level parallelism is less affected by GC (i.e., while one die is involved in GC, other dies can service I/O requests destined to them). However, the case is opposite for plane-level parallelism. Due to the architectural constraints[1] of flash chip, if one of the planes in a die is busy with GC, other planes in the same die remain *idle*. We call this *plane under-utilization during GC*. To the best of our knowledge, this plane under-utilization problem is not revealed and resolved in any of the prior works. According to our investigation, prior proposals

[1] Planes in a die share a single command register and address register.

(including the state-of-the-art [32]) attempting to avoid/hide the negative impact of GC on I/O response times still have the plane under-utilization problem (a detailed discussion can be found in Section 3.3). So, most of the prior GC-centric works miss a potential to further improve the SSD performance by taking advantage of the under-utilized resources (i.e., idle planes) during GC.

Motivated by such under-utilized planes during GC, we propose a novel I/O scheduling scheme, called **I/O-parallelized GC**, to improve the low flash resource utilization caused by GC. Our strategy is to parallelize I/O requests with GC operations by using the under-utilized planes. Unfortunately, not all I/O requests can be parallelized with GC operations. That is, due to circuit-level limitations, multiple planes in a die can be simultaneously activated, if and only if the target requests have the same command type (read or write) and their address offsets are same. In fact, the current flash devices provide an advanced operation form (called multi-plane command) to make parallel accesses to two or more planes at the same time, which can be constructed only when certain command and address constraints (explained shortly) are satisfied. Accordingly, our proposed I/O-parallelized GC explores the potential of parallelizing (or building multi-plane commands based on) *normal I/O requests* (i.e., ones from the host) with *GC reads or writes* (i.e., page migrations). At a given time during a GC execution, if there is a read (or write) request waiting at the flash chip's queue and is mapped to a die that is currently executing GC, it can be combined with one of GC reads (or writes), if their operation types and address offsets are identical.

Clearly, the efficiency of our proposed scheme depends on the appearance frequency of the normal I/O requests whose address offsets are *identical* to those of GC reads and writes – that would be quite random and application dependent. With the goal of enhancing the efficiency of the proposed scheme, we introduce two design modifications in the FTL algorithm and implementation. First, we propose to have two active blocks per plane (instead of one active block in the baseline). While the first active block works the same as the baseline, the second active block helps us keep every GC write aligned with an I/O write request in the queue. Second, for a GC victim block, we propose to select a block whose valid page pattern can generate the most page offset matches with read I/O requests in the queue (instead of the block the baseline FTL algorithm selects). By doing so, our proposed scheme is able to find more opportunities to parallelize I/O requests and GC operations.

We make the following main **contributions** in this paper:

- We reveal the plane under-utilization problem – while a plane is busy doing GC, the other planes in the same die remain idle, which has received little attention until now. We also notice that none of the prior GC-related proposals addresses this problem.
- Our proposed I/O-parallelized GC scheme can improve plane utilization during GC by exploiting plane-level parallelism. Our strategy is to build a series of multi-plane read (or write) commands, each of which is formed based on a GC read (write) and an I/O read (write).
- We propose to modify the underlying GC victim block selection algorithm and the active block management strategy in conventional FTLs, in order to substantially boost the delivered plane-level parallelism by our scheme.
- With all these enhancements, our I/O-parallelized GC scheme gives an overall plane-level utilization of 74%, compared to 50% of the default GC, in a configuration with 2 planes per die. Examining a 1TB SSD with 16 MLC flash chips, this increase in plane utilization leads to an improvement of 83% and 70% (over the baseline SSD) in response times of read and write I/O requests, respectively, that are affected by GC. This in turn results in an average improvement of 14.7% in total SSD service time.

2 PRELIMINARIES

2.1 SSD Internals

Figure 1a shows the internal architecture of a modern SSD composed of four main components:

• **Host interface (HI).** HI is the unit responsible for scheduling and performing I/O requests and data transfers between the host computer and the SSD.

• **Flash memory chips.** SSDs employ a number of flash memory chips as their storage medium. Detailed description of the internal organization of a flash chip is given below.

• **Flash translation layer (FTL).** This is a software layer which is responsible for mapping I/O requests onto flash chips and handling the basic flash functionalities like Garbage Collection (GC). To run FTL, SSDs employ a processor equipped with an SDRAM cache. We explain GC, which is the focus of our optimization, below. Details of the other FTL functionalities can be found in [8, 29, 30].

• **Memory channels (buses).** The flash chips are connected to the processor and cache through multiple buses. By employing multiple buses and flash chips, SSDs provide coarse-grain (inter-chip) parallelism.

2.2 Flash Chip Organization

As shown in Figure 1b, each flash chip is composed of multiple dies with two or four planes in a die. There are thousands of blocks in a plane and hundreds of pages in a block. A block is the unit of erase whereas a page is the unit of read and write operations and has a typical size of 4KB to 16KB in today's flash chips. As (1) each page write has to be preceded with an erase operation and (2) units of write and erase operations are different, flash memory employs an out-of-place update policy – that is, the update data is written into a clean page and the old page data is invalidated. Note also that there is an asymmetry in latencies of flash operations, i.e., a write is slower than a read, and an erase is slower than a write.

For each plane, the FTL keeps one active block into which new data is written. In order to avoid cell-to-cell interference between pages, the writes in an active block *must* be done in a sequential order (first write to Page 1, second write to Page 2, and so on) [6]. Once clean pages run out in the active block, FTL selects a clean (erased) block as the new active block. The current clean page (where a new write get serviced) in the active block is called the active page.

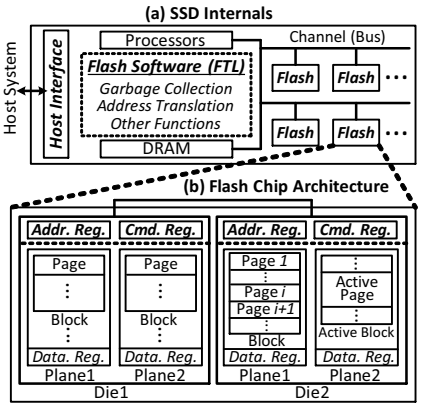


Figure 1: (a) SSD internals and (b) flash organization.

2.2.1 **GC Process.** When the number of clean blocks in an SSD becomes lower than a threshold, called the overprovisioning threshold, FTL invokes a GC process through which a number of blocks are erased (and added to clean block pool). A GC operation has three phases: (i) a Victim-Select algorithm identifies one or more (used) blocks as victim blocks; (ii) the valid pages in victim block(s) are moved to clean pages; and (iii) the victim block(s) are erased. GC generally takes a long time since it moves (reads and writes) all valid pages of the victim blocks. To keep this cost low, the Victim-Select algorithm may prefer to choose blocks with the least number of valid pages (i.e., GREEDY algorithm [5, 10, 13]). We use GREEDY as our baseline Victim-Select algorithm in this paper.

We remark on some important properties of GC. First, since an SSD always needs to guarantee a predetermined amount of free space for correct operation, GC has the highest priority, and, when invoked, it blocks the service of normal I/O operations. Second, GC can be a "global" or a "local" process. In a global GC, valid pages in victim blocks are allowed to move to blocks in different chips/dies/planes. In a local GC, on the other hand, the source and destination blocks are in the same plane. Our proposed mechanism is general and applicable to both global and local GCs. However, in order to simplify our description, we first assume a local GC, and later discuss how our proposal can work with a global GC.

2.2.2 **Intra-chip Parallelism.** A modern flash chip has a highly-parallelized internal architecture. There are two types of parallelism provided inside a flash chip:

• **Die-level parallelism.** Each die has its own address, command and data registers, which enables dies operate independently. Indeed, dies of a chip only share the chip's internal bus to communicate with outside. There is a *die interleaving* command in modern flash memories which enables multiple dies at the same time. Note that, as each die has its own command and address registers, die interleaving commands do not have any (command) type or address restrictions.

