Pattern-based Write Scheduling and Read Balance-oriented Wear-leveling for Solid State Drivers

Jun Li*, Xiaofei Xu*, Xiaoning Peng†, Jianwei Liao*†
*College of Computer and Information Science, Southwest University, Chongqing, China, 400715
†College of Computer Science and Engineering, Huaihua University, Huaihua, Hunan, China, 418000
Corresponding author: Jianwei Liao, Email: *liaotoad@gmail.com*

***Abstract*—This paper proposes a pattern-based I/O scheduling mechanism, which identifies frequently written data with patterns and dispatches them to the same SSD blocks having a small erase count. The data on the same block are mostly like to be invalided together, so that the overhead of garbage collection can be greatly reduced. Moreover, a read balance-oriented wear-leveling scheme is introduced to extend the lifetime of SSDs. Specifically, it distributes the hot read data in the blocks with a small erase count, to heavily erased blocks in different chips of the same SSD channel, while carrying out wear-leveling. As a result, internal parallelism at the chip level of SSD can be fully exploited for achieving better read data throughput. We conduct a series of simulation tests with a number of disk traces of real-world applications under the *SSDsim* platform. The experimental results show that the newly proposed mechanism can reduce garbage collection overhead by `11.3%`, and the read response time by `12.8%` in average, comparing to existing approaches of scheduling and wear-leveling for SSDs.**

***Index Terms*—Solid state drivers, pattern, scheduling, wear-leveling**

I. Introduction

NAND flash memory-based solid-state drives (SSDs) have non-volatile nature, and are then widely leveraged in digital devices. Specially, SSDs are featured with small size, high performance, random-access performance and low energy consumption [1], [2], [3]. However, flash units (e.g. blocks of SSDs) are written upon, they must be erased before they can be written again. Generally, the basic unit of NAND flash memory has an erase limit, and the unit will become unreliable once its erase count reaches the limit. For example, the erase limit of single-level cell (*SLC*) is about `10` thousands, and the limit of multi-level cell (*MLC*) and triple-level cell (*TLC*) is approximately `1/10` and `1/100` of *SLC*, though the latter two can hold more data per cell, in contrast to *SLC* [4].

Consequently, a part of SSD blocks might wear out before the SSD device wears out, that lead to smaller available capacity of a SSD device at the late stage of the device's lifetime [5], [6]. Considering this fact, the wear-leveling algorithm is proposed, which works at Flash Translation Layer (FTL) of SSDs, to guarantee that all blocks within the SSD device can be uniformly worn out. In other words, through evenly spreading the number of P/E cycles (i.e. the erase limits) that is taken place across different SSD blocks, the SSD device can decrease the aging heterogeneity across all blocks, to further extend the lifespan of the device [6].

Specifically, the wear-leveling mechanism intends to minimize the variance of the wearout amount across blocks through remapping the heavily written blocks to less frequently written ones [5]. With respect to different application contexts, many wear-leveling algorithms have been developed [5], [7], [8], [9]. For example, Liao et al. [5] have introduced an adaptive wear-leveling approach, to allow varied wear-leveling policies at different stages of device lifespan. Chang et al. [7] proposed a scheme on the basis of hot/cold data swapping, to better enhance wear-leveling effectiveness after categorizing data blocks into several sets.

On the other side, the data stored in SSD blocks can be divided into three categories: hot read data, cold read data, hot write data, by referring their access characteristics. Although the existing wear-leveling methods differ from each other in technical details, they commonly migrate a block of (valid) data to another free SSD block in the same SSD plane. Thus, the erase counts across all blocks can be guaranteed. In other words, the factor of hot read data and cold read data within the same SSD block is not taken into account. But migrating hot read data to different chips of SSDs can obviously benefit to exploiting the chip-level internal parallelism for real-time responses [12].

To address this issue, we propose a read balance-oriented wear-leveling mechanism. This method groups both hot read data and cold data in the migrant blocks (on which the valid data are required to be migrated) as a units, and then move the units to the available blocks in different SSD chips by following a round-robin fashion.

For facilitating the proposed wear-leveling scheme, we have also introduced a pattern-based scheduling approach at Flash Translation Layer of SSD, to dispatch write requests. This approach groups the frequently write data with pattern itemsets and maps them to the same SSD blocks, when these data are required to be flushed to the disk. Consequently, it is able to boost the effectiveness of wear-leveling, as the read data and the frequent write data can be dispatched onto the different SSD blocks. Moreover, since the relevant hot write data, that are combined and saved on the same block, are likely to be invalidated together, it can consequently decrease the garbage

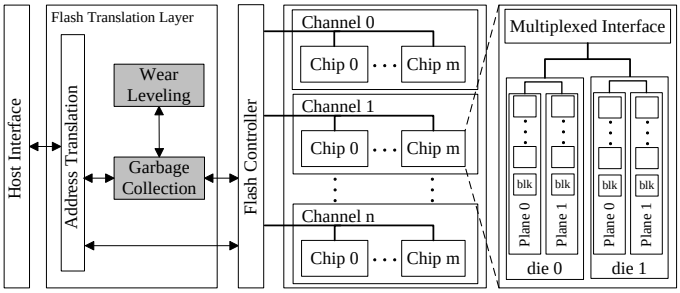


Fig. 1. The architecture overview of parallelism in SSDs (including channel-level, chip-level, die-level, and plane-level parallelism [2], [17]).

collection overhead.

