Intra-page cache update in SLC-mode with partial programming in high density SSDs

Jun Li, Minjun Li, Zhigang Cai, François Trahay, Mohamed Wahib, Balazs Gerofi, Zhiming Liu, Min Huang, Jianwei Liao

Intra-page Cache Update in SLC-mode with Partial Programming in High Density SSDs

Jun Li
Southwest University of China
junli95@email.swu.edu.cn

Minjun Li
Southwest University of China
liminjun@email.swu.edu.cn

Zhigang Cai*
Southwest University of China
czg@swu.edu.cn

Francois Trahay
Telecom SudParis
francois.trahay@telecom-sudparis.eu

Mohamed Wahib
National Institute of Advanced Industrial Science and Technology
RIKEN Center for Computational Science
mohamed.attia@aist.go.jp

Balazs Gerofi
RIKEN Center for Computational Science
bgerofi@riken.jp

Zhiming Liu
Southwest University of China
zhimingliu88@swu.edu.cn

Min Huang
Southwest University of China
hmin@swu.edu.cn

Jianwei Liao
Southwest University of China
State Key Lab. for Novel Software Technology, Nanjing University
liaotoad@gmail.com

ABSTRACT

Modern high density SSDs commonly designate a part of their capacity as a cache using an Single-level Cell (SLC)-mode region. Partial programming is then adopted for reducing space fragmentation in the SLC-mode pages, but it exacerbates program disturb including both in-page disturb and neighbouring page disturb. This paper proposes a partial programming scheme (called `intra-page update`) by updating hot, small size data inside a given page to minimize the negative impact induced by program disturb. Moreover, we introduce a novel data movement principle to separate hot and cold write data in the SLC-mode cache when updating the data or carrying out garbage collection. As a result, the hot updated data can be kept in the SLC-mode cache and the cold data will be flushed onto the high density SSD region. Simulation tests on several realistic disk traces show that our proposal improves bit error rate by `9.2%`, and I/O performance by `9.3%` on average, compared to state-of-the-art methods, without a noticeable decrease in total endurance.

CCS CONCEPTS

• **Computer systems organization** → **Embedded software**;

KEYWORDS

SSDs, SLC-mode Blocks, Partial Programming, Hot/Cold Data Separation, I/O Performance, P/E Cycles

ACM Reference Format:
Jun Li, Minjun Li, Zhigang Cai, Francois Trahay, Mohamed Wahib, Balazs Gerofi, Zhiming Liu, Min Huang, and Jianwei Liao. 2022. Intra-page Cache Update in SLC-mode with Partial Programming in High Density SSDs. In

*Corresponding author.

ICPP '21, August 9–12, 2021, Lemont, IL, USA
2021.

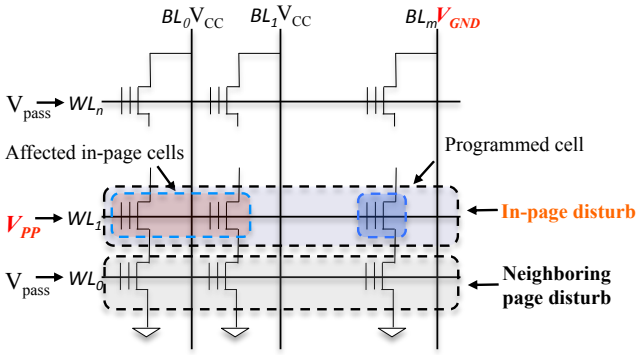

Figure 1: Illustration of partial programming-induced disturb.

Proceedings of 50th International Conference on Parallel Processing (ICPP '21). ACM, New York, NY, USA, 10 pages.

1 INTRODUCTION

Flash memory-based SSDs have become the dominant storage devices thanks to their nature of small size, energy efficiency, low latency, and collectively massive parallelism [1]. In order to further cut down the per-unit price of SSDs, the feature size of NAND flash memory cells has been pushed to the limit of the `nm` level [2]. Consequently, flash density increase is now driven by multiple-level cell (MLC) and triple-level cell (TLC) [3, 4]. However, such high density flash memory has lower read/write performance and lower endurance than single-level cell (SLC) flash memory [5, 6]. In order to hide the low performance and extend the lifetime of high density SSDs, most recent TLC/MLC SSDs adopt a hybrid SSD architecture, which contains an SLC region in addition to the multi-level cells [8, 9]. Specifically, the SSD blocks in the SLC region are

programmed in SLC-mode (i.e., they store only one bit per cell), and thus they can offer lower access latency and better erase endurance than the high density blocks.