• **Plane-level parallelism.** Within a die, all planes share the same address and command registers, but each plane has its own data register (and cache register [23]). This organization imposes two restrictions on parallelizing operations over multiple planes of the same die, which are realized by a *multi-plane* advanced command [11]. The restrictions are of two types: (i) **command type restriction**. Operations on multiple planes have to be of the same type, i.e., read or write; and (ii) **address restriction**. The multi-plane commands must have the same address offsets, i.e., the page numbers in the target blocks have to be identical, while the block numbers do not matter. Figure 2 shows the timing diagrams of a two-plane read and a two-plane write. A two-plane read is composed of three steps: (i) the controller sends the multi-plane command, followed by addresses for Plane ❶ and Plane ❷; the block numbers can be different, but the page offsets must be identical. (ii) the read command fetches data from both the planes in parallel and stores them in their data registers; and finally, (iii) the data in both the data registers are transferred out of chip one by one. Similarly, the two-plane write begins by sending the command and addresses, followed by serially transferring the data to each plane (to be stored in the data registers). Lastly, the data loaded into the data registers are written into the active pages of the planes in parallel.

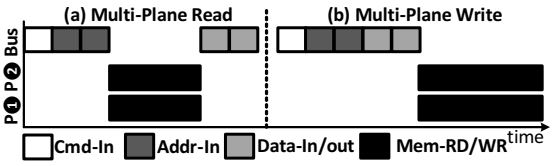


Figure 2: Simplified timing diagram of two-plane read (a) and two-plane write (b) operations. The exact format can vary depending on device specifications.

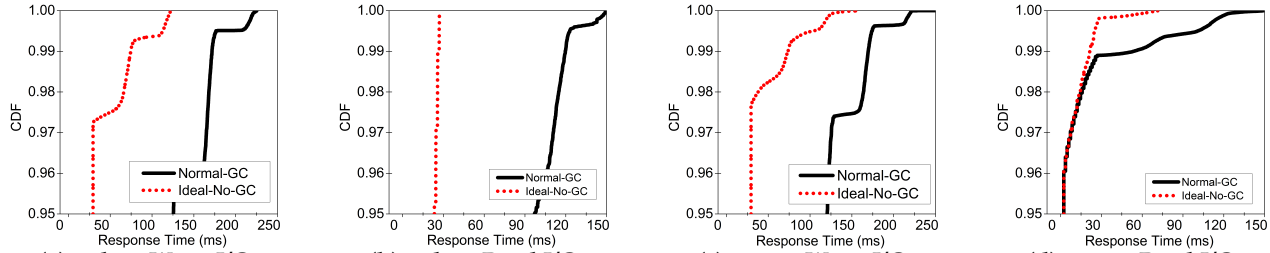
Some recent works [1, 7, 14, 31] exploited the potential of advanced commands (multi-plane and die-interleaving) and proposed schedulers that aim to boost intra-chip parallelism by serving multiple I/O requests in a chip at the same time. In this work, on the other hand, we exploit the intra-chip parallelism opportunities (particularly, plane-level parallelism) and multi-plane commands, in order to serve normal read and write I/O operations (sent by the host computer) in parallel with reads and writes of GC – doing so can significantly improve the flash resource utilization during GC and reduce the response times of I/O requests that get stalled by GC.

3 OUR MOTIVATION: FLASH PLANE UNDER-UTILIZATION DURING GC

In this section, we first revisit the well-known problem of increasing I/O response times due to the long-lasting GC process. We then explain a resource waste problem that occurs during the GC, called plane under-utilization problem. Finally, we discuss that state-of-the-art techniques attacking the GC-induced long I/O response times still suffer from the plane under-utilization problem.

3.1 Impact of GC on I/O Response Time

The time-consuming GC invocation prevents I/O requests from being processed until it finishes, and this significantly increases read and write I/O response times. Prior works have extensively studied the performance impact of GC on SSDs [18, 28], and demonstrated that GC causes sudden increases in write traffic to flash substrate (named as *write cliff*). The write



(a) mds_0 Write I/O (b) mds_0 Read I/O (c) proj_0 Write I/O (d) proj_0 Read I/O

Figure 3: The 95th percentile CDFs of write and read I/O response times for two of our workloads. Each graph compares two systems: (1) *Normal-GC*, the baseline SSD that executes GC when required, and (2) *Ideal-No-GC*, an SSD that is assumed to execute GC with zero latency overhead. This analysis confirms that GC is the main source of generating the tail latencies.

cliff is the primary reason for increasing I/O request response time by tens to hundreds times. This performance problem caused by GC is also referred to as the "tail latency" problem, since response times of a considerable fraction of I/O requests reach up to tens of milliseconds [9].

To quantify the impact of GC on I/O response times and to observe the tail latency problem, we conducted a simulation-based experiment and measured I/O response times for two different scenarios. In the first scenario, GC executes normally and its latency overheads (read, write, and erase latencies) are accurately modeled. In the second scenario, the functionality of GC (i.e., generating free space) is modeled; however, we assume that GC has no latency overhead (i.e., its execution latency is set to zero). Figure 3 shows the 95th percentile cumulative distribution function (CDF) of response time of (read and write) I/O requests in two representative workloads (`mds_0` and `proj_0`) for the two scenarios; Scenarios 1 and 2 are labeled as *"Normal-GC"* and *"Ideal-No-GC"*, respectively. Note that we used the SSD simulation settings given in Section 5.

We make the following observations from these plots:

• For the two workloads tested, we can observe that SSD suffers from the tail latency problem; see I/O response times of the baseline system, *"Normal-GC"*, for each workload – that would be in range of hundreds of milliseconds for write response times, and tens of milliseconds for read response times.

• GC is the main contributor to significantly increase the write I/O response times. We can see that there are outstanding gaps between the baseline system (*"Normal-GC"*) and the ideal system (*"Ideal-No-GC"*). By assuming zero GC overhead, more than 95% of the write I/Os get serviced in less than 25ms.

• The significant increase in read I/O response times also originates from the GC – there is a remarkable gap between Ideal-No-GC and Normal-GC. Particularly, for `mds_0`, a considerable amount of read I/Os are disturbed by the GC.

3.2 Impact of GC on Plane Utilization

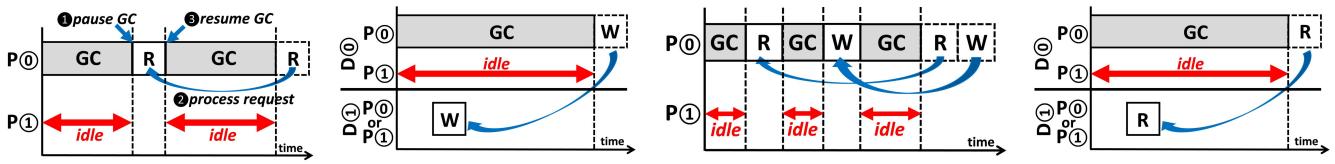
In addition to this well-known GC problem (i.e., the significant increase in I/O response times) demonstrated above, we identify another critical impact of GC; that is, the flash resource is severely *under-utilized* during GC. More specifically, during GC, the FTL scheduler reads the valid pages of the GC-victim block (source block) and writes them in the active block. This series of data movements always makes one plane busy with GC sub-operations (reads and writes) and leaves all other planes in the same die idle. In fact, as all planes in a die share the same (single) address and command registers, GC sub-operations prevent I/O requests in the queue from being served in any other plane. We refer to this characteristic as the *plane underutilization problem* during GC. Assuming that each die has two (or four) planes, the plane-level utilization is 50% (or 25%) during GC in conventional FTLs.

