Taking Garbage Collection Overheads off the Critical Path in SSDs

Myoungsoo Jung, Ramya Prabhakar, Mahmut Taylan Kandemir

The Pennsylvania State University

Abstract. Solid state disks (SSDs) have the potential to revolutionize the storage system landscape, mostly due to their good random access performance, compared to hard disks. However, garbage collection (GC) in SSD introduces significant latencies and large performance variations, which renders widespread adoption of SSDs difficult. To address this issue, we present a novel garbage collection strategy, consisting of two components, called *Advanced Garbage Collection* (AGC) and *Delayed Garbage Collection* (DGC), that operate collectively to migrate GC operations from busy periods to idle periods. More specifically, AGC is employed to defer GC operations to idle periods in advance, based on the type of the idle periods and on-demand GC needs, whereas DGC complements AGC by handling the collections that could not be handled by AGC. Our comprehensive experimental analysis reveals that the proposed strategies provide stable SSD performance by significantly reducing GC overheads. Compared to the state-of-the-art GC strategies, P-FTL, L-FTL and H-FTL, our AGC+DGC scheme reduces GC overheads, on average, by about 66.7%, 96.7% and 98.2%, respectively.

1 Introduction

Over the past decade, different computing domains, ranging from high performance computing and enterprise server platforms to embedded systems, are adopting SSDs [1] [2], due to their technical merits such as good random access performance, low power consumption, higher robustness to vibrations and temperature, and higher read/write bandwidth than hard disks [3]. NAND flash capacity is increasing by two to four times every two years [4] and SSD prices are expected to continue to fall to the extent of becoming cheaper than high-speed hard disk [5], which can in turn enable widespread deployment in diverse computing domains.

Modern SSDs internally employ a flash translation layer (FTL), managing two intrinsic properties of NAND flash memory to emulate it as a block device: first, no write is allowed before erasing a block, called the *erase-before-write* property. Second, NAND flash makers adopt a write sequence in a block due to the page-level program disturbance behavior [6] [7], which has a deep relationship with modern NAND flash memory reliability and data integrity. In addition to the erase-before-write property, this *in-order-update* property in a block necessitates *out-of-place updates* for write operations. To enable such out-of-place updates in the SSD, FTL remaps the logical addresses that conventional block devices provide to the physical addresses presented by the NAND flash memory. In addition, the FTL employs a garbage collector, which reclaims the invalid pages, incurred during the out-of-place update process. At a high-level, the garbage collector relocates valid pages in certain blocks to new blocks, which are

prepared in advance, and erases them in order to make rooms for new writes. This operation is referred to as *garbage collection* (GC).

The biggest problem with existing garbage collectors is that their worst-case latency can be as high as 64∼128 times than that of normal write operations [8] [9]. Our own experiments show that GCs introduce numerous blocking I/Os, and once a GC operation begins, the response time of write operations on SSD increases substantially. Further, GC overheads significantly reduce available bandwidth in most recent commercial SSDs. Unfortunately, this interaction between the GC and writes introduces significant performance variations/degradations during I/O, which may not be acceptable in many I/O-intensive computing environments.

Motivated by this, most current FTLs optimize mapping policies to minimize the number of GC invocations and hide their undesired latency. For example, existing buffer management schemes are specialized to reduce the number of writes to NAND flash. Also, some SSDs employ partial block cleaning techniques [10] [11] that attempt to provide stable GC performance by balancing the number pages/blocks between production and consumption of them using an extra non-volatile buffer. However, there is yet another dimension to avoiding GC overheads. Specifically, *the presence of idle I/O times in workloads can be exploited by shifting garbage collections from busy periods to other periods where they can be accommodated with minimum performance penalty.*

In this paper, we propose a novel GC strategy, an approach that removes GC overheads and provides stable I/O performance in SSDs during the I/O congestion periods. Our proposed GC strategy consists of two components, called *Advanced Garbage Collection* (AGC) and *Delayed Garbage Collection* (DGC). More specifically, AGC tries to secure free blocks and remove on-demand GCs from the critical path *in advance*, so that users do not experience GC-induced latencies during the I/O-intensive periods, whereas DGC handles the collections that AGC could not handle, by delaying them to future idle periods. Since our approach mainly reschedules garbage collections, it can work with any existing FTL.

Shifting GC operations however can increase program/erase (PE) cycles, which makes the life time of SSDs shorter. For example, if a garbage collector heedlessly reclaim blocks, which have the potential to be further utilized or used for new writes, it can introduce unnecessary PE cycles in relocating valid pages within them. To prevent this problem, we propose two different implementations for AGC, called *look-ahead garbage collection* and *proactive block compaction*, based on the duration of the idle period under consideration and the style of GC detection. Specifically, the look-ahead GC utilizes short idle periods and reclaims block based on the online information extracted from a device-level queue, whereas the proactive block compaction targets long idle periods and perform GCs only related to fully utilized blocks.

As shown in Figure 1, the main **goal** behind our strategies is to perform as many GCs as possible in the *idle periods*. Our **contributions** in this paper can be summarized as follows:

• *Eliminating GC overheads:* When using our garbage collection strategies, applications do not experience GC overheads. This is because our strategies successfully migrate on-demand GCs from busy periods to idle periods. Experimental results show that our proposed GC strategies result in stable I/O performance under various types of workloads.

• *Avoiding additional GC operations:* The proposed schemes (AGC and DGC), when applied together, do not increase the original number of GC operations. They only reschedule the GC operations that would be invoked soon by

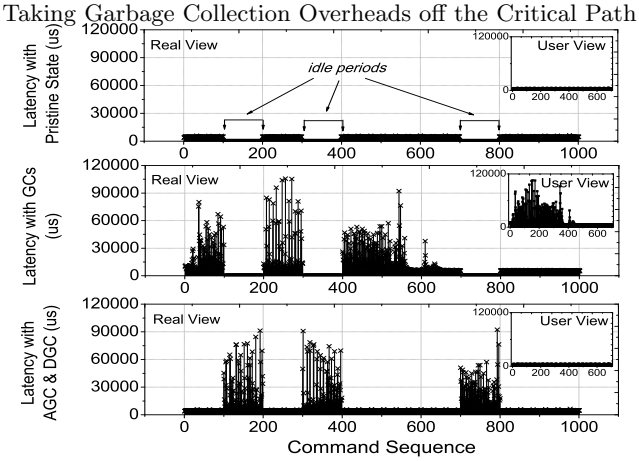


Fig. 1: Overview and comparison of SSD latencies with/without our proposed GC strategies (AGC and DGC), tested by random write pattern with 2048KB request size. Note that AGC and DGC shift GC overheads to idle periods (as shown in the real view), thereby providing stable I/O performance like a pristine state (as shown in the user view).

speculating their GC activities and identifying appropriate idle periods based on their durations (short or long). If the frequency with which idleness occurs is not high enough, then the GC invocations are postponed to future idle periods without affecting latency of I/O operations. As a result, we do not incur any additional GC operations.

• *Compatibility with underlying FTL schemes:* Most optimized garbage collectors proposed in the literature need additional non-volatile (NV) blocks on the SSD and/or require customized FTLs for successful execution. In contrast, our proposed schemes do *not* require any extra NV blocks or major modifications to the existing data structures, and can therefore work with diverse FTLs.

To test the effectiveness of our GC schemes, we implemented them in a simulator that models bus-level transactions and collected statistics using a variety of workloads. Our experimental results show that the proposed schemes reduce GC overheads (without causing additional write/erase operations) between 66.7% and 98.2%, in terms of the worst-case response time. Further, our schemes prepare free blocks in advance to help prevent the block thrashing problem. Consequently, they reduce the number of block erase operations by 16.6%, compared to a conventional FTL.