On the other hand, the high density SSDs commonly have a relative large flash page size, which leads to a mismatch between the request size and the underlying write unit. This has been reported as the main cause of page fragmentation (or called as internal fragmentation) [12, 16]. For better utilization of the SSD page space, the technique of partial programming has been introduced that enables subpage programming in varied parts of an SLC-mode page [10, 11]. As shown in Figure 1, the data that is smaller than a page (i.e., 16KB in the paper) can be partially programmed. Specifically, a high partial programming voltage of V_{pp} is applied to the target word line of WL_1, while a pass voltage of V_{pass} is imposed to the other word lines. Meanwhile the bit lines except for programming cells are driven to a V_{CC}, for reducing the voltage drop across the tunnel oxide. Then, different parts of a page can be programmed multiple times, which can consequently alleviate the problem of page fragmentation [12, 17].

However, partial programming exacerbates program disturb towards not only the data in the neighboring pages, but also the previously flushed data in the same page [19]. Figure 1 also illustrates both kinds of disturb caused by a partial programming process, i.e., ***in-page disturb*** and ***neighboring page disturb***. Consequently, more Error Correction Code (ECC) time is needed to correct the disturbed data when reading them. Considering this fact, SSD manufacturers suggest the number of partial programming to the same SLC page should be limited to 4, because more partial programming operations will increase the read latency, and even destroy existing data in the same page and neighboring pages [11, 18].

In addition, partial programming requires a second-level mapping table to record pairs of a logical address and the corresponding physical address for the subpages in SLC-mode pages, which results in higher address translation latency and needs more memory for holding the mapping table.

To address the issues of eliminating in-page disturb and minimizing the size of second-level mapping table with respect to partial programming in modern high density SSDs, this paper proposes an intra-page cache update scheme to efficiently buffer small hot write data in SLC-mode pages. In brief, this paper makes the following contributions:

- We introduce an intra-page update scheme with partial programming in the SLC-mode cache of high density SSDs, where the basic idea is to program small size updated data inside a given page. Thus, the in-page disturb induced by partial programming can be obliterated. In addition, it is not needed to maintain a second-level mapping table to record the subpage information, since an SLC-mode page only holds the valid data from a single request.
- We present a data movement method to separate hot and cold write data in the SLC-mode cache when updating data or carrying out garbage collection (GC). It utilizes three levels of blocks to shift the hot updated data and then migrates cold data to the low-level blocks till the data is not in the SLC-mode cache.
- We evaluate our proposal on several disk traces of real-world applications. As measurements indicate, our proposal reduces I/O response time by 9.3%, and decreases read error rate by 9.2% on average, compared to state-of-the-art methods.

Table 1: Size distribution of updated requests in block I/O traces

Trace	Size∈(0, 4K]	Size∈(4K, 8K]	Size>8K
*ts*0	**69.8%**	17.9%	12.3%
*wdev*0	**73.2%**	6.8%	20.1%
*lun*1	**85.2%**	7.3%	7.5%
*usr*0	**66.3%**	12.1%	21.6%
*lun*2	**92.6%**	2.5%	4.9%
ads	**74.5%**	14.1%	11.4%

The remainder of the paper is organized as follows. Section 2 introduces related work on SLC-mode cache in high density SSDs and the motivations. Section 3 designs the proposed intra-page update approach and hot/cold data management in the SLC-mode cache. Section 4 shows the evaluation methodology and reports the experimental results. Finally, the paper is concluded in Section 5.

2 RELATED WORK AND MOTIVATION

2.1 Related Work

By rethinking the write/read/erase granularity of NAND flash memory, the technologies of partial programming [10, 12], partial read [13], partial erase [14, 15] have been proposed to better serve I/O requests with smaller sizes. Many studies adopt partial programming to improve SLC-mode page space management. Kim et al. [10] presented a method to partition a SLC-mode page into smaller subpages and then support subpage programming. Therefore, the SSD endurance cycle can be greatly enhanced because of better space utilization. Similarly, Feng et al. [12] designed a subpage management method for SLC-mode pages to enhance the lifetime of the native MLC SSDs, which makes use of a two-level mapping table to record subpage information in the same pages. Although these partial programming schemes can greatly alleviate the fragmentation problem, they need more memory space for the mapping table and induce in-page disturb, accompanied with more read errors.

Considering that partial programming induces more raw bit error rate, Kim et al. [17] proposed a subpage-aware retention model to avoid worsening data retention by exploiting bit error rates induced by partial programming. Besides, Zhang et al. [19] proposed an in-place delta compression in a SLC-mode page, which first holds the original data in the given page and keeps the updated compression parts and ECC information in the same page by utilizing partial programming.

However, partial programming does pose the threat on both in-page error and neighboring page disturb. These errors should be taken into consideration.