In brief, this paper makes the following two contributions:

- *Combining hot read data and cold read data in migrant units.* In the wear-leveling process, the hot read data and the cold read data of migrant blocks are combined with units. Then, these units are migrated to the free blocks of different chips in the same SSD channel, according to the round-robin fashion. The primary motivation is to maximize read data throughput by exploiting chip-level internal parallelism, through carrying out wear-leveling.
- *Group dispatching write requests with frequent patterns.* The frequent patterns of write requests have been mined, to direct mapping logical addresses of requests at FTL. Specifically, the write requests in the pattern are supposed to be flushed to the same blocks of storage device. The purpose is to separate the hot write data from other data, for benefiting the newly proposed wear-leveling scheme. Furthermore, it can minimize the write amplification in garbage collection, to reduce its overhead. This is because the relevant hot write data in the same pattern are more likely to be invalidated together.

The rest of paper is organized as follows. Section II describes an overview of NAND Flash architecture, related work and our study motivation. Section III presents the design and implementation specifications of the newly proposed mechanism. Section IV shows evaluation experiments and relevant discussions. Finally, we conclude the paper in Section V.

II. Background and Motivation

A. SSD Architectural Overview

Figure 1 shows an architectural overview of SSDs, including the software and hardware. As seen, the main software layer is Flash Translation Layer (FTL), which takes charge of translating logical sector address into physical address that the flash memory can identify. Furthermore, it supports garbage collection and wear-leveling: garbage collection is carried out to reclaim the space occupied the invalid data (e.g. outdated pages) due to the out-place updates. Since each SSD block affords a limited number of erases, it becomes critical to take advantage of the mechanism of wear-leveling, to extend the lifespan of flash memory by uniformly distributing erases across all blocks [26].

On the other hand, for enhancing the I/O performance of the SSDs, the parallelism feature has been exploited in the internal structure of the SSDs. Hu et al. [17] categorized the internal parallelism of the SSDs into four levels: channel-level, die-level, chip-level and plane-level, as shown in Figure 1. Supposing there are two incoming requests in the queue, which can be concurrently dispatched to two different chips, even in the same channel, by resorting to the chip-level internal parallelism. This feature is specially beneficial for read requests, as the I/O response time can be significantly reduced. But for write requests, it may bring about some negative impacts on the efficiency of garbage collection [12].

B. Wear-leveling Approaches

As discussed, the primary goal of the wear-leveling algorithm is to prolong the lifespan of flash memory, by preventing any single block from prematurely reaching the erasure cycle limit. The (static) wear leveling algorithms try to migrate cold data to more worn blocks for yielding more even distribution of wear. But, moving cold data around without any update requests must result in certain overhead [22].

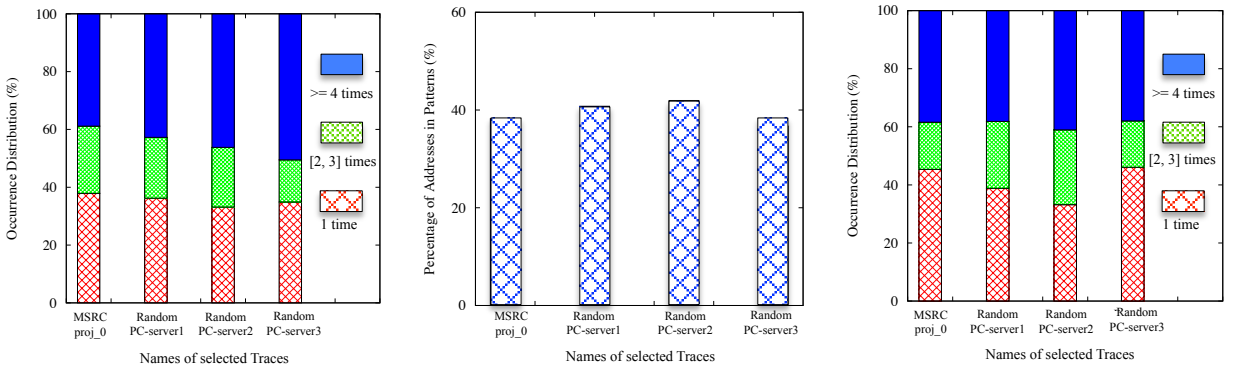
Chang et al. [11] proposed a static wear-leveling, which holds a Block Erase Table (*BET*) and two procedures of SWL-Procedure and SWL-BETUpdate. When flash memory needs conducting static wear-leveling, the SWL-Procedure function is invoked. It triggers garbage collection and selects adaptive block to write back data. The SWL-BETUpdate procedure updates erase count information in *BET* after erasing blocks. On the basis of this work, they also introduced a mechanism, for swapping hot data with cold data to improve performance of wear-leveling [7].

To reduce the cache space for holding *BET*, conventional mechanisms use a 1-bit flag as the sign of two or three blocks, but which degrades precision of wear leveling [10]. To overcome this problem, Kim et al. [10] presented a round robin-based wear-leveling algorithm called *RRWL*, which takes advantage of a one-to-one mode based on the round robin method, to increase accuracy of cold block identification.

Liao et al. [5] proposed an adaptive wear-leveling algorithm, to avoid unnecessary erasing operations. In early lifespan of Flash memory, the wear-leveling operations are purposely avoided. It triggers wear-leveling to migrate cold write data to the block, only if the distribution of block erasures becomes noticeably uneven, and is beyond a threshold.

Chang et al. [8] presented a lazy wear-leveling algorithm. It uses a RAM counter to indicate the average wear of the entire flash, and it automatically selects a threshold to make balance between wear evenness and overhead. Kwon and Chung [14] proposed a mechanism to boost the performance and durability of the flash memory. While executing the algorithm, the target data, which are likely to be updated, are supposed to be evenly distributed onto other blocks.