2 Background and Related work

2.1 Flash Translation Layer

The NAND-based flash consists of physical *blocks*; a physical block is the erase unit and is composed of several *pages*, which are the read/write units in the NAND flash. One of the drawbacks of the NAND flash is that a page needs to be updated in-place within the block. In addition, writes to a formerly written page are not allowed before erasing the entire block corresponding to it. Since a block is much larger than a page and an erase operation is more expensive than a write operation, the NAND flash alone is not sufficient to build an SSD. Therefore, a *Flash Translation Layer* (FTL) is required within the SSD to prepare physical blocks ahead of time. Whenever an SSD receives write requests, it forwards

them to a temporal block called an *update block*. The FTL then serves requests by physically (in-place) writing them into a block. This allows logical out-of-order update by mapping addresses between the in-place and out-of-place update sequences.

The FTL also hides the latency of block erase and unnecessary read/write operations in copying valid pages that are live in a block [8] [12]. Similarly, to provide data consistency and coherence between the original block (also called the *data block*) and the update block, the FTL internally maintains mapping information and address translations. In this way, by internally managing the flash specific characteristics, the FTL provides compatibility with commodity storage systems. Typically, based on the number of the data block(s) and update block(s) in a logical block, FTLs are classified into three types. Block-mapping FTL manages a logical block by combining one data and one update block (1:1 mapping). Hybrid mapping FTLs manage a logical block by composing n data block(s) and m update block(s) ($n : m$ mapping). Finally, pure-page mapping FTLs leverage only update blocks for serving I/Os, and can allocate them in any physical page location.

2.2 Garbage Collection

If the FTL does not have enough free pages in its update block, it has to perform GC in an attempt to reclaim available blocks to which write request, can be forwarded. This type of GC is referred to as *update block reclaiming GC*. Similarly, in cases where the FTL has insufficient free blocks, it should secure free blocks by evicting some other logical blocks, called *free block reclaiming GC*. These processes require migrating all valid pages from the update and original blocks to a new free block (called *page migration*) and erasing these two blocks. Thus, *the GC latencies are typically much larger than that of normal I/O operations*. In addition, the FTL carries out these GC operations during runtime on a need-basis, meaning that the collections are postponed as long as the SSD can accept new data and are only performed when required. The reason why GCs are executed on demand is that a block erase operation, which is part of the garbage collection activity, can significantly affect the SSD's lifetime and reliability [8]. For example, if a garbage collector heedlessly reclaim blocks, which have the potential to be further utilized or used for new writes, it can introduce unnecessary program/erase (P/E) cycles for relocating valid pages within them. Due to this property, GC latencies typically piggyback on ordinary I/O requests, leading potentially to very high I/O latencies. Several FTL based studies [9] [8] attempted to reduce GC overheads and hide their latencies. Other approaches like the real-time GC [10] and the partial block cleaning [11] [13] aimed to provide stable GC performance by balancing the number pages/blocks between the production and consumption of them using an extra non-volatile buffer.

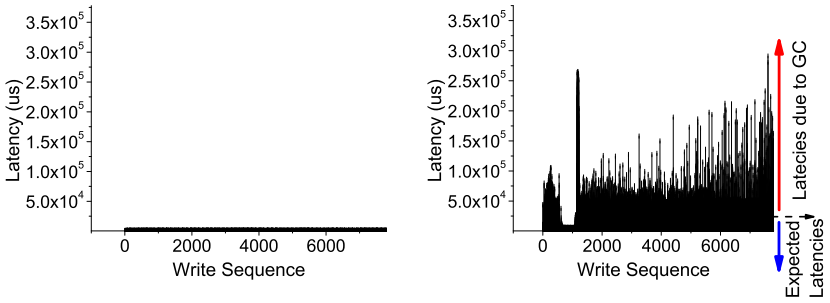
3 Impact of Garbage Collection in Commercial SSDs

To measure the impact of GCs in state-of-the-art SSDs, we evaluated their latencies and bandwidth with/without GCs.[1]

Latency impact: Figures 2a and 2b plot normal latencies and extra latencies due to GCs, respectively. In this empirical test, we used a 256GB MLC-based

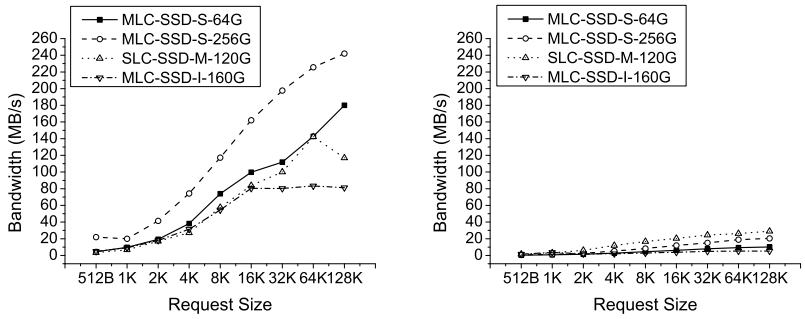
[1] All SSDs we tested (e.g., 64GB, 256GB, 160GB MLC-based SSDs and 120GB SLC-based SSD) were deployed in 2010 $\sim$ 12.

SSD which employs two 128MB internal DRAM buffers and measure latencies of individual I/O operations using ULINK's DriveMaster [14]. The DriverMaster is a commercial tool that captures detailed storage-level latencies and tests SSDs in a physical level. We wrote data with 1MB transfer size into the SSD using a random pattern. While a pristine SSD was used for the normal latency test, we later filled the SSD completely and introduced a one hour period before evaluating the GC latencies. As illustrated in Figure 2, GCs introduce numerous blocking I/Os, and once a GC operation begins, the response time for write operations increases substantially. Further, irrespective of the large amount of idleness that we artificially introduced, high latencies of GC are observed from the beginning of the GC latency test.



(a) Normal Latencies (b) Latencies with GCs

Fig. 2: Latency comparison for a random write access pattern with 1MB request size using a real MLC-based SSD.



(a) Normal Bandwidth (Pristine) (b) Bandwidth with GCs (Fragmented)

Fig. 3: System throughput for four state-of-the-art SSDs (different vendors and NAND types). Note that all SSDs tested suffer from significant performance degradation once garbage collections begin.

Bandwidth impact: From a system designer viewpoint, throughput might be a more important performance metric. In this test, we measured performance with/without GCs of four commercial SSDs (three 64GB, 256GB and 160GB MLC-based SSDs and one 120GB SLC-based SSD) using Intel Iometer [15].

Figures 3a and 3b plot bandwidth with the pristine state and bandwidth with fragmented state, respectively. To make an SSD fragmented, we first wrote 4KB data in random order and fully utilized its storage space. Similar to the latency impact test, we artificially introduced a one hour idle time before evaluating this bandwidth impact test. As shown in Figure 3, GC overheads significantly reduce available bandwidth in all four commercial SSDs tested, regardless of the idle time introduced.

4 High Level View of GC Scheduling

To avoid performance degradation and variations caused by GCs, we propose novel garbage collection strategies. Unlike previous GC strategies that reduce the number of GC invocations or GC overheads at runtime, our proposed GC strategies fully utilize device-level idle times, which are invisible to the user, to perform GC activities. To efficiently exploit such idle times, we classify them into two groups based on their lengths. Using our idle period classification, we then invoke different types of GCs to ensure that the user does not experience long GC-induced latencies. Our approach allows the other components of FTL to work without any modification, making our approach highly portable.