2.2 Motivation

Previous work has reported that partial programming incurs higher bit error rates than conventional programming [19]. As shown in Figure 2, for instance, the block having 4000 Program/Erase (P/E)

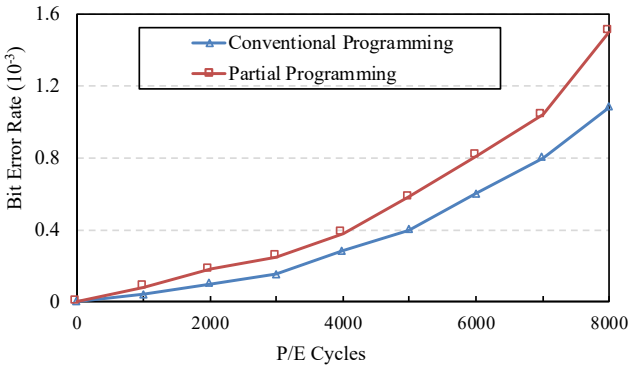

Figure 2: Comparison of the bit error rate of conventional programming and partial programming [19].

cycle indicates `0.00028` and `0.00038` bit error rate with conventional programming and partial programming, respectively. That is, progressive partial programming may bring about negative effects on bit error rates towards the valid data that have been programmed within the same page. Intuitively, it adversely effects read latency due to the longer ECC decoding time on the in-page data affected by partial programming. Moreover, the bit error rate difference becomes more pronounced as the P/E cycle is getting large [12, 19]. On the other hand, invalid or free (sub)pages are not affected by program disturb, as they have been labeled as dirty portions or not been previously programmed [17].

We have analyzed the distribution of updated request sizes in several block I/O traces of real-world applications [20–22]. Table 1 presents the statistics. As seen, `4K`-size requests account for more than `66.3%` of all requests. This fact verifies that we can better utilize SSD space by flushing multiple pieces of a small size data into the same page with partial programming, since applications have many small size update requests. In addition, we have also observed that more than a half of requests are identified as cold that are not updated frequently in the selected traces. Then, it is necessary to separately manage the data according to their update frequency, for better efficiently utilizing the SLC-mode cache in high density SSDs.

Such observations motivate us to keep the small size updated data in the remaining free space inside of the same page by using partial programming called **Intra-page Update**, to eliminate negative impacts on previously flushed in-page data induced by partial programming. Furthermore, it is expected to categorize the data according to their update hotness, and separately save them in either SLC-mode blocks or other native high density blocks of SSDs (e.g. TLC or MLC blocks), to boost the use efficiency of the SLC cache and extend the lifetime of high density SSD blocks.

3 DESIGN AND IMPLEMENTATION

This section describes the design of the proposed intra-page update scheme and provides details on our implementation.

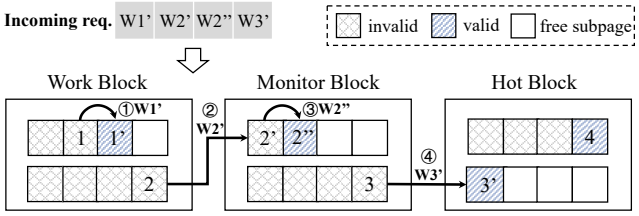

Figure 3: Overview of intra-page update with upgraded data movement.

3.1 Intra-page Update with Partial Programming

The basic principle of our approach is to utilize partial programming to fulfill small size of update requests in the same pages that hold the previous version of data. Then, the program disturb caused by partial programming can be mitigated, since obsolete data have been invalidated and they never suffer from program disturb. We define three levels of SLC-mode blocks, to identify hot write data and keep them in the cache. Specifically, they are *High-density Block*, *Work Block*, *Monitor Block*, and *Hot Block*, with the ascending order. The first one is the native blocks of high density SSDs, and the remainders are with the SLC-mode.

Figure 3 illustrates a high-level overview of proposed intra-page update with partial programming. The new data should be flushed into SSDs, and a *Work Block* is the target cell. When the updated request comes, it should be flushed onto the page which has the previous version of data, such as ❶ `W1'`. In case that the remainder free space in the same page cannot accommodate the updated data, the data will be flushed on to a free page with a higher level label. For instance, the data of `2'` should be written into a new page in a *Monitor Block* (i.e. the upgraded data movement), and it also indicates this piece of data is frequently updated. Similar to the case of ❷ `W2'`, the data of ❹ `W3'` also should be flushed into a new page in a *Hot Block*.

In brief, we can identify cold/hot update data, by using the intra-page update scheme with partial programming. Then, we can keep hot update data in high level SLC-mode blocks for better I/O responsiveness on the one side, and maintain the cold data in the *Work Blocks* or even move them to *High-density Blocks* via GC operations for freeing SLC cache space on the other side.