Yang et al. [9] made use of the hardware-dependent information to estimate the reliability level of each block, and then proposed an efficient reliability-aware wear-leveling scheme. By utilizing the reliability level of each block, the proposed



(a) Access distribution of logic sector addresses (b) Percentage of frequent access addresses in patterns (c) Read distribution of logic sector addresses

Fig. 2. Logical sector number occurrence analysis of collected traces. (a) Access (read and write) frequency distribution; (b) Ratio of frequent access addresses in patterns; (c) Read frequency distribution.

scheme can maximize the number of good blocks that have evenly low error rates.

C. Garbage Collection

The functionality of garbage collection is to reclaim the page contained invalid data, and migrating valid pages may result in extra write operations (i.e. write amplification [24], [27]). Regarding garbage collection, prior approaches [2], [12], [23], [24] mainly intend to shorten the interference between garbage collection and I/O transactions, for minimizing the garbage collection time. Specially, Huang et al. [13] have presented a write buffer algorithm, which exploits correlations of pages in the SSD cache to eject the buffered pages together to the blocks, for improving performance of garbage collection.

More importantly, Guo et al. [12] propose a garbage collection aware I/O Scheduler called *PGIS*. It depicts a hot data identification approach, and then groups the frequently updated data to the same SSD blocks, for eventually decreasing the garbage collection overhead. But, the clustered hot data on the SSD block may be lack of correlation, and they may not be invalidated together in the same time period.

Besides, to further cut down the overhead of data migration, many technical efforts attempt to conduct wear-leveling through garbage collection if needed [7], [8], [9], [11], [26]. In other words, the migrant blocks are selected according to the principle of wear-leveling, their valid pages in these blocks are supposed to be migrated to other free blocks having a relative high erase count, by performing garbage collection.

D. Trace Access Characteristics and Motivation

Inspired by [12], we have further analyzed the access frequency distribution of logical sector address in the collected disk traces. Figure 2 presents the analyzed results of *MSRC proj_0* and three collected *random PC server* traces[1]. We have found that a non-negligible portion of logical sector addresses have been read or written more than `4` times. For instance, around `40%` of logical sector addresses have been requested beyond `4` times in the *proj_0* trace, as reported in Figure 2(a).

Then, we have generated the frequent access patterns (i.e. sets) of logical sector numbers of requests in each time window having `1024` requests. The minimum support for an itemset to be identified as frequent is configured as `4`, and the element number of each pattern is set larger than `2`. More details about the mining approach for frequent access patterns of logical sector numbers are depicted in Section III-B1. After that, we count the occurrence distribution of addresses, which have been flushed not less than `4` times, are in these frequent patterns or not. As shown in Figure 2(b), the results demonstrate that a vast majority of frequently accessed addresses are requested with other related addresses in the same pattern. Take the *proj_0* trace for instance, `38.2%` of frequent requested logical sector numbers are associated with the mined frequent access patterns.

In brief, the most significant clue revealed by Figure 2(b) is about the frequently accessed logical section numbers tend to be requested with fixed patterns. Although the read patterns do not benefit request scheduling, the write patterns may contribute to system performance. If the addresses in the pattern are flushed to the same blocks of storage device, write amplification can be consequently eliminated or alleviated, because these logical sector addresses are more likely to be updated together. At the same time, other logical sector addresses are mapped to SSD blocks with the default scheme, their data will become either hot read data, or cold read data at the late stage.

Moreover, we have also disclosed the read frequency distribution of the logical sector addresses about the selected traces, and Figure 2(c) presents the results. Similarly, many of logical sector addresses have been read for multiple times, we can refer their data as hot read data. At the same time, a half of logical sector addresses have been requested for only once, and their relevant data can be regarded as cold read data. Obviously, accelerating the responses to the requests for the hot read data may contribute to better system performance.

Such observations motivate us to separately deal with write

[1]We target at the applications having both relative large numbers of read and write requests. Section IV will present the specifications on these selected traces.

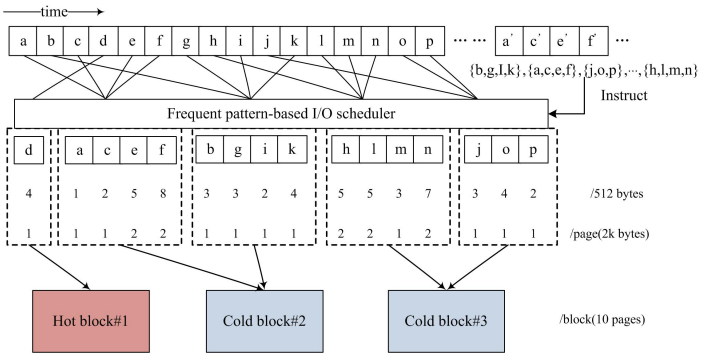


Fig. 3. Pattern-based mapping mechanism overview.

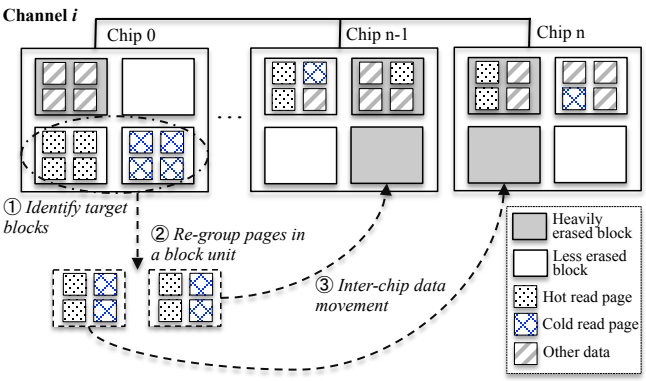


Fig. 4. High level overview of batched inter-chip wear-leveling.

and read requests, by taking the access characteristics into account. (1) we propose a pattern-based scheduling method for write requests, to effectively reduce the garbage collection overhead. (2) we re-distribute the frequently read data onto different places (e.g. SSD chips) while carrying out wear-leveling, so as to ultimately minimize the time required for accessing the hot data with a parallel manner.