Figure 5a illustrates this problem through an example scenario. Without loss of generality, we make three assumptions in this example: (i) the flash chip has a single die with two planes; (ii) the read and write operations take 1 and 2 cycles, respectively; and (iii) the GC source (victim) and destination blocks belong to the same plane (Blocks 2 and 7 of Plane 0 are the GC source and destination, respectively). Figure 5a also shows the request queue status at "Time 0": there are 2 requests mapped to Plane 0 (one read "K" and one write "N") and 6 requests mapped to Plane 1 (four reads "I", "M", "O", and "P", and two writes "J" and "L"). Here, **A** to **H** are GC operations, **I** to **P** are normal I/O requests, **(R)** and **(W)** indicate reads and writes requests, **P0** and **P1** are target plane numbers, **B❶** to **B❾** are target block numbers, and **P①** to **P⑱** are target page numbers. At "Time 0", Plane 0 enters the GC mode and the valid pages in its Block 2 (i.e., Pages 3, 7, 9, 15, ...) are moved to Block 7, one by one. We can see that, with the baseline I/O scheduler (the current practice), Plane 1 is completely "idle" (underutilized) during GC on Plane 0. When GC finishes ("Time 50"), the controller resumes servicing normal I/O requests in a FCFS fashion. This example demonstrates that, during GC ("Time 0" to "Time 50"), plane-level utilization is limited to 50%. This example also shows that GC increases the response times of the requests in two different ways:

• Response times of requests mapped to Plane 0 (i.e., requests "K" and "N") increase due to the plane conflict with GC. These two requests cannot be served before "Time 50", because the GC is given the highest priority by the scheduler.

• Response times of requests mapped to Plane 1 (i.e., requests "I", "J", "L", "M", "O", and "P") increase because the single control path (the address and command registers which are shared among all planes) is dedicated to the GC operations in Plane 0. Even though Plane 1 is idle during the entire GC execution, these requests cannot get serviced before "Time 50".

We are interested in the I/O requests in the second category; they are stalled without getting any service, even though their



(a) PGC of ISPASS'11 [21] (b) SSDsim of ICS'11 [14] (c) HIOS of ISCA'14 [16] (d) TTFlash of FAST'17 [32]

Figure 4: Representative state-of-the-art proposals that successfully hide the GC impact on I/O response times fail to resolve the plane under-utilization problem during GC – while a plane (P0) is busy with GC, the other plane (P1) still remains idle. All these proposals miss a potential to further improve the SSD performance by leverging the idle plane during GC.

target plane remains idle during the GC. Since GC generally takes a long time, letting the plane being idle during GC translates into a significant resource waste. This problem gets worse in cases where a die consists of four planes; while one plane is involved in GC, all the other three planes stay idle. If the under-utilized plane(s) could serve the waiting I/O requests regardless of the GC execution in the same die, it would be a great opportunity to improve the I/O response times and the flash resource utilization. To the best of our knowledge, the plane under-utilization problem has not been investigated yet, and there has been no study to take advantage of the under-utilized planes during GC for improving the SSD performance.

3.3 Limitations of State-of-the-art Proposals

There have been a wide range of proposals to hide GC overheads, with the goal of minimizing its impact on the I/O response times. Even though most of the prior works are quite effective in suppressing the increase in I/O response times due to the GC, none of them pays attention to the plane under-utilization problem or makes use of the idle resources (i.e., planes) to further improve the SSD performance. To see how they prevent the I/O response time from increasing but fail to address the plane under-utilization problem, we revisit four representative state-of-the-art proposals:

• **Preemptive GC** (*PGC in ISPASS'11* [21]): Figure 4a illustrates that PGC avoids the impact of GC on I/O response times by allowing an I/O request to preempt ongoing GC process. That is, given a read request, ❶ the GC in progress is suspended first, ❷ the read request get serviced next, and then ❸ the pending GC is resumed. Even though doing so successfully prevents the read request from increasing due to the GC, one can see that the other plane (**P**①) still suffers from the under-utilization problem, while a plane (**P**⓪) is doing GC.

• **Dynamic allocation** (*SSDsim in ICS'11* [14]): Figure 4b shows that SSDsim hides GC overhead by redirecting write I/O requests to one of other GC-free dies, based on a dynamic address allocation. Specifically, given a write request destined to a plane (**P**⓪) of a die (**D**⓪) where GC is ongoing, it can get serviced by another GC-free die (**D**①) without waiting for the GC execution to finish. Although doing so helps write I/O requests evade the increase in their response times, one can notice that the other plane of the die performing GC (**P**① of **D**⓪) still remains idle.

• **GC distribution** (*HIOS in ISCA'14* [16]): Figure 4c depicts that HIOS minimizes the GC-induced delay by segmenting GC overheads and distributing them over multiple I/O requests. To be specific, observing that a typical I/O request has a slack to its deadline, the entire GC operations (reads and writes) are divided into multiple chunks and each of them is processed using the slack of each I/O request. In this way, all I/O requests can be completed within their deadlines in the plane doing GC (**P**⓪); however, one can find that the other plane (**P**①) is still under-utilized during each chunk of the GC.

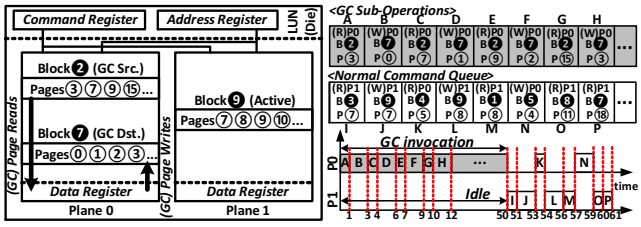
• **Exploiting redundancy** (*TTFlash in FAST'17* [32]): Figure 4d describes that TTFlash successfully resolves the tail latency problem caused by GC by maintaining multiple copies of data in different places and reading the replica from a GC-free die. Since the same data are maintained in multiple dies (both **D**⓪ and **D**①), given a read request, it can get serviced from a GC-free die (**D**①), instead of the original target die that is busy doing GC (**D**⓪). However, one can see that the other plane (**P**①) still remains idle while a plane is performing GC (**P**⓪). It is also to be noted that TTFlash assumes a single-plane die configuration where the plane under-utilization problem cannot be observed.

Above state-of-the-art proposals are quite effective in hiding the negative impact of GC on I/O response times; however, none of them can address the plane under-utilization problem. This implies that they miss the potential of further improving the SSD performance by wasting a lot of flash resources during GC. We want to highlight that our proposal (explained in the next section) is orthogonal to such state-of-the-art techniques; consequently, it can be combined with them to further improve the SSD performance by taking advantage of the under-utilized plane during GC.

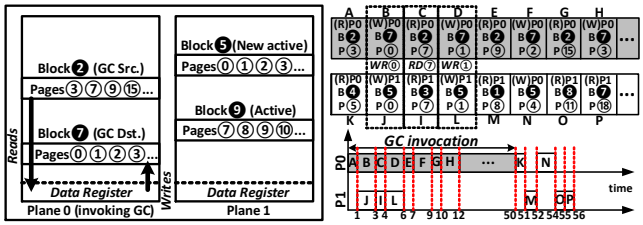
4 OUR PROPOSAL: I/O-PARALLELIZED GC

4.1 Basic Mechanism

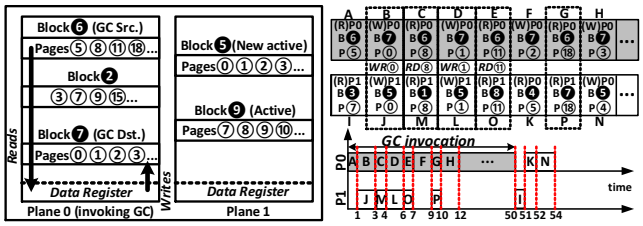
Parallelizing I/O with GC – The principle behind our proposed scheme is to build "multi-plane commands" by associating normal I/O requests (waiting in the queue) with GC sub-operations and thus utilize the idle plane(s) during GC. Figure 5b depicts how our new scheduler serves some normal I/O requests during GC. For example, at "Time 3", we can make a two-plane read command by associating the read of "Page 7" from "Block 2" on Plane 0 ("C" as a part of GC) and the read of "Page 7" from "Block 3" of Plane 1 (request "I"). Reading these two pages completely satisfies the constraints for the multi-plane read command, i.e., the same operation type and same page offset (note that the block numbers can be different). In this fashion, the scheduler can increase the plane-level parallelism in a die that is busy with GC. Furthermore,



(a) Baseline I/O scheduler (Section 3.2); the GC-free plane (Plane 1) in the same die is under-utilized during GC on Plane 0.



(b) I/O-parallelized GC scheduler with an enhancement for more multi-plane writes (Section 4.2); employing a new active block, Block 5, (instead of the current active block, Block 9) in Plane 1 can synchronize its write point with that of Block 7 in Plane 0.