3.2 GC Policy for SLC-mode Cache

The greedy policy has been commonly used in a conventional GC process of SSDs. Specifically, it traverses all the blocks in the target plane, and selects the block that has the largest number of invalid pages. But, in the context of partial (subpage) programming, the granularity of invalid page number should be changed into the subpage level. That is, when the GC threshold is reached, we should select the GC target by referring to the number of invalid subpages. For this end, we define the metric of invalid subpage ratio (i.e. *ISR*) in Equation 1, to represent the capacity of freeing space after a GC operation on the candidate block. Then, the block having the largest *ISR* value will be selected as the GC target.

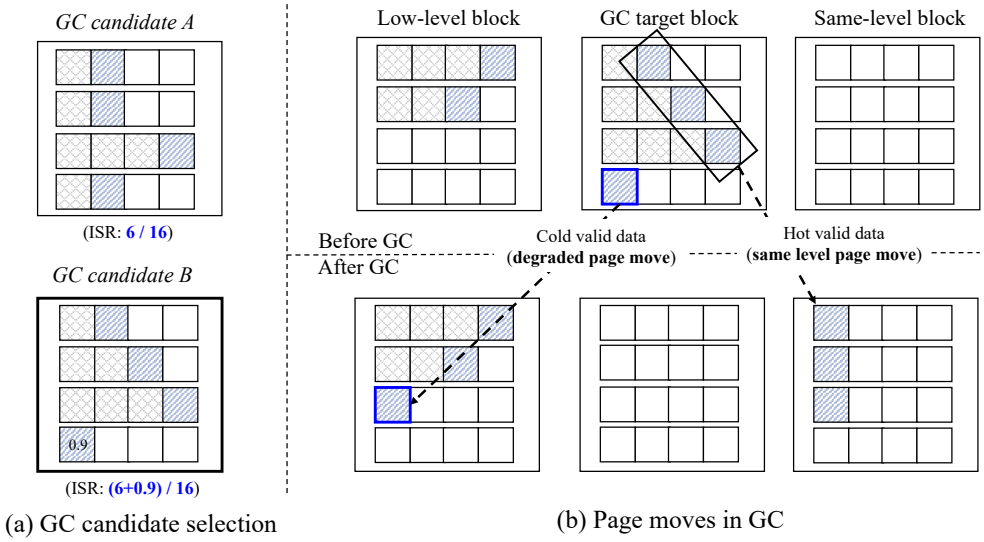

Figure 4: GC illustration in SLC-mode blocks with degraded data movement.

$$ISR_i = \frac{IS_i + IS'_i}{TS_i} \tag{1}$$

where IS_i and TS_i mean the number of invalid subpage and total subpage of i_{th} block, and IS'_i measures the weight of valid subpage in i_{th} block.

Note that the cold write data is calculated through IS'_i, as we consider that it is better to eject such cold valid data from the SLC-mode cache.

$$IS'_i = \sum_{j \in J} (1 - e^{-\frac{t_{ij}}{T_i}}) \tag{2}$$

where J is the index set of all subpages in the i_{th} block that have never been updated, t_{ij} means the access interval time of j_{th} subpage in the i_{th} block, and T_i represents the average access interval time in all subpages. Assuming that the number of updates associating with each subpage per unit time follows the *Poisson* distribution with the parameter of $\frac{1}{T_i}$ [23], the access interval time of each subpage in the block will follow the exponential distribution with the parameter of $\frac{1}{T_i}$. That is to say, $1 - e^{-\frac{t_{ij}}{T_i}}$ indicates that the probability of the access interval time of the un-updated j_{th} subpage is less than t_{ij}, which can be used to approximately measure the invalid degree of the subpage.

In breif, while the value of IS_i can estimate the number of invalid subpages (i.e. invalid one as 1), the value of IS'_i indicates the weight of valid subpages, that ranges from `0` to 1. If t_{ij} is long enough to T_i, the IS'_i value of subpage may approach to 1. Then, our GC selection metric is an increasing function of the number of invalid subpage and the weight value of valid subpage. In other words, The block having more invalid subpages and cold valid subpages prefer to be processed for GC, which reaches another purpose of separating and ejecting cold data during GC process.

As the example shown in Figure 4(a), the value of *ISR* of *GC candidate A* is `6/16` (`6` divided by `16` total subpages), and that of *GC candidate B* is `6.9/16` (`6` of *IS* and `0.9` of *IS'* divided by `16` total subpages). Then, *GC candidate B* will be selected for carrying out a GC operation, since we can reclaim more available space.