III. Design and Implementation

A. System Overview

In this newly proposed mechanism, we first identify hot write requests with frequently access patterns, and then map these requests to the same SSD block. It is able to boost the garbage collection efficiency and benefit wear-leveling at the late stages. After that, we can re-distribute the frequently read data accompanying with the less accessed data onto the different chips (or dies, or planes), when conducting wear-leveling[2]. Consequently, the hot read data can be accessed concurrently for achieving better system performance, through taking advantage of the internal parallelism of SSD devices.

Note that only the frequent write data are supposed to be flushed to the SSD blocks with patterns, when scheduling the write requests. On the other side, the less modified write data will be dispatched by using the default algorithm. They will become cold read data or hot read data after running the application for a while. Such data will be re-organized and re-distributed during the process of wear-leveling.

B. Pattern-based Write Scheduling

In order to reduce the garbage collection overhead and maintain the block erase number, the scheduling mechanism can group the frequently accessed logical numbers of write requests in the I/O queue. Then, the data of these requests will be flushed to the SSD blocks having a small erasure count.

A high level overview of system architecture is shown in Figure 3. There are some requests in the I/O queue, labeled as logical sector numbers $(a, b, c, ...)$. By referring the frequent mined patterns, the requests hit in a specific pattern should be grouped and dynamically mapped together to a lightly erased block. On the other side, the rest of requests that are rarely updated, are dispatched to the block that has a relatively heavy erasure count. Specifically, Figure 3 assumes that each block has 10 pages and each page holds $2k$ bytes. The request hit in patterns, like $\{a, c, e, f\}$, should be grouped and mapped to cold blocks, which have a small erasure count (i.e. #2 in Figure 3). The requests that are not hit in patterns, like d, are dispatched to hot blocks having a large erasure count.

1) Mining Frequent Patterns: To support the proposed scheduling scheme, generating frequent patterns is the first step. The logical sector addresses of requests, such as lsn_1, lsn_2, lsn_3, ..., lsn_n, are needed to be mined. We take advantage of the *FP-Growth* algorithm for mining frequent item sets [21], to generate frequent patterns from write requests in the I/O queue. As a result, we can achieve a number of frequent item sets. For example, the pattern of $\{lsn_a, lsn_b, lsn_c\}$ has three logical sector numbers of requests, and these address combinations have occurred multiple times (i.e. more than predefined threshold).

2) Pattern Matching: In the scheduling process, we have to determine whether the current request is associated with a mined pattern, for mapping it to a block. To this end, we build a matrix to improve the effectiveness of matching process. The row of matrix is relevant to the requests in the I/O queue which labeled as logical sector numbers. And the column is the pattern which previously generated while the value of matrix is the number of accessed times about logical sector numbers.

C. Read Balance-oriented Wear-leveling

The read balance-oriented wear-leveling scheme can adaptively combine frequently read data with infrequently read data, and move them to a block having a relative large erasure count. The motivation is not only to balance erasure distribution of SSD blocks, but also to maximize read data throughout, by spreading the frequently read data onto different chips (dies or planes) of SSDs with a parallel manner.

Figure 4 illustrates a high-level overview of the proposed wear-leveling approach to re-group read data and map them

[2]It completes wear-leveling by garbage collection, in the meanwhile the SSD channel is not available [7]. Therefore, we only migrate the data within the channel while carrying out wear-leveling.

to the blocks on the different chips. As seen, it first identifies the target blocks, which hold the pages of data needing to be migrated. Next, it re-groups the frequent read data with the less read data. At last, the re-grouped data will be migrated onto the blocks of different SSD chips located at the same channel. Note that during the process of garbage collection, the target SSD channel is not available, so that we do not support inter-channel data movement in wear-leveling.

To better illustrate our wear-leveling mechanism, Algorithm 1 shows its pseudocodes with more details. First, we make use of *BET* to record whether the block has been erased or not (a bit for an entry). The parameter of $ecnt$ indicates the total erase number of all blocks. The parameter of $fcnt$ means the number of `1`s in *BET*, and $findex$ represents the block index in *BET*. The predefined threshold of T is utilized to trigger the process of wear-leveling, which is defined as $ecnt/fcnt$. We set it as `10` in our evaluation, by referring [7].

Furthermore, we propose *PRT* (Page Read Table) to log whether the page has been read or not (a bit for an entry). Similar to *BET*, we employ `1` to represent the corresponding page has been read; otherwise, the bit is set as *0*. Note that all relevant entries will be reset as *0*, in the case of the associated block is erased. That is to say, *PRT* can be used for directing the read balance-oriented wear-leveling. The parameter of *rfcnt* is the number of `1`s in *PRT*, which will be used for identifying the read type of a target block.

As seen in Lines `10-11` of Algorithm 1, we retrieve *BET* to find a target block that may have a small erase count, after triggering wear-leveling. On the one hand, either frequent read block or less read block is supposed to be split, and the split data will be distributed to the free blocks on the different SSD chips. Lines `14-34` demonstrate the specifications on processing this case. Specifically, we use a block set that has two blocks, to be the destinations for moving data in. And each block in the set is supposed to hold a half of frequent read data and a half of less read data. On the other hand, it carries out normal garbage collection for the target block having an index of *findex*, as seen in Line `36`, to fulfill wear-leveling.