4.1 Idle Period Classification

Short idle periods: Several applications exhibit short idle intervals interleaved with parallel I/O requests in a device-level command queue [16], which allows a storage system to determine actual data transfer times and implement out-of-order execution of I/O commands. To enable this, most host interface protocols bring I/O commands, along with preinformation including request type, addresses and request size, to the storage system before the actual data transfer begins.

To measure how many commands with their preinformation are available at a given time and the duration of idle periods, we executed Intel Iometer workloads [15] and employed a 265GB MLC-based SSD that used in Figure 2. The LeCroy protocol analyzer [17] is used for analyzing the SATA protocol at the physical layer. We observed that 3-17 commands are delivered to the device-level command queue in parallel before the actual data communication starts, and the storage-level idle times experienced by the I/O requests vary between 1.8 μs and 15.2 ms, based on the operation type and transfer size.

This storage-level short idle periods that we measured can be detected by looking through the I/O commands with their preinformation. Specifically, one can preview I/O commands in the queue before they get executed, and identify the short idle intervals between successive I/O commands. Even though this interval is short, one benefit gained from utilizing these short idle intervals is that it allows one to investigate a request through preinformation and accurately predict what will happen to the request during the idle time. Each short idle period can be expressed as follows:

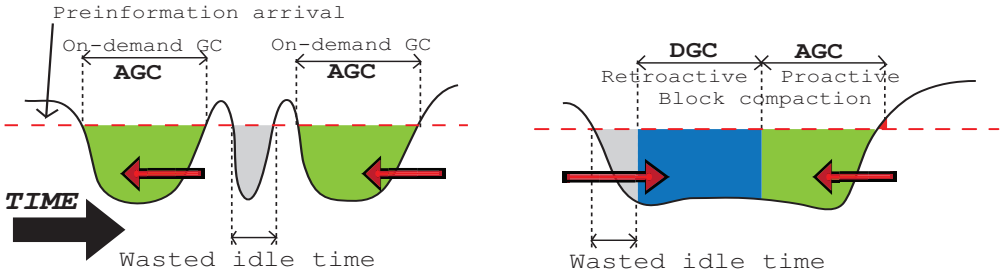
$$T_{short-idle} = t_{start_{i+1}} - (t_{start_i} + t_{exe_i} * l_i) \quad : \forall i, 1 < i \leq n, \tag{1}$$

where i+1 denotes the index of the I/O command following the i^{th} I/O command in the queue, t_{start_i} denotes actual transfer start time, t_{exe_i} is the execution time based on a page, and l_i is the page length of I/O command i. Clearly, short idle

periods exist only if $T_{short-idle}$ is larger than zero and there are I/O commands sitting in the queue (i.e., at least two commands). Here, n depends on the queue size accommodated by the host interface nuance. For example, NCQ [16] provides 32 entries, whereas TCQ [18] typically provides 256 entries. AGC exploits just two entries for previewing the I/O commands.

Long idle periods: We also observed that many applications exhibit relatively long idle periods with no enqueued I/O commands. We classify an idle period as a long idle period if its length is larger than a certain threshold [19] [20] [21]. The fraction of I/O instructions that experience these long idle periods ranges between 38% and 83% under various workloads tested [22] [23] when the threshold is set to 1 sec. Note that, to detect these idle periods, we cannot take advantage of the device-level command queue and preinformation since it is empty most of the time. Consequently, long idle periods should be handled differently.

Depending on whether idle periods are short, long or none, our proposed strategies schedule GC operations and secure free blocks differently.



(a) short idle utilization (b) long idle utilization

Fig. 4: A high-level view of our proposed GC strategy and idle time utilization.

4.2 Shifting Garbage Collection Overheads to Idle Periods

We start by observing that scheduling a GC on an arbitrary idle period can introduce extra block reclaimings (P/E cycles) and reduce opportunities for block reuse. This can in turn potentially shorten SSD lifetime and affect its reliability. Therefore, in our proposed schemes, we migrate the GC operations to *carefully-chosen* idle periods without increasing the original number of GC operations. We also minimize the overheads incurred by on-demand GC invocations by securing the available free blocks as much as possible in advance during the idle periods.

We explore two different strategies for shifting GC overheads, depending on the amount of idleness and on-demand GC needs, as shown in Figure 4. First, short idle periods are mainly exploited by shifting on-demand GCs that will be invoked during busy periods (Figure 4a). In this case, the garbage collector monitors upcoming device-level I/O tasks to determine when a collection will be needed and performs the necessary tasks *proactively*. If there is no on-demand GC need, the garbage collector performs block compactions to reclaim fully-occupied blocks thereby retrieving free blocks in advance (Figure 4b). We refer to this strategy as the *Advanced GC (AGC)*. Second, if the amount of short idleness is not sufficient to avoid on-demand GC invocations at a certain point, our proposed GC strategies prevent them from being invoked on the critical path by delaying the GC execution (Figure 4b); we refer to this strategy as the

Delayed GC (DGC). In other words, AGC shifts GC activities to idle periods in advance, whereas DGC handles the on-demand collections that AGC could not handle, by delaying the GC invocations to future idle periods.

These different GC strategies based on the type of idle periods and GC needs allow our strategies to shift GCs from busy periods to idle periods, as illustrated in Figure 1. At the same time, they help us minimize the potential side effects on SSD reliability and eliminate the extra storage space requirement in the SSD for the operation of the proposed schemes.

5 Implementation of our GC strategies

Recall that we quantified the impact of garbage collection on commercial SSDs in Section 3. To alleviate the overheads caused by garbage collections, we classified the types of idle periods in Section 4 and presented a high-level view of our proposed approach. We next describe the technical details of AGC and DGC in Sections 5.1 and 5.2, respectively. Section 5.3 discusses how AGC and DGC works together.

5.1 Details of Advanced GC Strategy (AGC)

AGC tries to remove the on-demand GCs from the critical path and secure free blocks in advance so that users do not experience long GC-induced latencies during the I/O congestion periods. Depending on the type of the idle period we are dealing with, one can implement AGC in two different ways. The look-ahead garbage collection (Section 5.1) is a type of AGC that targets on-demand collections by utilizing short idle periods, whereas the proactive block compaction (Section 5.1) secures free blocks by utilizing long idle periods.

Look-ahead Garbage Collection To shift GC invocations to earlier idle periods, this scheme exploits the device-level command queue and short idle periods. It starts by calculating the number of GC operations that can be executed in short idle periods. In this step, the look-ahead GC checks the queue entries and extracts I/O request information such as the length of I/O request and the Logical Sector Number (LSN) and associated Logical Block Number (LBN). It then finds the Physical Block Numbers (PBNs) corresponding to the LBN by looking up the mapping table of the underlying FTL. The look-ahead garbage collector then checks whether the available space, especially the number of free pages, is sufficient to service the I/O request of the specified size (length). If not, an update block reclaiming GC is required.