Moreover, we introduce a degraded page movement scheme in GC for sifting the cold valid data, so that they will be moved onto a lower level block, and finally ejected from the SLC-mode cache to a *High-density Block*. Figure 4(b) illustrates a GC example in our context. As read, the valid cold subpages have not been updated in that level of blocks, and they should be migrated into low-level blocks (e.g. from *Monitor Block* to *Work Block*). Otherwise, other subpages should be moved in the same-level blocks for preserving their hotness. Note that, if the pages in *Work Blocks* have not been updated, they need to be ejected from the SLC-mode cache.

3.3 Implementation

Algorithm 1 shows the implementation details on intra-page update with partial programming and GC operations in the SLC-mode cache. As illustrated, Lines `2-13` identify the process of dealing with write requests. The new data (not the updated data) will be directly flushed to a new page in a *Work Block* (Line `5`). For the updated data, the page having their old data is the target of partial programming if the remainder space in the page is larger than the size of updated data (see Line `8`). Otherwise, a new page in the a higher level block (see Line `11`) will be selected to hold the latest data. Note that, the `block_flag` identifies the three-level blocks, and lower level blocks can be instead selected only if no available block can be found.

In addition, Lines `14-19` present the process of page move in the GC operation. Specifically, when the page data have been updated, they should be moved onto a same level SLC-mode block. Otherwise, we refer these data as the cold data, and migrate them onto a lower level block till the data is not in the SLC-mode cache (see Line `18`).

```
Input: args of req, args of block, available_slc_space;
Output: null;
/*block_flag (0, 1, 2, 3) stand for (High-density, Work,
  Monitor, Hot) block respectively*/
Function insert_slc_buffer(req)
    if search_map_table(req) == NULL then
        /*new data, find a new page in Work Block*/
        find_page(1,size);
    end
    else if size < page_left_space then
        /*intra-page update*/
        goto Line 12;
    end
    else
        /*find a new page in a higher level block*/
        find_page(block_flag+1,size);
    end
    write_page();
    update_map_table();
Function move_page(block_flag)
    for valid_page in block do
        find_page(block_flag-1+update,size);
        /*page was updated ? update == 1 : 0*/
        write_page();
        update_map_table();
    end
/*main function starts*/
insert_slc_buffer(req);
if available_slc_space < gc_threshold then
    select_target_block();
    move_page(block_flag);
    block_flag = −1; //reset block flag
    erase_block();
end
```

Algorithm 1: Intra-page partial programming update

4 EXPERIMENTAL EVALUATION

4.1 Environment Setup

We have performed trace-driven simulation with SSDsim [24], which has been modified to support partial programming. Table 2 demonstrates our settings of experiments. In the table, page settings, latencies of SLC/MLC mode[1], and read/write/erase information are referred to [8, 9, 25], and Bose-Chaudhuri-Hocquenghem (BCH) ECC settings are referred to [26]. P/E cycle is set as 4000 in default. For reflecting varieties of SSD use stages, the case study of different P/E cycles will be discussed in Section 4.5. Beside, the bit error rate induced by partial programming is referred to [19].

We employed 6 commonly used disk traces. Specifically, *ts0*, *wdev0* and *usr0* are from the block I/O trace collection of Microsoft Research Cambridge [20], and *ads* is from Microsoft Production Server [21]. The reminder two recent block I/O traces are recently collected from a part of an enterprise virtual desktop infrastructure (VDI) [22]. Specifically, they are additional-01-2016021615-LUN0 (*lun1*), additional-03-2016021719-LUN2 (*lun2*). The detailed specifications on the traces are reported in Table 3, and the metric of *Hot Write* means the ratio of hot access addresses if they have been requested not less than 4 times.

[1]We take the MLC SSDs as the case of high density SSDs in our tests, as we have MLC hardware statistical data on bit error rates induced by partial programming.

Table 2: Experimental settings of *SSDsim*

Parameters	Values	Parameters	Values (ms)
Block number	65536	*SLC read time*	0.025
SLC mode ratio	5%	*MLC read time*	0.05
SLC/MLC Page	64/128	*ECC min time*	0.0005
Page size	16KB	*ECC max time*	0.0968
GC threshold	5%	*SLC write time*	0.3
Wear-leveling	static	*MLC write time*	0.9
FTL scheme	Page	*Erase time*	10

Table 3: Specifications on selected traces (ordered by write ratio)

Trace	# of Req.	Write R	Write SZ	Hot write
ts0	1,801,734	82.4%	8.0KB	50.5%
wdev0	1,143,261	79.9%	8.2KB	58.2%
lun1	1,073,405	73.1%	7.6KB	10.0%
usr0	2,237,889	59.6%	10.3KB	36.5%
lun2	1,758,887	19.3%	9.7KB	8.5%
ads	1,532,120	9.5%	7.0KB	18.3%

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

- *Baseline*: which indicates the default dynamic page-level mapping scheme, and partial programming is not enabled.
- *MGA (Mapping Granularity Adaptive Method)*: which utilizes sub-page granularity management with partial programming, to boost space utilization. We argue that it is the most related work to our proposal.
- *IPU (Intra-page Update)*: which is the proposed method. It supports intra-page update with partial programming and separates hot and cold write data in the SLC-mode cache, when updating data or carrying out GCs.