Algorithm 1: Read balance-oriented wear-leveling

Input: $ecnt$, $fcnt$, $findex$, BET, T, PRT, $rfcnt$
Output: $NULL$

```
1  if fcnt == 0 then
2      return;
3  while ecnt/fcnt ≥ T do
4      if fcnt == size(BET) then
5          ecnt = 0;
6          fcnt = 0;
7          findex = 0;
8          return;
9      /*find the target block for moving its valid data out.*/
10     while BET[findex] == 1 do
11         findex = (findex + 1) % size(BET);
12     /* block(findex, rfcnt) returns the read type of
           block having an index of findex: 0 for less read, 1
           for frequent read, 2 for others.*/
13     type = block(findex, rfcnt);
14     if type == 0 or type == 1 then
15         /* blockset_head points an list. Its node holds
               the info about the blocks for moving data in.*/
16         blockset=blockset_head;
17         while blockset ≠ NULL do
18             /* find matched blocks for the current one*/
19             if type+blockset.type ≠ 1 then
20                 blockset=blockset → next;
21             else
22                 break;
23         /* the matched blocks are found?*/
24         if blockset ≠ NULL then
25             first=blockset.blocka;
26             second=blockset.blockb;
27             delete(blockset);
28         else
29             /*new a blockset node with two free blocks*/
30             p=new blockset(first, second);
31             p.type=type;
32             insert(blockset_head, p);
33         split block valid data moving to first, second;
34         Eraseblock(findex);
35     else
36         GCblock(findex);
```

IV. Experiments and Evaluation

This section depicts the experimental settings for evaluating the newly proposed mechanism, and then presents the experimental results. First, we describe the experimental setup, including the experimental platform and the used comparison counterparts. Next, the evaluation results and relevant discussions are reported, to show the feasibility and applicability of our newly proposed scheme. At last, we make a brief summary about the findings obtained from the evaluation experiments.

A. Experimental Setup

We conducted trace-driven simulation with *SSDsim (ver2.1)*, which has a wide range of modules and a diverse set of configurations [18]. Consequently, it has been employed in many studies for measuring the efficiency and performance of SSD systems. We applied our proposal as a part of *SSDsim*, for scheduling the logical addresses of requests to the physical page numbers in the SSD blocks, as well as carrying out wear-leveling. Since we aim to observe more erasures in garbage collection and wear-leveling, we emulated a small capacity of SSD device with `16 GB`, and Table I presents relevant parameters.

Four disk traces from real applications have been chosen in our tests. One is *MSRC proj_0*, which is in the Microsoft

TABLE I
EXPERIMENTAL SETTINGS OF *SSDsim* (MLC)

Parameters	Values	Parameters	Values
Channel Size	2	*Read latency*	0.03ms
Chip Size	4	*Write latency*	0.6ms
Plane Size	4	*Erase latency*	3ms
Block per plane	2048	*Erase limit*	3000
Page per block	64	*GC Threshold*	5%
Page Size	4KB	*FTL Scheme*	Page mapping

TABLE II
SPECIFICATIONS ON SELECTED DISK TRACES

	# of Req	Write ratio	Avg. Req size
MSRC proj_0	4224524	87.5%	39.0 KB
Random_server_1	5076765	68.8%	136.6 KB
Random_server_2	6658794	52.4%	123.6 KB
Random_server_3	11404881	30.7%	101.9 KB

Research Cambridge (*MSRC*) block I/O trace collection. Another three traces are collected from daily computer activities such as web surfing, email, multimedia and document editing. Table II reports the specifications on these traces.

Besides, we used the following comparison counterparts for measuring the performance of our proposed mechanism:

- *Baseline*: which indicates the dynamic mapping scheme adopted by *SSDsim* by default. And the functionality of wear-leveling is not supported in our evaluation.
- *SWL*: which means the approach of static wear-leveling [7]. It intends to proactively migrate cold data, to evenly distribute block erases over flash memory.
- *PGIS+SWL*: *PGIS* [12] is a garbage collection aware I/O scheduler that decreases garbage collection overhead by dispatching hot write data into the same physical block. Because *PGIS* does not support wear-leveling, we have applied static wear-leveling in the native *PGIS*, for comparison fairness.
  We argue that *PGIS+SWL* might be the most related work to ours, as it considers the fact of read/write frequency to schedule the write requests. But, this mechanism does not take the correlation factor into account, when group hot write requests.
- *Pattern*: which is the newly proposed scheme. It combines *pattern-based write scheduling* and *read balance-oriented wear-leveling* to cut down garbage collection overhead and boost read data performance.

In addition, the number of I/O requests in each time window is setting to 1024. And we select write requests in the first 256 requests in each time window to generate frequent patterns and then the write requests in the same time window match with the patterns to instruct dispatching process. Table III reports the write pattern statistics on the selected traces in our experiments. For example, the *MSRC proj_0* trace has 4126 time windows. Each window includes 15.3 frequent write patterns, and each pattern consists of 6.4 logical sector addresses, in average.

TABLE III
WRITE PATTERN STATISTICS ON SELECTED DISK TRACES

	# of Win	Patterns per Win	Avg. pattern size
MSRC proj_0	4126	15.3	6.4
Random_server_1	4958	15.2	7.7
Random_server_2	6503	13.4	5.7
Random_server_3	11138	8.6	5.5

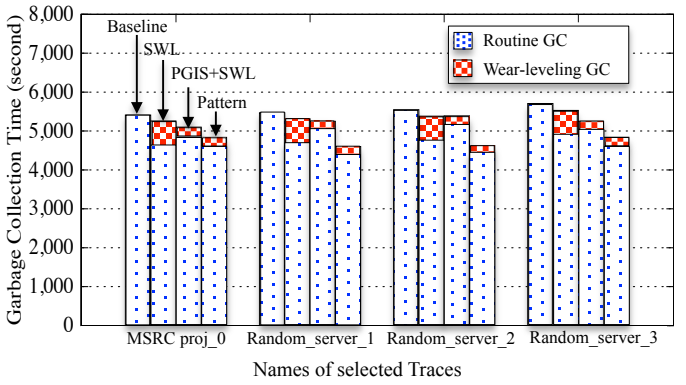


Fig. 5. The time required for garbage collection.