Once the need for GC is identified, our scheme next calculates a GC latency in order to accurately perform on-demand GC in advance. Let κ denote the number of physical blocks per logical block (e.g., in a block mapping scheme, the value of κ is one. On the other hand, if the system employs a 2:8 hybrid mapping scheme, κ can be up to ten). Further, let t_{load}, t_{write}, and t_{erase} denote execution latencies for page load (read), page write and block erase operation, respectively, and let t'_{write} represent the time for writing metadata to confirm the fact that a certain physical block was erased after GC (this helps to ensure mapping consistency in the FTL). Since the look-ahead GC knows PBN(s) for the logical block and has all the relevant mapping information, it can determine the number of valid pages for the PBN(s); say, n_{page}^{valid}. In this way, for each I/O

command i that is involved in the GC, its GC latency (T_{gc_i}) can be calculated using the following expression:

$$T_{gc_i} = \underbrace{(t_{load} + t_{write}) * n_{page}^{valid}}_{page\ \ migration} + \overbrace{(t_{erase} + t'_{write}) * \kappa}^{block\ \ cleaning}. \quad (2)$$

This expression captures the page migration latency for each valid page from the update/data block to a free block, as well as the block cleaning latency for these blocks. Typically, t'_{write} is approximately the same as the latency it takes to write one page (t_{write}). This is because the metadata is designed to fit in a single page to reduce the overhead of storing the metadata itself. The total amount of time taken by the look-ahead GC to perform collections over n blocks is given by ($\sum_{i=1}^{n} T_{gc_i}$). Using the GC latencies of individual blocks, we determine the number of blocks (n) that can be reclaimed at runtime, under the constraint that the total GC time for the determined number of blocks is less than or equal to the short idle time given by Equation (1). Once AGC determines the number of blocks, n, to be claimed, it performs look-ahead GCs for these n blocks in advance.

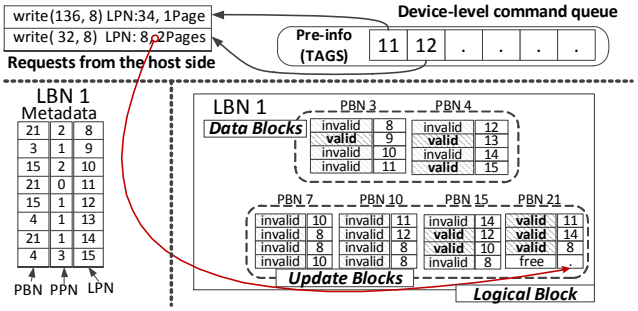


Fig. 5: An example of the look-ahead GC with a hybrid mapping scheme. By inquiring the mapping information, our AGC scheme figures out that the GC for LBN 1 will be invoked soon.

For instance, in Figure 5, the look-ahead GC identifies the request with an LSN of 32 and I/O size of eight sectors. Since the logical block corresponding to that request has only one free page, our scheme executes the GC operation in the short idle time. In addition to providing stable and better SSD performance, this implementation performs GCs *only when* an on-demand GC is about to occur and the short idle periods are suitable to perform GC. Therefore, it ensures a similar level of reliability compared to a standard FTL.

Proactive Block Compaction In order to exploit long idle times that could not be exploited by the look-ahead GC, we propose a proactive block compaction mechanism strategy. In this strategy, we detect the blocks (in a logical block) that are fully occupied with valid/invalid pages, and compact them in advance during the long idle periods. Compacting blocks involves enforcing all valid pages

from the fully-occupied physical block to a new, clean block, and removing the invalid pages in the former by erasing them. Consider as an example Figure 6 where we have two fully-occupied blocks, namely, LBN 5 and LBN 32768. AGC can compact these two blocks in advance during long idle periods. In order to avoid the scanning penalty required to identify the fully-occupied blocks, we add the LBN of the fully-occupied block to the AGC job list, while the FTL is serving the I/O requests so that the blocks can be compacted *proactively* without scanning the entire storage address space.

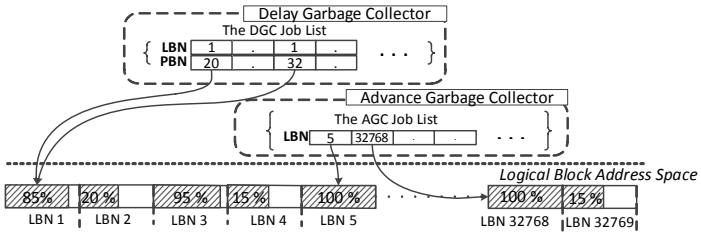


Fig. 6: Job lists for AGC and DGC.

Even though proactive block compaction is relatively simple, it can be very effective in practice as far as enhancing idle time utilization and securing free blocks are concerned. It should be noted that the proactive block compaction mechanism is executed *only if* the number of free blocks is less than the free block threshold (i.e., an on-demand GC would be invoked very soon). Therefore, similar to the look-ahead GC mechanism discussed earlier, this proactive block compaction mechanism also minimizes the number of unnecessary erase operations, which in turn helps to improve SSD endurance and reliability.

Incremental Garbage Collection One concern regarding AGC is that it could lead to undesired performance degradations and prevent the GC latencies from being hidden, if idle periods are too short or do not occur frequently enough. To avoid this, our implementation of AGC splits GC activities into smaller ones delimited by *checkpoints*, and performs the GCs step-by-step based on the checkpoints. As illustrated in Figure 7, the checkpoints are inserted at the end of every NAND I/O completion point and constitute the boundaries across the neighboring GC steps. Inspired by the checkpointing strategy described by [24], AGC incrementally performs a given GC operation one step at a time; this is referred to as the *Incremental Garbage Collection* (*Incremental GC*) in the remainder of this paper.

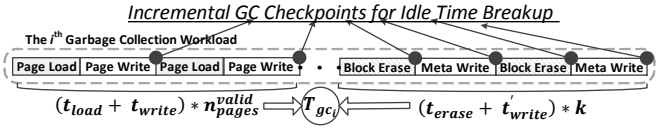


Fig. 7: Checkpointing for incremental GC. At each checkpoint, by checking the device-level queue, the garbage collector can decide whether it can perform further collections or not.

Whenever AGC reaches a checkpoint, the incremental GC determines whether further collections can be performed or not by checking the device-level queue. If there are no I/O requests until the next checkpoint, it goes ahead and executes the next step of the GC operation. The same procedure is repeated as long as there are no I/O requests. If on the other hand AGC detects an I/O request at a particular checkpoint, it postpones the remaining GC steps to the next idle period. To do this, it marks the current GC job status and inserts this marked status information into another job list that is managed by DGC (this will be revisited in Section 5.2). This incremental GC operation allows AGC to avoid the potential drawbacks of very short idle periods, and smoothly pass the control of GC operations to DGC. As a result, the SSD is able to serve the bursty I/O requests that can potentially create very short idle periods.

5.2 Details of Delayed GC Strategy (DGC)

Even though idle periods are typically long enough [20] [22] [25] for AGC to prepare available free blocks ahead of time and execute GC in advance, in cases where idleness does not occur frequently, AGC may not be very successful. The main goal behind our *Delayed Garbage Collection Strategy* (DGC) is to address this situation by delaying GC invocations. Its operation can be divided into two steps as explained below.

Update Block Replacement As stated earlier, the main reason why GCs degrade system performance is *page migrations*. To avoid this degradation, DGC defers the page migration activity to *future* idle periods. Whenever an on-demand GC occurs in a busy period, DGC allocates free block(s) as update block(s). Normally, commodity FTLs migrate valid pages from the update and data blocks to an allocated free block. In contrast, DGC skips this process and serves the urgent I/O requests. Rather than migrating pages, DGC adds the LBN and PBN(s) corresponding to the migration into a job list it maintains (called the *DGC job list*). This delayed page migration activity is later resumed in a future idle period by the DGC's retroactive block compaction (see Section 5.2).