4.2 Tests and Benefit Illustration

To measure validity of the proposed mechanism that aims to utilize the SLC mode cache, we use the following two metrics in our tests: (a) *average latency*, (b) *read error rate*.

4.2.1 I/O Performance. Figure 5 presents the results of I/O latency distribution after replaying the selected traces. Compared with *Baseline*, both *MGA* and *IPU* can reduce the overall I/O time by 6.4% and 14.9% on average. We consider partial programming in the SLC-mode can reduce space fragmentation to improve space utilization, so that the SLC-mode cache can absorb more requests for better I/O performance.

More exactly, our proposal of *IPU* decrease write latency by 23.8% and 17.9%, compared with *Baseline* and *MGA*. This is because

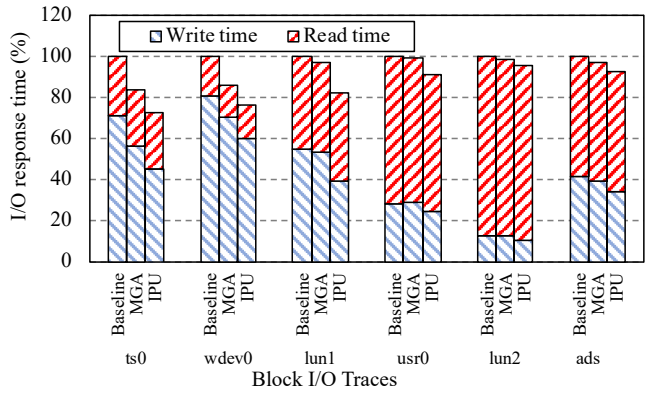

Figure 5: I/O response time distribution after running the selected block I/O traces.

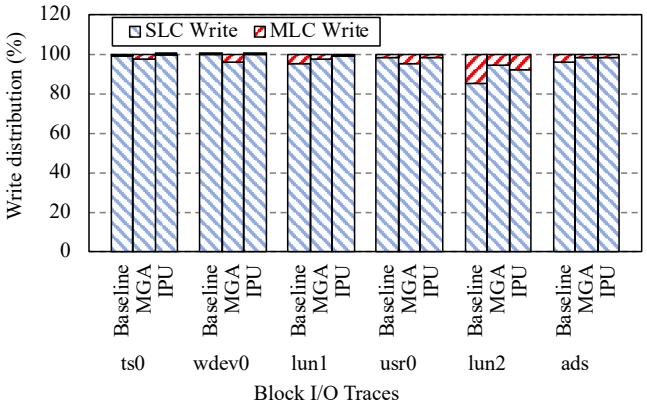

Figure 6: Completed writes distribution in SLC/MLC blocks after running the selected block I/O traces.

our proposal can efficiently identify hot update data, and then absorb them in the SLC-mode cache by updating the data in the same page. Besides, the proposed data movement policy in GCs can also contribute to separate hot and cold update data, and move them to SLC-mode blocks or MLC blocks. In order to clearly illustrate this fact, we present the distribution of writes in both SLC-mode and MLC regions with Figure 6. As seen, *IPU* yield the lowest count in the MLC region, that indicates the SLC-mode cache can fulfill a large number of (hot) write requests and lead to better write performance than *Baseline* and *MGA*. Note that this is also the cause of resulting in more erases in SLC-mode blocks and less erases in MLC blocks (see Section 4.3).

On the other hand, Figure 7 shows the updated write distribution occurred in there-level blocks. As seen, write completed in *Hot Block* by `32.9%` on average. These write data are the most frequently updated data, and hold in SLC-mode blocks to utilize the I/O benefits. `62.7%` of write occurred in *Work Block*, which infrequently updated (cold) data should be ejected into *High-density Block* or hot data could be flushed into *Monitor Block* and eventually into *Hot Block*.