B. Results and Discussions

1) Garbage Collection Cost: Grouping frequent write data and mapping them to the same SSD blocks intend to cut down the overhead of garbage collection. We have recorded the time required for carrying out garbage collection, which includes the time needed for moving valid pages and the time required for block erases. Note that, all comparison counterparts complete the task of wear-leveling by garbage collection. Therefore, the garbage collection time consists of the time of routine garbage collection and the time of wear-leveling relevant garbage collection.

Figure 5 shows the results of processing the selected disk traces. On the one side, *Pattern* and *PGIS+SWL* can reduce the garbage collection time by up to 921.5 seconds, comparing to *Baseline* and *SWL*. On the other side, we can see that *SWL* introduces the largest overhead for completing the wear-leveling task, since more page moves are needed when migrating the cold data. In other words, certain routine garbage collection is done by the wear-leveling triggered garbage collection while adopting *SWL*. This fact verifies that clustering the hot data on same SSD blocks can effectively refrain write amplification during garbage collection.

More importantly, the proposed *Pattern* scheme can further outperform *PGIS+SWL*, and can save up to 14.2% of garbage collection time. As we emphasized, the frequent pattern-based write scheduling takes the correlation factor into account, when grouping relevant write requests together, which can better decrease write amplification in garbage collection.

2) Endurance Improvement: We measure the endurance improvements by two metrics: *first failure time* and *distribution of block erases*. The measurement of first failure time means the time when the first block wears out, and a longer period

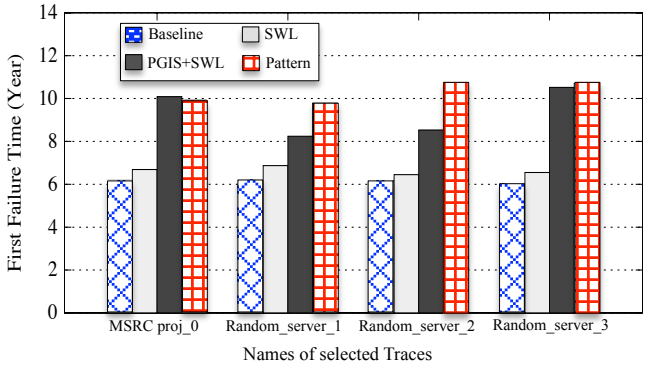


Fig. 6. The results of first failure time in the emulated SSD.

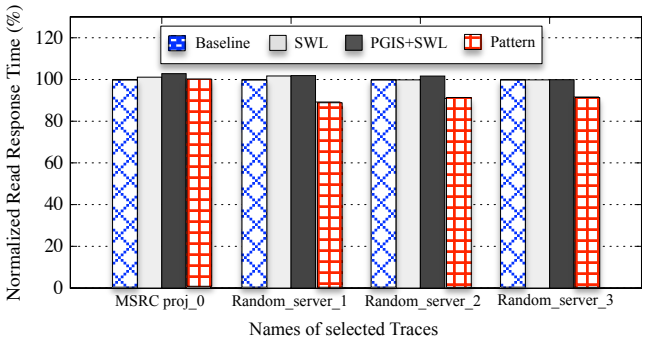


Fig. 7. (Normalized) Average read response time.

of first failure time is preferred. The metric of *distribution of block erases* can be reflected with the total erase numbers and the standard deviation of block erases across all blocks.

Figure 6 shows relevant results of first failure time in our simulated SSD device when processing the collected traces. As seen, the wear-leveling schemes can noticeably yield a longer first failure time, which implies the lifetime of SSD can be prolonged. Furthermore, compared with *SWL*, both *PGIS+SWL* and *Pattern* can achieve more attractive outcomes of first failure time. We argue this is because scheduling the hot write data and other data separately can benefit boosting the effectiveness of wear-leveling.

Table IV reports more details about the erasure counts and the standard deviation of all block erasure numbers. Obviously, the proposed mechanism brings about less (total) erase counts for the selected traces, in contrast to the selected comparison counterparts. In fact, this is the primary reason for *Pattern* causes the least overhead of garbage collection.

Besides, Table IV shows the standard deviation of erasure distribution among all SSD blocks. Clearly, *Pattern* and *PGIS+SWL* yield almost the same standard deviation of erasure distribution, and the related work of *PGIS+SWL* even achieves more attractive standard deviation of erasure distribution for some cases. But note that the newly proposed *Pattern* technique results in less erase counts by up to `17.1%`, in contrast to *PGIS+SWL*.

TABLE V
MAIN MEMORY OVERHEAD FOR KEEPING *PRT* AND *BET* (UNIT: KB)

	16GB	*32GB*	*64GB*	*128GB*
MLC (4KB)	`512+8`	`1024+16`	`2048+32`	`4096+64`
TLC (8KB)	`256+4`	`512+8`	`1024+16`	`2048+32`
QLC (16KB)	`128+2`	`256+4`	`512+8`	`1024+16`

3) Response Performance: The proposed wear-leveling scheme distributes the hot read data across SSD chips, for yielding better system performance. Because the read response time varies when processing different traces, we then report the normalized read response performance in this section, as shown in Figure 7.

Clearly, *Pattern* causes the least average time for responding to the read requests. It proves that distributing hot read data onto different SSD chips can reduce the time required for accessing the required data, by making use of the chip-level internal parallelism. For example, compared with the most related work, i.e. *PGIS+SWL*, our proposal can decrease the response time by `7.1%` in average.