(c) I/O-parallelized GC scheduler with an enhancement for more multi-plane reads (Section 4.3); our victim selection algorithm picks another block, Block 6, (instead of the original victim block, Block 2) in Plane 0, which increases the utilization of Plane 1.

Figure 5: An example scenario illustrating the plane under-utilization problem (a), and details of our proposed mechanisms (b) and (c).

this GC-I/O parallelization does not hurt GC execution time, since (i) multi-plane read latency is similar to single plane read latency, and (ii) transferring GC data in/out of the chip is prioritized over the normal I/O requests.

Design details – When GC is invoked in a conventional FTL, the baseline I/O scheduler stalls all I/O requests in the queue and serves the GC sub-operations. As described in Algorithm 1, once a GC source (victim) block is selected, each valid page in it is read and written to the active page of the GC destination (target) block. Note that the series of these GC read and write requests are all normal single read or write requests. On the other hand, our I/O-parallelized GC simultaneously schedules I/O requests in the queue and GC sub-operations, if doing so is possible. Algorithm 2 describes how our proposed scheduler combines a GC read (write) with an I/O read (write) request. When each valid page is read from the GC source block, our proposed scheduler scans the I/O requests (specifically, read requests only) waiting in the queue, and it finds an I/O read request whose page offset matches with the page offset of the valid page. Once such an I/O read is found in the queue, it can be combined with the GC read, which

Algorithm 1: Conventional GC algorithm; each valid page in the source block is read and written to the destination block.

```
Input: GC source block (Blk)
Output: A series GC copy (read and write) operations
for V ←each valid page in Blk do
    issue a single read (V); // normal GC read
    issue a single write (V); // normal GC write
end
```

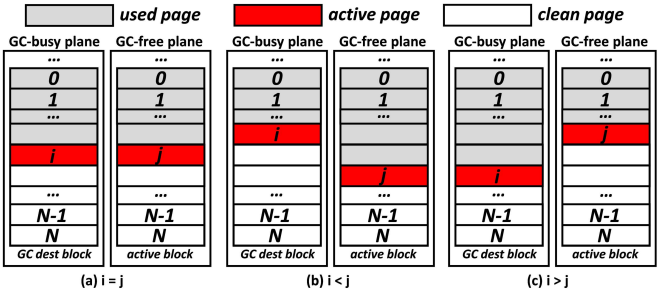


Figure 6: In a two-plane die configuration, the active page offset (i) of the GC-busy plane and that (j) of the GC-free plane have three possible scenarios; multi-plane writes are generated (a) from the beginning or (b) after a few issuing single GC writes, while (c) it is not likely to parallelize GC writes and I/O writes.

together generates a "multi-plane read". Otherwise, the valid page is read using a single read, as the conventional GC does. When the valid page is written into the destination block, the proposed scheduler checks the page offsets of the active pages in all planes, regardless of the page offsets of the GC valid pages or waiting I/O requests. More specifically, if the active page of the destination block (i.e., the active block of GC-busy plane) has the same offset with the active page of the active block in the paired (busy-free) plane, the GC write can be combined with an I/O write (specifically, the first write request) waiting in the queue, which together generates a "multi-plane write". Otherwise, the GC write uses a single write, as the conventional GC does. This process is repeated until all valid pages in the GC source block are moved.

Since a page write is served with the active page of the active block in the target plane, whether one can generate a multi-plane write or not is determined by the page offsets of the active pages in the paired planes. Figure 6 illustrates three possible scenarios of active pages in a two-plane die that is busy doing GC. The first scenario (Figure 6a) is that the active pages have the same offset in both the planes (i.e., $i = j$). Note that the active block numbers do not matter in generating multi-plane commands. In this scenario, each GC write can be combined with an I/O write in the queue, thereby forming a multi-plane write, until all valid pages are moved (i.e., GC finishes) or all I/O writes in the queue get serviced, whichever comes first. In the second scenario (Figure 6b), the active page offset of the GC-free plane is larger than that of the GC-busy plane (i.e., $i < j$). Hence, the GC writes cannot be combined with the I/O writes from the beginning; instead, the GC writes are served in the form of a single normal write one by one. However, after consuming a few active pages in the GC-busy plane, the

Algorithm 2: I/O-parallelized GC algorithm; for each page copy, it tries to find read and write I/Os from the queue, which can be parallelized with the GC reads and writes.

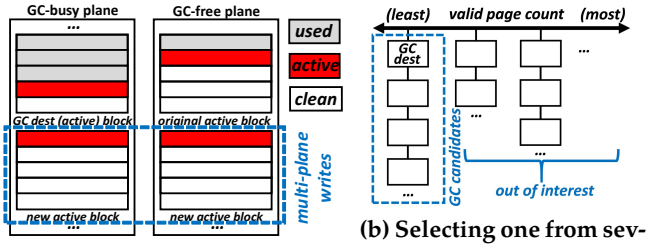
```
Input: GC source block (Blk), I/O queue (Q), active page of GC-busy plane
       (i), active page of GC-free plane (j)
// when active page is written, it is incremented by one
automatically; i = i + 1 and j = j + 1
Output: A series GC copy (read and write) operations
for V ←each valid page in Blk do
    v ← page offset of V;
    for R ← each read request in Q do
        r ← page offset of R;
        if v == r then // I/O-parallelized GC read
            issue a multi-plane read (V, R);
            moved ←1;
            break;
    end
    if moved ≠ 1 then // normal GC read
        issue a single read (V);

    if i == j then // I/O-parallelized GC write
        W ← fetch the first write request in Q;
        issue a multi-plane write (V, W) ;
    else // normal GC write
        issue a single write (V) ;
end
```

active page offsets can be the same in both the planes; now, the remaining GC writes and the I/O writes can form multi-plane writes, as in the first scenario. The last scenario (Figure 6c) shows that the active page offset of the GC-busy plane is larger than that of the GC-free plane (i.e., $i > j$). Unfortunately, in this case, multi-plane writes cannot be generated, as the active page offset of the GC-busy plane continues to increase, while the active page offset of the GC-free plane remains unchanged.

Timing overhead – Compared to the baseline scheduler (Algorithm 1), our proposed scheduler (Algorithm 2) can bring a timing overhead in examining a page read and/or write from each valid page movement can be parallelized with an I/O read and/or write. When a valid page is read, we need to scan the I/O queue to find a read request that can be parallelized with the page read. This process of scanning I/O queue is not a big burden, since (according to our analysis) the maximum number of I/O reads waiting at the queue is typically between 10 and 20. Assuming that the scheduler (as part of FTL) is executed using a high-speed ARM's Cortex processor in an SSD, the I/O queue scanning process takes at most hundreds of nanoseconds, which is much smaller than a flash read latency (115 microseconds in our experimental setup). When the valid page is written, scanning the I/O queue is not required; since FTL is always aware of the active block/page number (or offset) of each plane, deciding whether to construct a multi-plane write results in a negligible overhead.

Limitations – Although this scheme seems to be efficient at first glance, we do not expect it to help substantially boost plane parallelism. The reason is that it is very challenging to find a sufficient number of normal I/O requests (in the queue) for parallelizing with all GC operations (or at least a large number of them). In fact, the challenge is three-fold:

• The page offsets of the GC operations vary significantly, as the valid page patterns in victim blocks can be different from one another.



(a) Assigning a new active block. **(b) Selecting one from several GC candidate blocks.**

Figure 7: Enhancements to our I/O-parallelized GC to improve plane-level parallelism; (a) for more multi-plane writes, and (b) for more multi-plane reads.

• The page offsets of the read I/O requests headed to the planes not involved in GC (Plane 1 in our example of Figure 5a) vary significantly, since read I/O patterns are application-dependent. This limits chances for building multi-plane reads.

• There is no guarantee that the write point (or page offset) in the active block of the plane busy with GC is synchronized with that in the active block of the planes free from GC. This limits chances for constructing multi-plane writes.

We introduce two simple enhancements to the FTL software to substantially boost the chances for generating *both* the write and read multi-plane commands, which are discussed in Sections 4.2 and 4.3, respectively.