Another interesting clue shown in Figure 5 is that our proposal of *IPU* can decrease the read latency by up to `6.3%`, compared

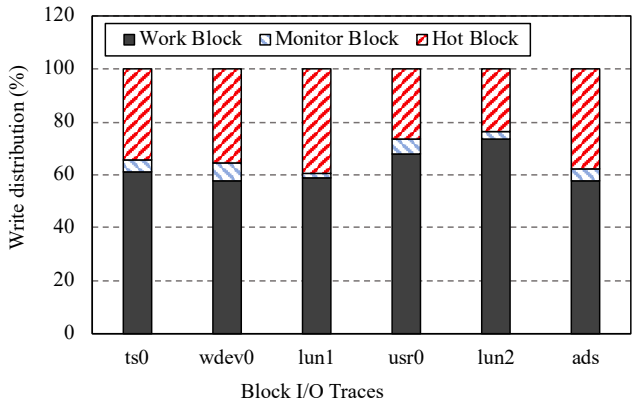

Figure 7: Occurred writes distribution in there-level blocks using the proposed method *IPU*.

with the most related work of *MGA*. This is because our proposal can decrease read error rate induced by partial programming, and then the ECC decoding time can be greatly reduced. The detailed information on the measurement of read error rate will be presented in Section 4.2.2.

4.2.2 Read Error Rate. Figure 8 shows results of read error rate after replaying selected block I/O traces. Both *MGA* and *IPU* enlarge the read error rate by `14.0%` and `3.5%` on average, compared with *Baseline*. As seen, partial programming does increase the bit error rate due to program disturb. But we can understand that *IPU* can mitigate the most of read error rates, in contrast to the related work of *MGA*. This is because our intra-page update can eliminate program disturb on the intra-page data, only slight neighboring program disturb increasing. Note that the read error rate impacts read latency, so *IPU* can achieve an improvement on read responsiveness, compared with *MGA*, which have been reported in Section 4.2.1.

4.3 Space Utilization and Endurance Analysis

Partial programming can reduce page fragmentation and then improve SSD endurance, this section unveils measurements of page utilization and block erases.

4.3.1 Page Utilization in SLC-mode Cache. We define the metric of page utilization as the ratio of the number of used subpages to the total number of subpages in all GC blocks in the SLC-mode cache. Figure 9 shows the results of page utilization after replaying all traces, in which a larger ratio means better page utilization.

As illustrated, the page utilization achieved by *Baseline* is about `52.8%` on average, since it does not enable partial programming and then brings about a serious fragmentation problem. On the other side, *MGA* and *IPU* yield `99.9%` and `73.0%` page utilization respectively on average, and we see the page fragmentation problem is alleviated.

Another interesting clue shown in the figure is that our proposal of *IPU* does worse in the measurement of page utilization than *MGA*. This is because *MGA* tries to aggregate many pieces of small size data belonging to different write requests into the same page,

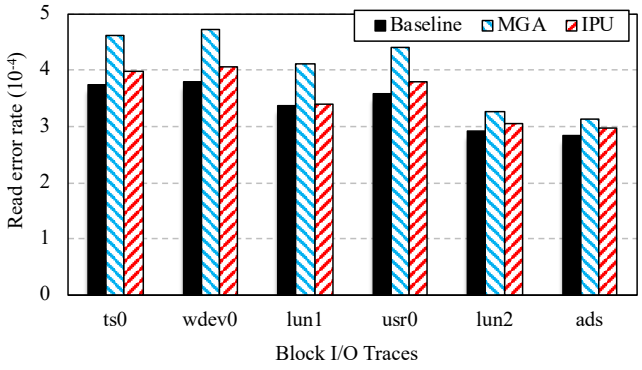

Figure 8: Average read error rate of the selected block I/O traces.

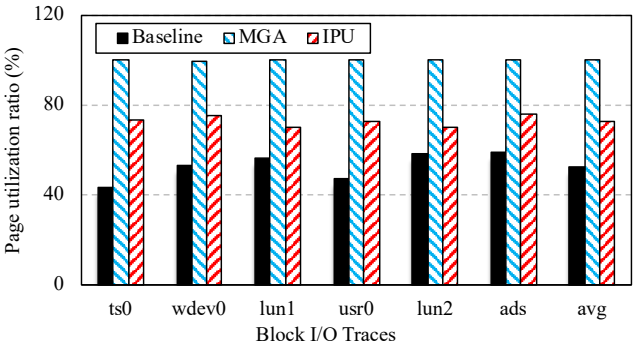

Figure 9: Page utilization ratio of GC blocks in the SLC-mode cache.

and the space utilization is the most important goal of *MGA*. But *IPU* aims to support intra-page update for minimizing program disturb, as well as separate hot and cold data for better utilization of SLC-mode cache. We emphasize that the metric of page utilization reflects the level of page fragmentation in the SLC-mode cache, but it does not directly indicate I/O performance, which have been demonstrated in Section 4.2.