The free block allocation carried out by DGC is similar to what a standard FTL would do during GC. The only difference is that DGC allocates the block as an update block (not a free block). Since the FTL already has an update block (but it is garbage), DGC intercepts the update block information and replaces the PBN of the update block with the allocated free block's PBN in the FTL mapping table. In this way, the FTL treats the allocated free block (called the delay block) as an update block, and is not required to manage the block mapping information. It explicitly manages replacing/updating a block for preserving consistency during information mapping. Further, DGC maintains this information using the DGC job list and hides this information from the FTL until the page migration process completes. In the meantime, if there is an I/O request, the FTL serves that request based on the available mapping information. This replacement and interception procedure is called the *Update Block Replacement*.

The main advantage of the update block replacement is that, as soon as the SSD receives an I/O request, it can serve the request without migrating the valid pages, even when AGC could not handle on-demand GCs in advance. Another benefit is that DGC does not require any additional NV memory space for delaying on-demand GCs, which is essential resources of prior works [11] [13]. This is because it replaces the update block with free blocks that belong to the

FTL address space. Note that the mappings employed by the FTL and DGC do *not* interfere with each other, and this allows DGC to work with various other mapping schemes used in current FTLs.

Retroactive Block Compaction When there is no I/O congestion, DGC performs page migrations and returns the relevant delay block and update/replace block to the free block space. The blocks returned DGC can be recycled as normal free blocks. To return a block, DGC first extracts the LBN and PBN for a replace/update block from the DGC job list. It then queries the PBN for the data and delay blocks by using FTL's block-level mapping table. That is, it looks up the mapping table entry for the LBN extracted from the DGC job list and gets the corresponding PBN from the table.

Once DGC collects all PBN(s) for the blocks related to the delayed logical block, it retroactively compacts the blocks and returns them to the original state (i.e., as free blocks). While compacting, DGC migrates valid pages deferred from all PBNs for each delay, replace/update, and data block. This page migration is simply executed by reading and writing pages in an ascending order. We want to point out that the number of pages requiring migration is less than or equal to the number of pages in a logical block, independent of the number of delay and data blocks involved. Thus, the migration cost of DGC is the same as that of original GC. During busy periods, DGC preferentially reads and writes pages to the delay block rather than the replace/update block to guarantee data consistency. The reason behind this order is that the delay block contains the latest data when compared to the data in the replace/update block(s). This also helps DGC to improve block utilization and reduce the amount of I/O activity while performing the collections since the replace/update block(s) can be erased without any read or write operation in the ideal case.

```
if irp.command != empty then
    if ftl.checkOnDemandGc(irp)
    then
        /* delay the on-demand GC   */
        UpdateBlockReplacement(irp)
        insertEntry(DgcJobList,
        irp.getLbn())
    /* call the FTL service         */
    ftl.ServeIo(irp.command, irp.lsn,
    irp.sectors)
else
    targetLbn := getDgcLbn(DgcJobList)
       /* DGC                       */
    if targetLbn != nullblock then
        consumed = RetroactiveBlock-
        Compaction(targetLbn)
    /* AGC                          */
    idleType :=
    checkIdleType(CommandQueue,
    consumed)
    if idleType = short then
        /* Calculate GC latency using
           Equation 1 & 2           */
        idletimes :=
        getIdleTime(CommandQueue,
        consumed) requiredTimes :=
        speculateExecutionTime()
        while idletimes ≥
        requiredTimes do
            LookaheadGc(irp)
    else if idleType = long then
        ProactiveBlockCompaction(irp)
```

Algorithm 1: IssueCommands(IoRequestPacket irp) of our proposed AGC+DGC algorithm. Note that the SSD just forwards I/O requests to the underlying FTL without performing any GC during the busy periods.

5.3 Putting the Two Schemes Together

When our two schemes, AGC and DGC, are applied together we expect that most GCs are invoked by AGC; DGC will be invoked only if the idleness at hand is insufficient or the number of free blocks secured by AGC is not enough. In fact, we observed during our experiments that the fraction of idle periods DGC handles accounts for at most 20%, and AGC manages the rest. Algorithm 1 describes the

steps involved in integrating DGC and AGC (called the *AGC+DGC scheme*). In summary, if an I/O request triggers an on-demand GC, DGC delays page migration to future idle periods using the *update block replacement* mechanism. During idle periods, DGC first performs *retroactive block compaction* **only if** a delayed GC block exists. And, AGC is invoked based on the type of idle period at hand. Specifically, if the idle period is short (just enough to perform the required GC), *look-ahead garbage collection* is invoked. Finally, *proactive block compaction* is invoked when the idle period is long. In each implementation, GC is performed incrementally, as explained in Section 5.1.

6 Experimental Evaluation

To evaluate the effectiveness of our AGC and DGC, we introduced them in a event simulation platform whose a typical SSD storage stack is fully implemented, including flash drivers, translation layers, and host interface controllers. Our simulator also models multiple channels and ways with a bus transaction-level clock accuracy such that different types of idleness can be accurately simulated with diverse workloads we tested.

SSD configuration. We implemented two different SSD-based disk arrays;

• **6SSDs-RAID**: the first disk array was setup based on the original MSN file server storage configuration [22], which consists of 6 disks (Disk0 $\sim$ Disk5). In this default array, we introduced six of 64GB SSDs and each SSD, which replaces each disk of MSN storage server, has 4 channels and 4 ways architecture. Further, we categorize this SSD array based on each disk of write-intensity.

- *6SSDs-RAID-LO* is the group of SSD0, 1, 2, and 3 with low I/O intensive workloads of which the fraction of write amount is under 20% of total I/Os.
- *6SSDs-RAID-HI* is another SSD group, consisting SSD4 and 5 with high I/O intensive workloads of which the write fraction of total I/Os is 80%.

• **3SSDs-RAID**: Another disk array leverages three SSDs, in which each individual SSD composes of 8 channels and 8 ways (128GB). This disk array was configured to measure performance impacts on a different SSD configuration. In this 3SSDs-RAID, disk0 and disk1 (of the MSN server) are replaced by SSD0, disk2 and disk5 are replaced by SSD1, and disk3 and disk4 are replaced by SSD2.

Both SSD arrays in RAID-0 configuration are viewed by the OS as a single device. Even though we model a Samsung K9KGA0B0M MLC NAND flash[2] [26] in our simulations, our proposed GC strategies can be applied to other NAND flash device models as well. Due to limitations of space, we are not able to show our evaluation on other devices, but the performance behaviors with the most current version of NAND flash packages (two planes and dual dies architecture) are very similar to the results shown in this section.

FTL implementation. We implemented a log-structured FTL (*L-FTL*) and a 2:8 hybrid mapped FTL (*H-FTL*) on the SSD-based disk array models [12] [8]. We also implemented a partial GC scheme based FTL (*P-FTL*) [11] [13] [10]. After some initial experiments, the percentage of free blocks and GC threshold are set to 3% and 1%, respectively, of the total SSD address space[3]. We also introduced

[2] This has 128 pages per a block. Based on a 4 KB page size, read, write and block erase latencies are 183.2 us, 860.36 us, and 2 ms, respectively.