4.2 Towards More Multi-Plane Writes

Strategy – The opportunity of building multi-plane writes from our scheduling scheme is determined by the page offsets of active pages in the planes, instead of the page offsets of GC writes and I/O writes. Motivated by this, we propose to (i) employ one more (new) active block for each plane and (ii) use the new active block for serving GC or I/O writes during GC. Doing so can make the write point in the active block of the GC-busy plane synchronized with that in the active block of the GC-free planes, since the active page offsets in both the new active blocks are all zero (we assume the first page number in a block is zero). Figure 7a illustrates that each of the planes (the plane busy doing GC and its pair) has a new active block and the active page offsets of the two planes are the same (i.e., zero). With this additional active block, GC writes can be combined with I/O writes from the beginning, until all valid pages are moved or all I/O writes in the queue get serviced, whichever comes first. After the GC finishes, each plane can have two partially-used active blocks; they can be used to serve future I/O writes until their pages are fully used, without assigning a new active block.

Example – Using the example scenario we showed before, Figure 5b describes how employing a new active block in each plane can increase the chances for constructing multi-plane writes, and in turn, improve plane utilization. We can easily synchronize the page offsets (i.e., zero) of two planes for all writes by assigning a new active block in Plane 1 ("Block 5"). In this example, the GC destination block (active block) "Block 7" of Plane 0 is assumed to be new and its page offset is zero; otherwise, we can assign a new active block as we do for Plane 1. As a result, the normal I/O writes "J" and "L" to Plane 1

Algorithm 3: Our proposed GC victim selection algorithm; among multiple candidate blocks, this picks the one that can construct the most multi-plane reads.

```
Input: GC candidate blocks (Blk[]), the number of GC candidate blocks (N_candidate),
       I/O queue (Q)
Output: GC source block number (i)
for i ←1 to N_candidate do
    for V ←each valid page in Blks[i] do
        v ← page offset of V;
        for R ← each "not-used" read request in Q do
            r ← page offset of R;
            if v == r then // I/O-parallelized GC read
                Count[i]++;
                mark R in Q as "used";
                break;
        end
    end
end
find the max of Count[i];
return i;
```

can get serviced in parallel with the GC writes "B" and "D". Compared to the baseline (Figure 5a), the utilization of Plane 1 increases as well. Note that the I/O read "I" parallelized with the GC read "C" has nothing to do with the effort of generating more multi-plane writes; it is a result of our base I/O-parallelized GC. When it comes to the end of GC execution, "Block 7" of Plane 0 and "Blocks 5 and 9" of Plane 1 can be used for furtur I/O writes, before each plane assigns a new active block.

Implementation – To maintain one additional active block for each plane, one extra 4-byte pointer indicating the block number is added to the FTL. Also, the worst-case fragmentation cost coming from keeping two active blocks is bounded by *"Number_of_Pages_per_Block - 1"*; it is very small, compared to the total capacity of a plane.

4.3 Towards More Multi-Plane Reads

Strategy – The chance of generating multi-plane reads in the I/O-parallelized GC is determined by the valid page offsets of the GC block and the page offsets of I/O reads. Since the valid page patterns significantly vary across blocks, we propose to select the best GC victim block among multiple candidates, instead of an arbitrary block that has the least number of valid pages. The most appropriate GC victim block indicates the one whose valid pages can make the most matches of page offsets with I/O read requests in the queue. One thing to keep in mind in finding the best victim block is that we are not allowed to increase the GC cost (i.e., the number of valid page movements) compared to the baseline – blocks with more valid pages can increase the opportunities of building multi-plane reads; however, this also increases the GC overhead we try to minimize its impact. Thus, when selecting the victim block, we first choose all the blocks with *the least number of valid pages* based on the baseline GREEDY algorithm (as shown in Figure 7b), and then, we pick the one with *the maximum number of valid pages aligned with the normal I/O read requests* currently waiting at the queue. Once GC victim candidate blocks are selected, Algorithm 3 picks one from them. More specifically, for each candidate block, it counts the matches of page offsets by scanning the valid pages in the block and the I/O reads in the queue. Finally, it returns the block with the highest count, which leads to the most multi-plane reads.

Example – With the employment of a new active block (discussed in Section 4.2), Figure 5c illustrates how our victim selection algorithm can increase the chances for generating multi-plane reads, and ultimately, improve the plane utilization. Instead of the block ("Block 2") selected by the baseline victim selection, our proposed victim selection algorithm picks another block ("Block 6") as the GC victim in Plane 0, assuming that both "Blocks 2 and 6" have the same number of valid pages. As a result, more I/O reads ("M", "O", and "P") on Plane 1 can be parallelized with GC reads ("C", "E", and "G") on Plane 0; in this scenario, we still parallelize I/O writes "J" and "L" with GC writes ("B" and "D") using a new active block. One can also observe that the increased multi-plane reads lead to the increase in the utilization of Plane 1, compared to the baseline (Figure 5a) and the enhancement for multi-plane writes (Figure 5b).

Timing overhead – Our proposed victim block selection mechanism (Algorithm 3) brings a timing overhead in examining which block would be the most beneficial among multiple candidates. Specifically, for each candidate block, its valid page offsets need to be compared with those of I/O reads in the queue, which is almost the same process as the proposed scheduler (Algorithm 2) goes through. This is a tolerable delay, since our analysis reveals that the maximum number of candidate blocks is at most 10; note that we limit this number by taking only the blocks with the least number of valid pages across the GC target plane. Executing this algorithm on top of a modern ARM's Cortex SSD processor takes at most a few microseconds, as examining a candidate block takes around hundreds of nanoseconds. Overall, the incurred timing overhead is much smaller than a flash read latency (115 microseconds in our evaluation setup); it is worthwhile to bear as plane parallelism increases with more multi-plane reads.

4.4 Discussion

Combination with other proposals – Our proposed I/O-par allelized GC with the two enhancements discussed above is quite effective in hiding the GC impact on I/O response times by simultaneously processing the majority of I/O requests and the GC operations. However, in concert with prior proposals (Section 3.3), our proposal can be expected to perfectly avoid GC-induced increase in I/O response times as well as increase the plane utilization. For example, during GC, our proposal can miss to serve (i) the I/O read whose page offset has no match with the GC read and (ii) the I/O read whose destination is the plane which is doing GC. Those I/O requests can get serviced using the preemptive GC [21]; that is, when such I/Os are given, the I/O-parallelized GC is paused, and those get serviced, and then the I/O-parallelized GC is resumed.

Applicability to Global GC – In a global GC, the source and destination blocks may reside in different chips, dies and planes. This configuration also has the plane under-utilization problem. When reading valid pages from a (source) plane in a certain die of a chip, all the paired planes in the die of the chip are under-utilized. Again, when the (valid) page data are written back to a (destination) plane in another die

Table 1: Configuration of the baseline SSD and flash.

SSD	1TB capacity, PCIe 3.0 4-lane host interface (12GB/s), sixteen 64GB flash chips [24], 4 channels, 4 flash chips/channel.
Flash chip [24]	MLC (2 bits per cell), 4 dies/chip, 2 planes/die, 1048 blocks/plane, 512 pages/block, 16KB page.
Timing params [24]	115us for page read, 1.6ms for page write, 3ms for block erase, 166MT/sec data transfer rate of the chip
FTL	GREEDY GC [5], wear-aware block selection, multi-plane read/write support, Priority levels: GC (highest), reads (second highest), writes (lowest)

of another chip, all their planes in the same die are under-utilized. In this setup, the under-utilized planes are located in different dies and chips. Therefore, in a global GC, our scheme can only parallelize read I/O requests with GC reads in the source block/plane/die/chip, and it can only parallelize write I/O requests with GC writes in the destination block/plane/die/chip.

5 EVALUATION METHODOLOGY

Framework: We use Disksim [4] with the Microsoft SSD extension [3] to evaluate the proposed scheme. This simulator is widely used in similar studies and provides a highly parameterized framework that implements important SSD functions such as GC, inter-chip parallelism, and intra-chip parallelism.