4) Overhead: After presenting the positive effects brought about by the newly proposed mechanism, this section discloses the negative effects of overhead. We first show the main memory overhead resulted by keeping the introduced data structure in our proposal. Next, the mapping overhead caused by clustering frequent write requests with patterns is reported. At last, the wear-leveling cost is discussed.

Memory overhead: Because a one-bit flag is needed for each page, *PRT* results in the major memory-space overhead on the controller so as to maintain the read status of SSD pages. In addition, *BET* requires memory space for keeping a one-bit flag for each block. As shown in Table V, the memory overhead depends on the type and the capacity of SSDs. For instance, the needed memory size is `4MB+64KB`, for a 128GB MLC flash memory.

Mapping Overhead: Analyzing frequent access pattern sets and matching requests with the sets must bring about certain overhead. This section presents the time needed for mapping logical sector numbers to the physical page numbers at Flash Translation Layer, during processing the disk traces.

Figure 8 shows the results of mapping overhead by taking advantage of three schemes. As illustrated, *PGIS* and *Baseline* cause similar time for mapping the requests before flushing the data to the SSD blocks. But, *Pattern* does bring about a little more time for grouping the relevant write request, by less than `138ms` in all cases. In brief, the proposed mechanism of pattern-based mapping can cut down the number of page movements in garbage collection and the number of block erasures, with acceptable mapping overhead.

Wear-leveling Overhead: In order to reduce the overhead of wear-leveling, we adopt the policy that completes the task of wear-leveling by garbage collection [7]. Thus, the overhead of wear-leveling was presented as a part of garbage collection overhead (the red part in the bar) in Figure 5. As seen, except for *Baseline*, that does not carry out wear-leveling, the newly

TABLE IV
ERASURE STATISTICS

Traces	Total Block Erases				Standard Deviation of Block Erases			
	Baseline	*SWL*	*PGIS+SWL*	*Pattern*	*Baseline*	*SWL*	*PGIS+SWL*	*Pattern*
MSRC proj_0	1206907	1270809	1226205	1223387	17.145858	14.030977	8.988854	9.134764
Random_server_1	1208876	1274601	1242155	1217567	17.434723	14.029503	9.798412	9.288180
Random_server_2	1211246	1273852	1269034	1222484	17.345198	13.999297	8.673104	9.173870
Random_server_3	1215976	1277092	1242155	1223387	17.319114	13.960486	9.811857	9.378567

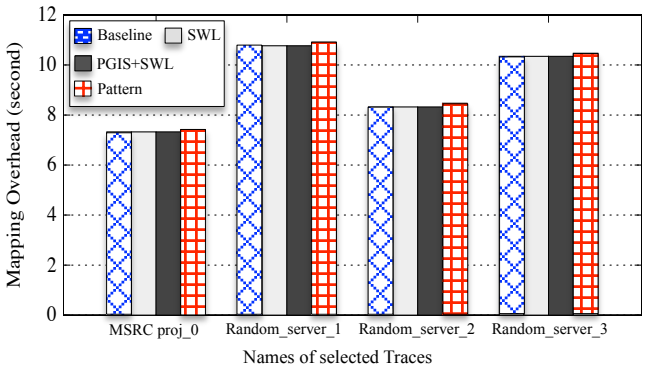


Fig. 8. Mapping overhead comparison of three schemes.

proposed mechanism does cause the least time for completing the task of wear-leveling.

C. Summary

The evaluation experiments have demonstrated that our newly proposed mechanism can effectively and practically reduce the garbage collection overhead, as well as the response time for reading data.

With respect to comparing the proposed scheme to conventionally used scheduling and wear-leveling algorithms for SSDs, we emphasize the following two key observations. *First*, the proposed pattern-based write scheduling is able to achieve better relative performance on garbage collection, through resulting in less valid data movements and erase operations. *Second*, the approach of read balance-oriented wear-leveling cannot only yield an even distribution of erasures, but also offers attractive I/O responses for reading data. This is because the hot read data are distributed onto different chips of SSD channel in the process of wear-leveling.

V. CONCLUSION

In this paper, we first have proposed a frequent pattern-based I/O scheduler for dispatching write requests, to purposely cut down the garbage collection overhead. Specifically, it maps the hot write data that are most probably to be updated in the near future into the same SSD blocks having a small erasure count. Therefore, the cost of reclaiming such blocks after the relevant data have been invalidated can be decreased greatly. After that, we have introduced a read balance-oriented wear-leveling method. It targets at not only extending the lifetime of SSDs by maintaining an even block erasure distribution, but also exploiting the internal parallelism of SSDs to speed up read responses.

From the simulation evaluations on the selected disk traces of real applications, we can conclude that the newly proposed write scheduling mechanism can significantly cut down the overhead of garbage collection by up to 16.6%. In addition, the approach of read balance-orient wear-leveling can enhance the read responsiveness for hot read data by more than 5.2%, as well as maintain an uniform block erasure distribution to yield an attractive lifetime of SSDs.

ACKNOWLEDGMENT

This work was partially supported by "National Natural Science Foundation of China (No. 61872299)", "Natural Science Foundation Project of CQ CSTC (No. CSTC2018jcyjAX0552)", "Hunan Provincial Natural Science Foundation of China (No. 2018JJ2309), and "Chongqing Graduate Research and Innovation Project (No. CYS18090)". The authors would like to thank Prof. Yutaka Ishikawa@RIKEN Japan, for his valuable advice on performing evaluation experiments.

REFERENCES

[1] L. Grupp, J. Davis, and S. Swanson. The bleak future of NAND flash memory. In Proceedings of *the 10th USENIX conference on File and Storage Technologies*, Article No. 2, 2012.