[3] Some industries employ even higher GC thresholds with more free blocks, which renders SSDs expensive. Since there is a variety of configurations for GC threshold,

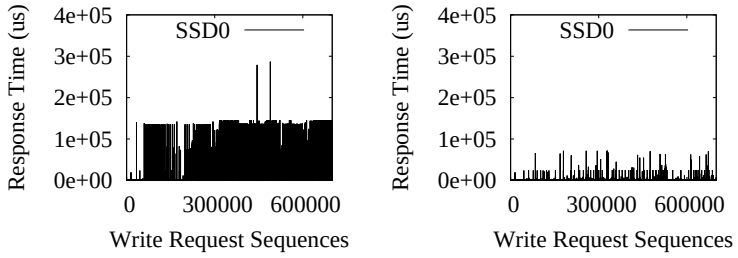
	The number of I/O requests	Total amount of requests (KB)	Total amount of writes (KB)	Idle Periods (%)
Disk0	1,509,397	32,490,240	3,051,918	38.6
Disk1	2,221,728	35,383,340	17,722,159	81.3
Disk2	500	1,958	1,958	56.6
Disk3	4,352	2,392,445	2,387,767	83.0
Disk4	12,627,396	117,607,983	24,835,283	42.4
Disk5	12,981,710	130,033,924	31,777,436	41.3
Total	29,345,083	317,909,889	79,776,520	64.1

Table 1: Important characteristics of our traces. The last column gives % of I/O requests containing sufficiently long idle (> 1 sec) periods.

a 14 GB extra space to P-FTL for each SSD in the 6SSDs-RAID and 28GB extra spaces to it for each SSD in the 3SSDs-RAID based on the results from the write buffer analysis [11]; these extra spaces are used as the non-volatile write buffer in an attempt to serve urgent I/Os and provide real-time support, and managed through the page-level mapping scheme in P-FTL, instead of employing a block-level mapping scheme.

Workloads. Enterprise traces tested are collected from the MSN file storage server over 5 days [22] [23]. The total I/O traffic studied was up to 1.8TB. Important characteristics of our traces are given in Table 1. In the traces used, 34.6% of idle intervals were long (larger than 1 sec) and less than 29.5% were short, and 35.9% of the requests were back-to-back with no idle time in between.

It should be mentioned that each I/O request of any trace we simulate has a time stamp associated with it, and all the different approaches we tested (for reducing GC overheads) take advantage of scheduling the I/O requests based on the corresponding time stamps (using NCQ). Our bus-transaction level simulator extracts access time information from the I/O commands, using which we synchronize the global timer of the simulator and check the I/O latencies at the end of every I/O completion. This enables us to accurately record idle/busy periods on the SSDs.



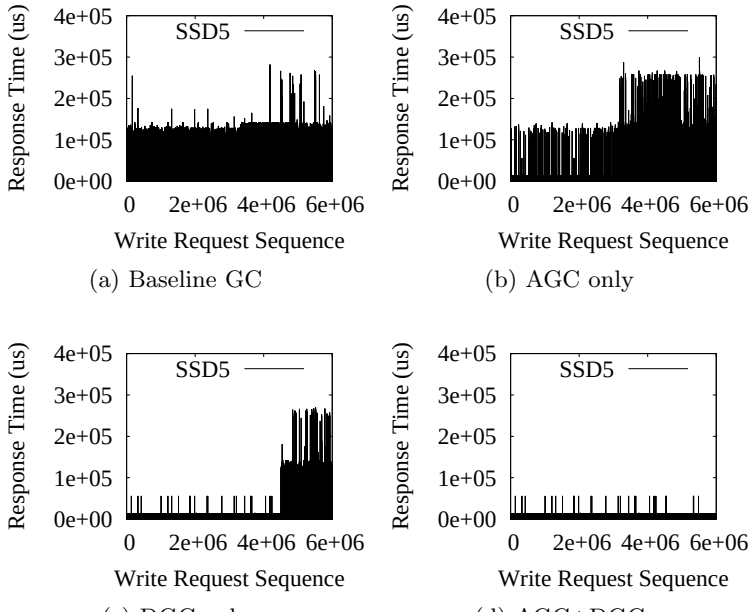
(a) Baseline GC (b) AGC only

Fig. 8: Performance of AGC with relatively low I/O intensive workloads (SSD 0 of 6SSDs-RAID).

we choose a lower bound value for our evaluation. We believe that alleviating GC overheads in our configuration (more complex) can be reduced the GC problem in such expensive SSDs configuration based on write amplification analysis with over-provisioning and spare factors [27].

6.1 Performance Comparison

We first evaluate the performance of our two GC strategies (AGC and DGC) using 6SSDs-RAID in isolation. Figures 8 and 9 plot the response times of SSD0 of 6SSDs-RAID-LO and SSD5 of 6SSDs-RAID-HI, respectively. As illustrated in Figure 8, AGC alone successfully hides almost all on-demand GCs in SSD0, leaving nothing for DGC. We see from Figure 9b, however, that AGC alone is not very successful with the high I/O intensive workloads. During the high write-intensive periods, a few on-demand GCs are invoked due to the very small amount of short idle periods in SSD5. Even though the number of these on-demand GC invocations is small, the FTL uses up available free blocks for new requests, which introduces more on-demand GC invocation. In a worst-case scenario, AGC suffers from both increased amount of GC invocations and short idle periods as the execution progresses. This is the reason why AGC requires DGC to handle such on-demand GCs. One can see from Figure 9c that DGC alone successfully hides the GC latencies until four million write requests are served. However, as soon as the available free blocks run out, DGC starts performing on-demand GCs. One can also see from this result that DGC needs AGC, which supplies free blocks, enabling the former to defer on-demand GCs. Both AGC and DGC, when applied individually, increase the number of GCs compared to the baseline GC, which is used to perform on-demand GC of L-FTL (see Figure 9a). However, when they are applied together, they successfully hide GC latencies, as illustrated in Figure 9d, and the total number of GCs does *not* exceed the baseline case (Section 6.5). In the rest of our experiments, we focus on this integrated AGC+DGC scheme.



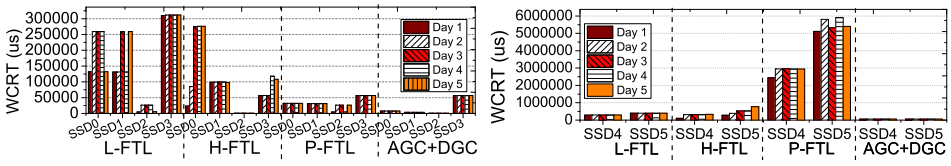
(a) Baseline GC (b) AGC only

(c) DGC only (d) AGC+DGC

Fig. 9: Performance comparison of different garbage collection strategies (SSD5 of 6SSDs-RAID with high I/O intensive workloads).

6.2 Worst Case Response Time

Figure 10 plot the worst-case response times (WCRTs) 6SSDs-RAID. We see from these graphs that, WCRT ranges from 131 *ms* to 311 *ms* in 6SSDs-RAID-LO, under both the L-FTL and H-FTL schemes. However, in both P-FTL and AGC+DGC we observe negligible WCRTs, which results in completely hiding the GC latencies from the I/O operations. We further observe that AGC+DGC reduces the WCRT by 65.2%, 98.6% and 96.4%, on average, over P-FTL, L-FTL and H-FTL, respectively. This is because AGC+DGC performs on-demand GCs only during the idle periods, and consequently, users experience no GC overheads during their I/O services.