Baseline Configuration: Table 1 gives the configuration of our baseline system, which is a 1TB SSD with sixteen 64GB MLC flash chips, a typical configuration used in high-performa nce computing and enterprise environments. There are 4 memory channels with 4 chips connected to each channel. HI is a PCIe with 12GB/s bandwidth [26] that is high enough for all workloads we used (so host-to-SSD bus never gets saturated in our settings). We used the timing information and organization setting of a modern MLC flash chip by Micron Technology [24]. In this device, there are 4 dies per chip, 2 planes per die, 1024 blocks per plane, and 512 16KB pages per block. We also assume that the FTL employs a CWDP static allocation [17] (Channel first, Chip second, Die third, Plane last). The GC's victim block selection algorithm is the GREEDY algorithm, which is wear-aware [5]. Finally, the FTL scheduler always gives the highest priority to GC; reads have the second highest priority; and writes have the lowest priority.

Evaluated Systems: We evaluate and compare the results of the following three systems:

• **Baseline:** This uses the baseline I/O scheduler and GC selection algorithm. This system stops servicing I/O requests to either planes of a die, if one of the planes is performing GC.

• **GC-PAR:** This employs our I/O-parallelized GC; the controller tries to parallelize reads/writes with GC operations as much as it can. But, it employs the baseline GC victim selection.

• **GC-VIC:** This system employs our GC-PAR scheduler equipp ed with our enhanced victim block selection algorithm.

Workloads: Throughout our evaluation, we use the I/O traces from the MSR Cambridge suite [25]. Among all 36 traces, we selected 13 traces that are *write-intensive*. Table 2 gives important characteristics of our workloads in terms of write-to-read ratio, average write request size, and ratio of the I/O requests affected by GC. Note that our scheme only improves the service time of the requests affected by GC and does not touch the scheduling of the remaining requests. Figure 8 reports the percentage of four I/O types in our 13 workloads: (i) reads affected by GC, (ii) writes affected by GC, (iii) reads not affected by GC, and (iv) writes not affected by GC; 1.4% of all reads on average and 24.9% of all writes on average have the potential of getting improved by our scheme.

Table 2: Important characteristics of our workloads.

Name	Description	Write Ratio (%)	Write Size (KB)	GC- Affected I/O Ratio (%)
hm_0	Hardware monitor	64.5	7.4	29.7
prxy_0	Firewall/web proxy	96.9	8.3	41.4
mds_0	Media server	88.1	23.7	47.3
src2_0	Source control	88.7	8.1	45.2
stg_0	Web staging	84.8	24.9	45.9
ts_0	Terminal server	82.4	8.4	37.1
wdev_0	Test web server	79.9	12.6	33.7
rsrch_0	Research projects	90.7	10.9	43.7
prn_0	Print server	89.2	22.8	17.6
proj_0	Project directories	87.5	17.8	13.4
src1_2	Source control	74.6	19.1	15.1
usr_0	User home dir	59.6	40.9	12.1
web_0	Web/SQL server	70.1	29.9	13.3

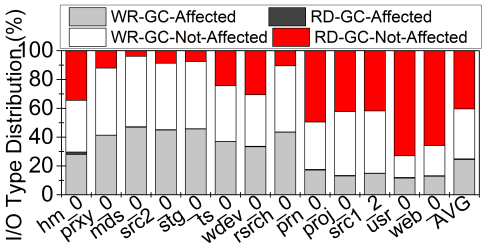


Figure 8: Distribution of four I/O types: WR-GC-Affected and RD-GC-Affected are the writes and reads affected by GC, whereas WR/RD-GC-Not-Affected are the remaining writes and reads, having no relation to GC.

6 EVALUATION

6.1 GC-Affected Resp. Time Enhancement

Figure 9 plots the read and write response times of the I/O requests affected by GC, optimized by our two proposed schemes, GC-PAR and GC-VIC. Compared to the baseline, GC-PAR and GC-VIC improve the read response times, on average, by 71% and 83%, respectively. For the write response times, both GC-PAR and GC-VIC bring, on average, 70% enhancement over the baseline. We make the following important observations:

• **Significantly-improved write response times:** As a result of parallelizing I/O requests with GC operations, GC-PAR significantly reduces the write response times. During GC, GC-PAR can remove almost all write requests in the queue and increase the plane utilization (see Section 6.2). In contrast, GC-VIC does not lead to an additional enhancement for write I/O, since it is aimed to increase the chances for constructing multi-plane reads.

• **Further improvements brought by GC-VIC in read response times:** GC-PAR could significantly enhance the read response time as well. Surprisingly, GC-VIC can find more opportunities for building multi-plane reads, which leads to a further (13% more) enhancement in the read response time. We observed that selecting other GC victim blocks (using GC-VIC) could remove almost all read I/O in the queue.

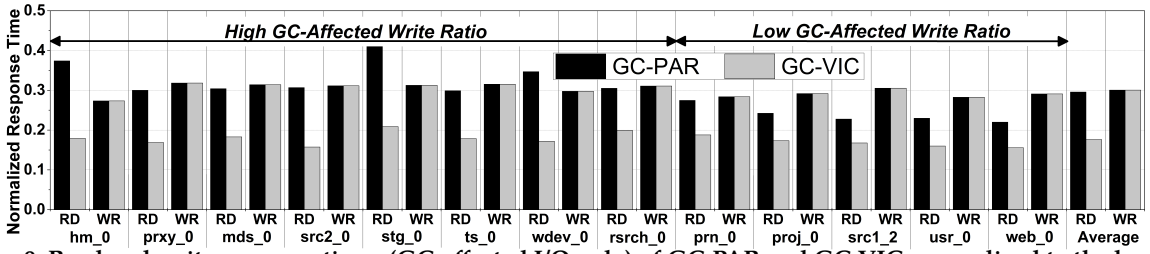


Figure 9: Read and write response times (GC-affected I/O only) of GC-PAR and GC-VIC, normalized to the baseline.

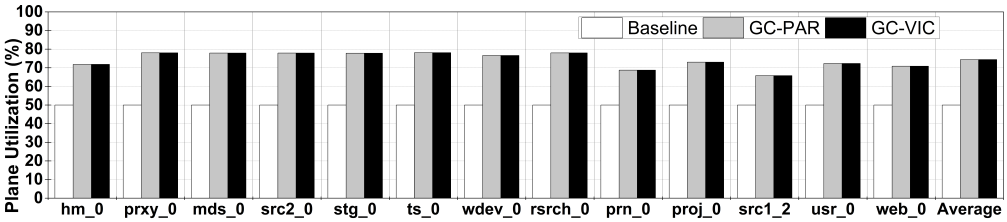
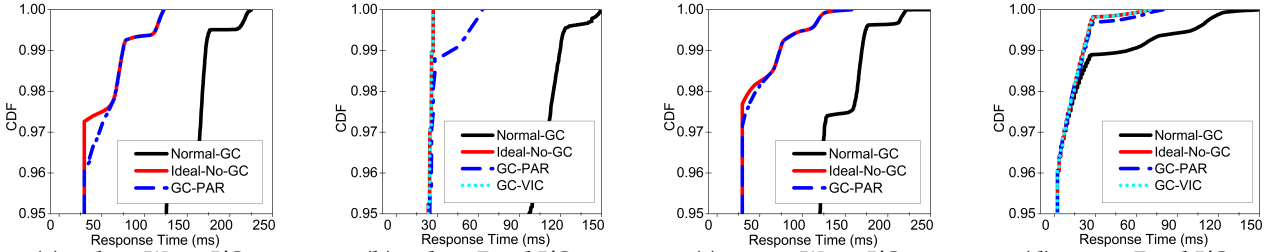


Figure 10: Plane utilizations of the baseline, GC-PAR, and GC-VIC in the two planes per die configuration.



(a) mds_0 Write I/O **(b) mds_0 Read I/O** **(c) proj_0 Write I/O** **(d) proj_0 Read I/O**

Figure 11: The CDFs of 95th percentile write and read I/O response times for our two representative workloads.