[2] C. Gao, L. Shi, and Y. Di et al. Exploiting Chip Idleness for Minimizing Garbage CollectionInduced Chip Access Conflict on SSDs. *ACM Transactions on Design Automation of Electronic Systems*, Vol. 23(2), Article No. 15, 2018.

[3] C. Matsui, C. Sun, and K. Takeuchi. Design of hybrid SSDs with storage class memory and NAND flash memory. Proceedings of the IEEE, Vol. 105(9): 1812-1821, 2017.

[4] H. Yassine, J. Coon, and D. Simmons. Index programming for flash memory. IEEE Transactions on Communications, Vol. 65(5): 1886-1898, 2017.

[5] J. Liao, F. Zhang, and L. Li et al. Adaptive wear-leveling in flash-based memory. *IEEE Computer Architecture Letters*, Vol. 14(1): 1–4, 2015.

[6] Y. Cai, S. Ghose, and E. Haratsch et al. Errors in Flash-Memory-Based Solid-State Drives: Analysis, Mitigation, and Recovery[J]. arXiv preprint arXiv:1711.11427, 2017.

[7] Y. Chang, J. Hsieh, T. Kuo. Improving flash wear leveling by proactively moving static data. *IEEE Transactions on Computers*, Vol. 59(1):53–65, 2010.

[8] L. Chang, T. Chou, and L. Huang. An adaptive, low-cost wear-leveling algorithm for multichannel solid-state disks. *ACM Transactions on Embedded Computing Systems*, Vol. 13(3):55:1–55:26, 2013.

[9] M. Yang, Y. Chang, C. Tsao, and P. Huang. New ERA: new efficient reliability-aware wear leveling for endurance enhancement of flash storage devices. In Proceedings of the 50th Annual Design Automation Conference (*DAC '2013*), pp. 163:1–163:6, 2013.

[10] S. Kim, J. Choi, and J. Kwak. RRWL: Round Robin-Based Wear Leveling Using Block Erase Table for Flash Memory. *IEICE Transactions on Information And Systems*, Vol. E100-D(5):1124–1127, 2017.

[11] Y. Chang, J. Hsieh, and T. Kuo. Endurance enhancement of flash-memory storage, systems: An efficient static wear leveling design. In Proceedings of the 44th annual Design Automation Conference (*DAC '2007*), pp. 212–217, 2007.

[12] J. Guo, Y. Hu, B. Mao, and S. Wu. Parallelism and Garbage Collection Aware I/O Scheduler with Improved SSD Performance. In Proceedings of 2017 IEEE International Parallel and Distributed Processing Symposium (*IPDPS '2017*), pp. 1184–1193, 2017.

[13] S. Huang and L. Chang. Exploiting Page Correlations for Write Buffering in Page-Mapping Multichannel SSDs. *ACM Transactions on Embedded Computing Systems*, Vol. 15(1):12:1–12:25, 2016.

[14] S. Kwon, and T. Chung. Hot-LSNs distributing wear-leveling algorithm for flash memory. *ACM Transactions on Embedded Computing Systems*, Vol. 12(1s):62:1–62:28, 2013.

[15] Q. Li, L. Shi, and C. Gao et al. Access Characteristic Guided Read and Write Regulation on Flash based Storage Systems. *IEEE Transactions on Computers*, Vol. 67(12): 1663-1676, 2018.

[16] Q. Li, L. Shi, and C. Xue et al. Access Characteristic Guided Read and Write Cost Regulation for Performance Improvement on Flash Memory. In Proceedings of 14th USENIX Conference on File and Storage Technologies (*FAST '2016*), pp. 125–132, 2016.

[17] Y. Hu, H. Jiang, and D. Feng et al. Exploring and exploiting the multi-level parallelism inside SSDs for improved performance and endurance. *IEEE Transactions on Computers*, Vol. 62(6):1141–1155, 2013.

[18] Y. Hu, H. Jiang, and D. Feng et al.. Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity. In Proceedings of the international conference on Supercomputing, pp. 163:1–163:6, 2011.

[19] D. Narayanan, E. Thereska, and A. Donnelly et al. Migrating server storage to SSDs: analysis of tradeoffs. In Proceedings of ACM European conference on Computer systems, pp. 145–158, 2009.

[20] `http://iotta.snia.org/traces/388`.

[21] C. Borgelt. An implementation of the FP-growth algorithm. pp. 1–5, 2005.

[22] M. Murugan, and D. Du. Rejuvenator: A static wear leveling algorithm for NAND flash memory with minimized overhead. In Proceedings of IEEE 27th Symposium on Mass Storage Systems and Technologies (*MSST '2011*), pp. 1-12, 2011.

[23] S. Hahn, J. Kim, and S. Lee. To collect or not to collect: Just-in-time garbage collection for high-performance SSDs with long lifetimes. In Proceedings of *DAC '2015*, 2015.

[24] V. Houdt. On the necessity of hot and cold data identification to reduce the write amplification in flash-based SSDs. *Performance Evaluatio*n, Vol. 82: 1-14, 2014.

[25] P. Tan, M. Steinbach, and A. Karpatne et al. Introduction to data mining. Pearson Education India, 2007.

[26] M. Yang, Y. Chang, and T. Kuo et al. Reducing data migration overheads of flash wear leveling in a progressive way. IEEE Transactions on Very Large Scale Integration Systems, Vol. 24(5): 1808-1820, 2016.

[27] A. Jagmohan, M. Franceschini and L. Lastras. Write amplification reduction in NAND Flash through multi-write coding. In Proceedings of IEEE 26th Symposium on Mass Storage Systems and Technologies (*MSST '2010*), pp. 1-6, 2010.