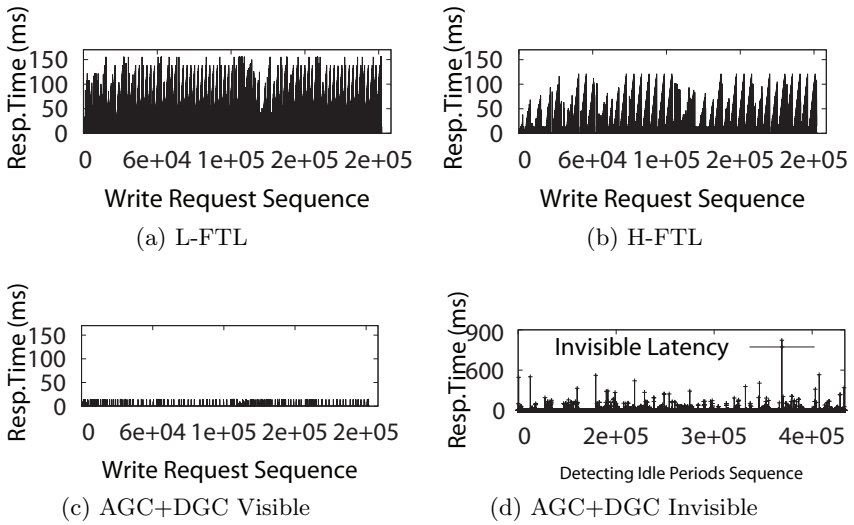


(a) 6SSDs-RAID-LO (b) 6SSDs-RAID-HI

Fig. 10: Worst-case response time (WCRT) analysis for 6SSDs-RAID. (a) With low I/O intensive workloads, P-FTL and AGC+DGC show deterministic behaviors while the performances of L-FTL and H-FTL fluctuate over time. (b) With high I/O intensive workloads, P-FTL experiences very high WCRT, whereas AGC+DGC continues to provide stable I/O performance.

However, in 6SSDs-RAID-HI, P-FTL's WCRT behavior fluctuates due to the *write buffer block thrashing* problem.[4] This causes P-FTL to perform out of order writes for a while and, as a result, WCRTs become ten times worse as compared to the L-FTL case. In contrast, AGC+DGC still serves I/O requests within the predefined latencies, and achieves about 53 *ms* latency, including the theoretic minimum for I/O processing, while the other approaches suffer from the performance fluctuations and experience long WCRT under heavy I/O requests. Further, SSDs supported by our AGC+DGC do *not* incur any GC latencies during busy periods, even in execution phases with very low idle times ($\leq 10\%$). This is because AGC eliminates on-demand GCs using idle times, and DGC postpones the GC latencies by shifting them to future idle periods, as plotted in Figure 11c. Figure 11 also explains how our proposed GC strategies collectively take GC overheads off the critical path. While L-FTL and H-FTL (see Figures 11a and 11b) incur GC latencies during the busy periods, AGC+DGC incurs (see Figure 11d) GC latencies *only* during the idle periods, which are not perceived by applications. This clearly shows that AGC+DGC provides stable and better SSD performance with no on-demand GCs taking place during the busy periods.

[4] This problem arises when the free pages in the write buffer (NV buffer) to which P-FTL writes urgent data are no longer available.



(a) L-FTL (b) H-FTL

(c) AGC+DGC Visible (d) AGC+DGC Invisible

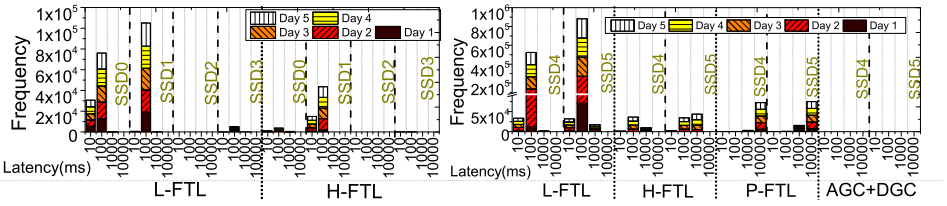
Fig. 11: Response times for a write intensive section (where the fraction of I/O executions with no idle time is account for about 90%). While H-FTL removes about 40% of the GC related overheads, AGC+DGC hides all on-demand GC latencies.

6.3 Excess Waiting Time

Figure 12 plots the amount of excess waiting time (EWT)[5] in 6SSDs-RAID. One can observe from Figure 12a that H-FTL significantly cuts down the GCs by maximizing the block-level locality. It also dramatically reduces the number of page migrations introduced by GCs. However, it can also be seen that, as the execution progresses (from day 1 to day 5), occurrences of EWTs increase, due to the shortage of available free blocks. To secure free blocks, H-FTL had to merge up to ten blocks into two logical blocks, and merge approximately fifteen thousand times a day, generating significant overheads. In contrast, P-FTL and AGC+DGC successfully hide GC overheads at runtime (and thereby All EWT of them is zero). However, the frequency of EWTs in H-FTL is less than that in P-FTL with high write intensive workloads (see Figure 12b). In this case, P-FTL could not fully hide GC latencies when the NV buffer was completely used by the large amount of I/O requests. This is because P-FTL incurs much longer latencies than H-FTL, due to the write buffer thrashing problem, which is the same as the one causing high WCRT.

On the other hand, our scheme successfully hides GC latencies because AGC can ahead secure available blocks (delay blocks) to DGC even under high write intensive workloads. Further, because of update block replacement scheme, the delay blocks are the same as the free blocks, thereby not requiring any extra blocks to manage different mapping schemes. Our proposed strategy essentially eliminates on-demand GCs by exploiting different types of idle periods and thus leads to stable GC latencies.

[5] EWT is defined as the difference between the actual wait time and the marginal response time (in this paper, it is assumed to be 30 ms).



(a) 6SSDs-RAID-LO (b) 6SSDs-RAID-HI

Fig. 12: Excess waiting time (EWT). The x-axis represents the upper bound on EWT. (a) L-FTL and H-FTL experience I/O blocking problem stemming from GCs while P-FTL and AGC+DGC have no such problem. (b) With heavy writes, even though P-FTL results in fewer GC invocations, its GC latencies are much longer than others

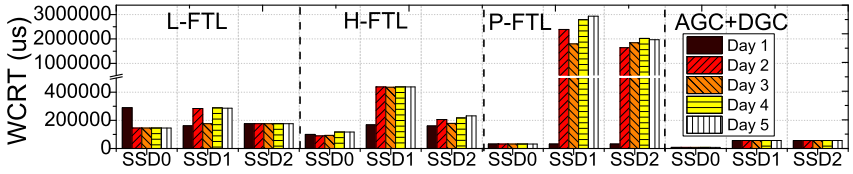


Fig. 13: Worst-case response time (WCRT) analysis for the 3SSDs-RAID.

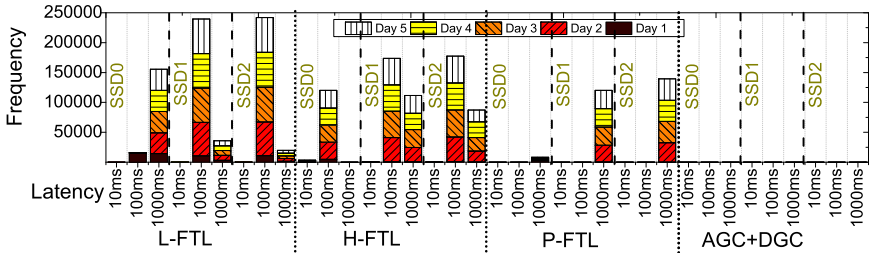


Fig. 14: Excess waiting time (EWT) analysis for the 3SSDs-RAID.