• **Varying effectiveness of GC-VIC:** The amount of improvement GC-VIC brings varies across the workloads. Workloads with *high GC-affected write ratio* get more benefits from GC-VIC than the ones with *low GC-affected write ratio*. This is because the applications in the former category have a tendency of generating consecutive GC invocations, which significantly increases the waiting time of blocked read requests. Accordingly, processing such reads suffering from the long-lasting stalls (by GC-VIC) is much more beneficial than doing it for applications with the low GC-affected write ratios.

6.2 Plane Utilization Improvement

Figure 10 compares the plane utilizations of our three evaluated systems. Compared to the baseline, GC-PAR and GC-VIC improve the plane utilization (on average) by 47.7% and 47.8%, respectively. The following analysis is worthwhile to note.

• **Always 50% in the baseline:** Only one plane is involved in the GC at any given time. The busy plane (involved in GC) can be any of the two planes; it is in either read or write. Thus, in a two-plane configuration, the utilization is always 50%.

• **Largely-improved utilizations:** GC-PAR could make both the planes busy in most of the GC period by successfully parallelizing the I/O requests in the queue with the GC operations. The magnitude of utilization improvement is mainly determined by (i) the number of read and write I/Os in the queue during GC, (ii) the number of page migrations in GCs, and (iii) the chances for constructing multi-plane commands based on I/Os in the queue and GC operations. According to our analysis, the number of GC page migrations is generally larger than the number of write I/Os in the queue; as a result, GC-PAR can process most of the write I/Os concurrently with GC writes. In case of the read I/Os, GC-PAR is effective to serve them during GC; more reads could be parallelized by GC-VIC.

• **Negligible benefits from GC-VIC:** The plane utilizations of GC-PAR and GC-VIC are almost the same, since the main contributor to the increase in plane utilization is not (multi-plane) reads, but writes. Note that the write latency is 10x longer than the read latency. However, we want to emphasize that GC-VIC is quite effective in reducing the read response times, despite its negligible impact on plane utilization.

6.3 GC Impact on Response Times

Figure 11 presents the 95th percentile CDFs of read and write response times for two representative workloads, presented in Figure 3 of Section 3.1. For each workload, the CDF of write response times compares the baseline (Normal-GC), the ideal (Ideal-No-GC), and GC-PAR systems; but, the CDF of read response times has one more system (GC-VIC). The closer the response times of GC-PAR and GC-VIC to those of the Ideal-No-GC, the less the GC impact on the SSD performance. Overall, GC-PAR and GC-VIC mostly overlap with Ideal-No-GC, which indicates that a majority I/O requests stalled during GC in the baseline system could be parallelized with GC, and in turn, the performance instability caused by GC is almost completely eliminated by our schemes.

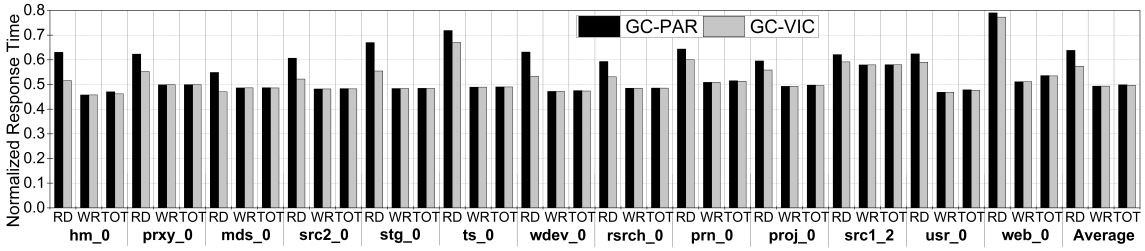


Figure 12: Normalized read, write, and total response times (entire I/O) of GC-PAR and GC-VIC to that of the baseline.

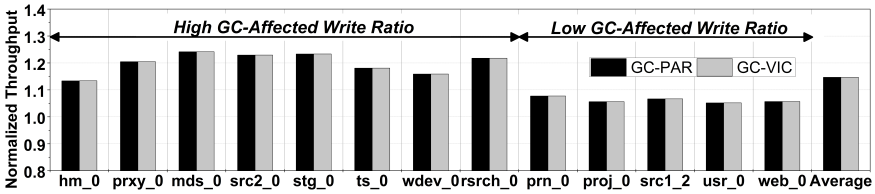


Figure 13: The device throughputs achieved by GC-PAR and GC-VIC, normalized to the baseline.

6.4 Overall Response Time Analysis

One might also be interested in the impact of optimizing only GC-affected reads and writes on the entire I/O response time. Figure 12 plots the read, write, and total response times (of both the GC-affected and non GC-affected I/O) achieved by GC-PAR and GC-VIC, *normalized* to the baseline. On average, GC-PAR and GC-VIC improve 36.2% and 42.6% for the read response times, 50.7% and 50.7% for the write response times, and 50.1% and 50.3% for the total response times. We point out two points in this analysis.

• **Impact of enhancing a few GC-affected reads:** Optimizing the response times of a small number of GC-affected reads poses a significant impact on the overall read response time. This is because a majority of the non-GC-affected reads have at most hundreds of microseconds, while a few GC-affected reads take up to tens of milliseconds. As a result, despite their small fraction, the GC-affected reads are the major contributor to the overall high read response time.

• **Little difference between the writes and total I/Os:** One can also see that the amount of improvement in the total I/O requests is quite similar to the enhancement in the write I/O only. This is because the ratio of the GC-affected writes is much higher than that of the GC-affected reads; hence, even though there is a significant response time gain in reads, this does not contribute to the overall I/O response time.

6.5 Throughput Improvement

We now discuss the impact of our schemes on the entire storage system throughput. Figure 13 plots the throughput improvements achieved by GC-PAR and GC-VIC. Both these schemes improve throughput by 14.7%, on average, over the baseline. Considering the fact that our two schemes work only during GC, the high overall throughput improvement is a remarkable outcome. In practice, in write-intensive applications, a high-overhead GC is frequently invoked, once the SSD becomes dirty, where our schemes could be a promising option. More specifically, the throughputs of the applications with high GC-affected write ratios increase over 10% (i.e., 13.4–24.2%) by our schemes, whereas the achievements remain below 10% (i.e., 5.1–7.7%) for the applications with low GC-affected ratios.

6.6 Four Plane Configuration Results

Even though our baseline has two planes per die, there are several flash memories in the market today with an internal structure of four planes per die [2]. Observing this, we evaluated our schemes in a four-plane device as well. In the four plane configuration, the improvements in the GC-affected I/O response times by GC-PAR and GC-VIC are similar to (or a little higher than) those observed in the two plane configuration – 75% and 84% for reads; 71% and 71% for writes, respectively. In fact, there are more chances for constructing multi-plane commands in the four plane configuration, as three planes out of four remain idle during GC. However, a further significant enhancement is hard to expect, since almost all I/O requests in the queue could get serviced during GC even in the two plane die. In the baseline, the utilization is always 25% as only one of the four planes is involved in GC and the remaining three planes remain idle. Our schemes can increase the utilization up to (on average) 38.1%; unfortunately, this increased utilization is still quite low. This is because the average number of I/O requests in the queue is relatively small.

7 RELATED WORK

SSD designers have investigated various approaches to relieve the significant performance degradation brought by GC. Such efforts can be categorized into three groups:

• **Segmenting and distributing GC [16, 19, 20]:** Works in this group (i) segment the whole GC into sub-operations (read, writes, and erases) and (ii) distribute them across normal I/O requests. For example, Jung et al. [16] exploit I/O slacks (between the actual service time and the defined deadline) to process the distributed sub-GC operations. Even though this work helps all I/O requests meet their deadlines, the actual I/O response times increase due to the additional GC burden. Lee et al. [20] assign higher priority to normal I/O requests over GC operations; so, when a new I/O request arrives, it can get serviced first, while the GC is suspended. However, in enterprise environments where I/O requests are quite intensive and GC invocations are consecutive, such an approach can quickly fail to secure free spaces for incoming writes.

• **Exploiting chip idleness [22]:** Works in this category schedule the I/O requests destined for flash chips where GC is not invoked first and the ones heading to GC-busy flash later. However, in a realistic I/O scenario, the number of I/O requests stalled due to GC is not a few; hence, the overall response times can remain quite high. Based on dynamic mappings [17], one might suggest to write data into an idle flash, instead of a fixed location which is busy for doing GC. However, this strategy cannot work for reads, since a read needs to access the place where the data exists.

• **Leveraging data redundancy [12, 32]:** These works maintain multiple copies of data (like RAID), and utilize the one in a GC-free flash chip among multiple replicas across different flash chips. He et al. [12] selectively keeps data based on their recency and blocking probability, whereas Yan et al. [32] propose a customized replica layout with a consideration of resource layout and wear-leveling. Despite their efforts, the data redundancy can still impose a high overhead on device capacity, performance, and lifetime.

Even though these works help to alleviate the GC-induced problems, in all cases, while a plane is involved in GC, the other planes still remain idle; that is, *the plane under-utilization problem is not addressed in any of these prior works*. Our proposal can be combined with them to further improve the SSD performance.

8 CONCLUSION

The GC is a main contributor to long I/O response times exhibited by state-of-the-art SSDs, which has brought numerous proposals. Even though all the prior works are quite effective in hiding the negative impact of GC on I/O response times, we observe that flash resources (planes) are significantly under-utilized during GC. Motivated by this, we attempt to use the under-utilized plane(s) by parallelizing the blocked I/O requests along with GC operations. We also propose novel strategies to increase the chances for maximizing the plane-level parallelism. Our extensive evaluations using 13 workloads reveal that, in a two planes-per-die configuration, the proposed schemes increase the plane utilization by 47.8%; and consequently, improve the response times of the GC-affected reads and writes by 83% and 70%, respectively.

ACKNOWLEDGMENTS

This work is supported in part by NSF grants 1302557, 1213052, 1439021, 1302225, 1629129, 1526750, and 1629915 and a grant from Intel. Dr. Jung's research is partly supported by Yonsei Future Research Grant (2017-22-0105), NRF 2016R1C1B2015312, DOE DEAC02-05CH11231, IITP-2018-2017-0-01015 and NRF 2015M3C4A7065645.

REFERENCES

[1] Abdul R. Abdurrab, Tao Xie, and Wei Wang. 2013. DLOOP: A Flash Translation Layer Exploiting Plane-Level Parallelism. In *ISPDP*.
[2] Michael Abraham. 2012. NAND Flash Architecture and Specification Trends. In *Flash Memory Summit*.
[3] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. 2008. Design Tradeoffs for SSD Performance. In *USENIX ATC*. http://www.microsoft.com/en-us/download/details.aspx?id=52332.
[4] John S. Bucy, Jiri Schindler, Steven W. Schlosser, and Gregory R. Ganger. 2008. The DiskSim Simulation Environment Version 4.0 Reference Manual. In *CMU-PDL-08-101*. http://www.pdl.cmu.edu/DiskSim/.
[5] Werner Bux and Ilias Iliadis. 2010. Performance of Greedy Garbage Collection in Flash-based Solid-State Drives. In *Journal of Performance Evaluation, VOL. 67, Issue. 11*.
[6] Yu Cai, Onur Mutlu, Erich F. Haratsch, and Ken Mai. 2013. Program Interference in MLC NAND Flash Memory: Characterization, Modeling, and Mitigation. In *ICCD*.
[7] Feng Chen, Rubao Lee, and Xiaodong Zhang. 2011. Essential Roles of Exploiting Internal Parallelism of Flash Memory based Solid State Drives in High-Speed Data Processing. In *HPCA*.
[8] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. 2009. A Survey of Flash Translation Layer. In *Journal of System Architecture, Vol. 55, Issue 5-6*.
[9] Jinhua Cui, Weiguo Wu, Xingjun Zhang, Jianhang Huang, and Yinfeng Wang. 2016. Exploiting Latency Variation for Access Conflict Reduction of NAND Flash Memory. In *MSST*.
[10] Peter Desnoyers. 2014. Analytic Models of SSD Write Performance. In *ACM Transactions on Storage, Vol. 10, Issue 2*.
[11] ONFI Working Group. 2011. Open NAND Flash Interface Specification 3.0. http://onfi.org/.
[12] Bingsheng He, Jeffrey Xu Yu, and Amelie Chi Zhou. 2015. Improving Update-Intensive Workloads on Flash Disks through Exploiting Multi-Chip Parallelism. In *IEEE Transactions on Parallel and Distributed Systems, Vol. 26, No. 1*.
[13] Benny Van Houdt. 2013. A Mean Field Model for a Class of Garbage Collection Algorithms in Flash-based Solid State Drives. In *SIGMETRICS*.
[14] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Shuping Zhang. 2011. Performance Impact and Interplay of SSD Parallelism through Advanced Commands, Allocation Strategy and Data Granularity. In *ICS*.
[15] Amber Huffman. 2017. Addressing IO Determinism Challenges at Scale with NVM Express – Part 2: Renegotiating the Host/Device Contract. In *NVMW*.
[16] Myoungsoo Jung, Wonil Choi, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut Kandemir. 2014. HIOS: A Host Interface I/O Scheduler for Solid State Disks. In *ISCA*.
[17] Myoungsoo Jung and Mahmut Kandemir. 2012. An Evaluation of Different Page Allocation Strategies on High-Speed SSDs. In *Usenix Hotstorage*.
[18] Myoungsoo Jung and Mahmut Kandemir. 2013. Revisiting Widely Held SSD Expectations and Rethinking System-Level Implications. In *SIGMETRICS*.
[19] Myoungsoo Jung, Ramya Prabhakar, and Mahmut Kandemir. 2012. Taking Garbage Collection Overheads Off the Critical Path in SSDs. In *Middleware*.
[20] Junghee Lee, Youngjae Kim, Galen M. Shipman, Sarp Oral, and Jongman Kim. 2013. Preemptible I/O Scheduling of Garbage Collection for Solid State Drives. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 32, No. 2*.
[21] Junghee Lee, Youngjae Kim, Galen M. Shipman, Sarp Oral, Feiyi Wang, and Jongman Kim. 2011. A Semi-Preemptive Garbage Collector for Solid State Drives. In *ISPASS*.
[22] Bo Mao and Suzhen Wu. 2015. Exploiting Request Characteristics and Internal Parallelism to Improve SSD Performance. In *ICCD*.
[23] Micron. 2006. NAND Flash 101: An Introduction to NAND Flash and How to Design It in to Your Next Product. In *Micron TN-29-19*.
[24] Micron. 2016. NAND Flash Memory, MT29F128G08CBECB, MT29F256G08CEECB, MT29F512G08C[K/M]ECB, MT29F1T08CUECB.
[25] Dushyanth Narayanan, Austin Donnelly, and Antory Rowstron. 2008. Write Off-Loading: Practical Power Management for Enterprise Storage. In *USENIX FAST*. http://iotta.snia.org/traces/388.
[26] PCI-SIG. 2012. PCI Express Base 3.0 Specification. http://pcisig.com.
[27] Chris Petersen. 2017. Addressing IO Determinism Challenges at Scale with NVM Express. In *NVMW*.
[28] Narges Shahidi, Mohammad Arjomand, Myoungsoo Jung, Mahmut Kandemir, Chita Das, and Anand Sivasubramaniam. 2016. Exploring the Potentials of Parallel Garbage Collection in SSDs for Enterprise Storage Systems. In *SC*.
[29] Ji-Yong Shin, Zeng-Lin Xia, Ning-Yi Xu, Rui Gao, Xiong-Fei Cai, Seungryoul Maeng, and Feng-Hsiung Hsu. 2009. FTL Design Exploration in Reconfigurable High-Performance SSD for Server Applications. In *ICS*.
[30] Robert Sykes. 2012. Critical Role of Firmware and Flash Translation Layers in Solid State Drive Design. In *Flash Memory Summit*.
[31] Arash Tavakkol, Pooyan Mehrvarzy, and Hamid Sarbazi-Azad. 2016. TBM: Twin Block Management Policy to Enhance the Utilization of Plane-Level Parallelism in SSDs. In *IEEE CAL, Vol. 15, Issue 2*.
[32] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminatahan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. 2017. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *USENIX FAST*.