6.4 Performance Compariosn of 3SSDs-RAID

Figure 13 and Figure 14 illustrate, respectively, WCRTs and EWTs for 3SSDs-RAID. In both WCRT and EWT analyses, performance of the 3SSDs-RAID is similar to 6SSDs-RAID except for P-FTL. Specifically, in day 1, P-FTL guarantees deterministic performance with the zero EWT value on even high write-intensive workloads (SSD1 and SSD2 of 3SSDs-RAID) because each SSD of 3SSDs-RAID has a larger storage capacity than an SSD in the 6SSDs-RAID configuration. In other words, P-FTL is more tolerant to update block reclaiming GC overheads as its NV buffer has more physical pages. However, as the amount of writes increases, the available physical pages also run out. As a result, P-FTL could not satisfy the deadline requirements again. 3SSDs-RAID with P-FTL has about 50% less impact on the write block thrashing problem compared to 6SSDs-RAID, mainly because, in addition to the larger physical pages on the NV buffer, P-FTL itself can secure abundant free block resource as well, thereby reducing potential GC overheads during free block reclaiming. However, due to reasons similar to the case of 6SSDs-RAID, over the time, P-

FTL makes 3SSDs-RAID performance worse than L-FTL and H-FTL. While the performance of P-FTL depends mainly on the size of NV buffer and are not able to essentially take GC overheads off the critical path of SSDs, AGC+DGC satisfies the performance requirements irrespective of different SSD configuration chosen and the I/O traffics tested.

6.5 Side-Effects of AGC and DGC

Figure 15a plots the breakdown of GCs across different collection schemes. Since AGC is responsible for preparing the free blocks, it is desired that the contribution of the AGC be larger than that of the DGC. We see that, as expected, AGC executes for at least 80% of the total number of GCs. As a result, DGC is able to secure enough free blocks when it performs update block replacement to delay GCs. We want to point out that the proactive block compaction is applied in a majority of the AGC operations. The proactive block compaction does not execute until the number of free blocks is less than the free block threshold (even though it is under the underlying FTL's GC threshold (3%)). Therefore, our scheme does not introduce any unnecessary erases, and thus reduces the potential side-effects of GC.

Figure 15b presents the average block erase counts under different free block thresholds when executing AGC. In this figure, the dotted vertical line indicates L-FTL's average erase count per block, which is twenty one. Since AGC is performed only if the target GC block is fully occupied or if an on-demand GC is to be invoked very soon, it only migrates necessary GC activities from busy period, thereby minimizing side effect in terms of SSD reliability.

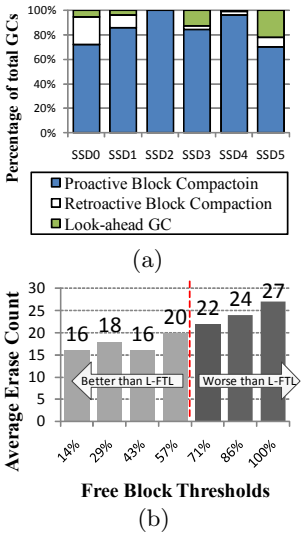


Fig. 15: (a) Garbage collection type breakdown of total collection. (b) Block erase impact by free block threshold.

We observed that the free block threshold should be less than 71% for the average erase count of the proactive block compaction in AGC to be comparable to L-FTL. If the proactive block compaction shifts on-demand GCs beyond this threshold, it makes wear-leveling characteristics worse than L-FTL. Interestingly, the erase counts with low free block thresholds are better than L-FTL. This is because preparing free blocks using fully-occupied blocks in advance helps to prevent the log block thrashing problem (in L-FTL), which can introduce improper erase operations. In our experiments, the best free block threshold for satisfying the wear-leveling requirement was found to be less than 43% of the original GC threshold.

7 Conclusions

We proposed novel a garbage collection strategy consisting of two main components, called *Advanced Garbage Collection* (AGC) and *Delayed Garbage Collection* (DGC), that cooperate in hiding GC overheads in SSDs. AGC tries to secure free blocks in advance and remove on-demand GCs from the critical path so that users do *not* experience GC latencies during I/O congestion. In comparison, DGC

handles GC invocations that could not be handled by AGC by differing them to future idle periods. Our experimental analysis using both enterprise workloads and high performance I/O workloads indicate that the proposed strategies (AGC and DGC) provide stable I/O performance. Compared to three state-of-the-art GC strategies, P-FTL, L-FTL and H-FTL, our integrated scheme (AGC+DGC) reduces GC overheads dramatically.

Acknowledgment. We would like to thank anonymous reviewers for their constructive feedback. This work is supported in part by NSF grants 1017882, 0937949, and 0833126 and DOE grant DESC0002156.

References

1. Caulfield, A. M. et al.: Understanding the impact of emerging non-volatile memories on high-performance, IO-intensive computing. In: SC. (2010)
2. Kgil, T., Roberts, D., Mudge, T.: Improving NAND flash based disk caches. In: ISCA. (2008)
3. Caulfield, A.M., Grupp, L.M., Swanson, S.: Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications. In: ASPLOS. (2009)
4. Lee, S.W., et al.: A case for flash memory SSD in enterprise database applications. In: FAST. (2011)
5. EMC: Raw drive capacity cost trends http://wikibon.org/w/images/a/a4/emcrawdrivecapacitycosttrends.jpg
6. Micheloni, Rino et al.: Inside NAND Flash Memories. Springer (2010)
7. Caulfield, A. M. et al.: Characterizing flash memory: Anomalies, observations,and applications. In: SC. (2009)
8. Kang, J.U., et al.: A superblock-based flash translation layer for NAND flash memory. In: EMSOFT. (2006)
9. Lee, Sang Won et al.: FAST: An efficient flash translation layer for flash memory. In: EUC Workshops. (2006)
10. Chang, L.P., Kuo, T.W.: Real-time garbage collection for flash-memory storage systems of real-time embedded systems. TECS (2004)
11. Choudhuri, S., Givargis, T.: Deterministic service guarantees for NAND flash using partial block cleaning. In: CODESS+ISSS. (2008)
12. Kim, J., et al.: A space-efficient flash translation layer for Compact Flash systems. In: TCE. (2002)
13. Jung, M., Yoo, J.: Scheduling garbage collection opportunistically to reduce worst-case I/O performance in SSDs. In: IWSSPS. (2009)
14. ULINK technology: http://www.ulinktech.com/
15. Intel: http://www.iometer.org/
16. Intel, Seagate: Serial ATA Native Command Queuing: An Exciting New Performance Feature for Serial ATA. Intel and Seagate, July (2003)
17. LeCroy: http://www.lecroy.com/
18. T10: http://www.t10.org/, Technical Committee T10 (2009)
19. Colarelli, D., Grunwald, D.: Massive arrays of idle disks for storage archives. In: SC. (2002)
20. Mi, N., et al.: Efficient management of idleness in storage systems. In: the ACM Transactions on Storage Journal. (2009)
21. Golding, R., et al.: Idleness is not sloth. In: USENIX ATC. (1995)
22. Narayanan, D., et al.: Migrating server storage to SSDs: Analysis of tradeoffs. In: EuroSys. (2009)
23. SNIA: http://iotta.snia.org/, IOTTA Repository (2006)
24. Kim, J.H., et al.: Incremental Merge Methods and Memory Systems Using the Same. U.S. Patent #2006004971A1, Jan. 5 (2006)
25. Narayanan, Dushyanth et al.: Everest: Scaling down peak loads through I/O offloading. In: EuroSys. (2008)
26. Samsung Electorincs: K9GAG0B0M. In: Data Sheet. (2008)
27. Hu, X.Y., et al.: Write amplification analysis in flash-based solid state drives. In: SYSTOR. (2009)