4.3.2 Erase Number. We record erase numbers in both SLC-mode and MLC blocks to reflect SSD endurance after running the benchmarks, and Figure 10 shows the results. Figure 10(a) reports the numbers in the SLC-mode cache, and it shows that *Baseline* performs the worst, because it does not support partial programming. More importantly, we see our proposal of *IPU* results in more SLC erases compared with the related work of *MGA*, this is because *MGA* can yield better page utilization, and endure less write requests in the SLC-mode cache.

Figure 10(b) presents the erase statistical data in MLC blocks, and the most important information is that *IPU* yields the least number of erases in MLC blocks. This is because *IPU* aims to hold the hot update in the SLC-mode cache, for minimizing write operations on high density blocks (e.g. MLC in our tests). Our motivation of this design is because high density SSD blocks can endure a small

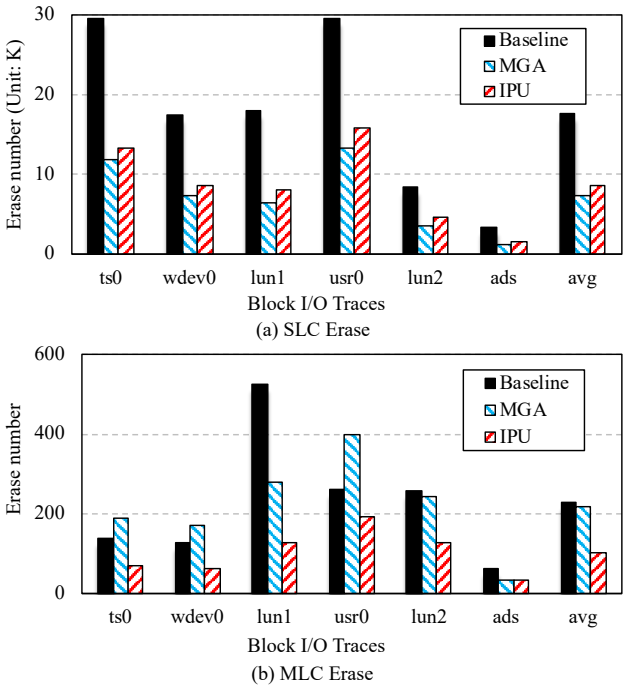

Figure 10: Erase number occurred in SLC and MLC blocks.

number of erase operations, but SLC-mode blocks can bear a large number of erases. More exactly, the endurance ratio between SLC and MLC is `10:1` [8], and this ratio comes to `100:1` or `1000:1` when the high density cell is TLC or Quad-Level Cell (QLC) [9]. In brief, our proposal can bring about better overall lifetime of high density SSDs.

4.4 Overhead Analysis

This section first analyzes memory overhead caused by mapping table and labeling block levels in the SLC-mode cache. Then, the computation overhead of GC policy is presented.

4.4.1 Memory Overhead. Partial programming demands additional memory space to hold the mapping table in a subpage granularity, and Figure 11 presents the comparison of normalized mapping table size. As seen, *MGA* expects the largest memory overhead that is `23.7%` more than that of *Baseline*. This is because *MGA* needs a two-level mapping table to record the subpage information, but *Baseline* employs page-level mapping scheme. On the other side, our proposal of *IPU* requires more memory space by `0.84%` on average, in contrast to *Baseline*. *IPU* leverages the same page to hold the different versions of same piece of data, so that it only records which part of subpage (i.e. offset in the page) corresponds to the latest version of data in the second level mapping.

Besides, labeling three-level blocks in the SLC-mode cache consumes memory space. It requires `820B` (`=2bit*5%` (SLC-mode ratio)`*65536` (block number)) in our context, taking a negligible amount of memory space in SSDs. To record the IS' values in our GC selection policy, it takes `819.2KB` (`=4B*5%*65536*64` (SLC

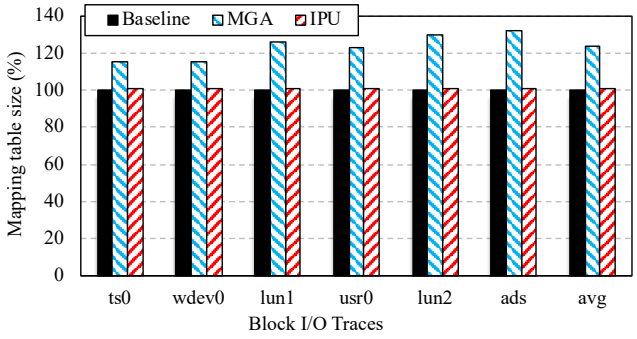

Figure 11: Normalized mapping table size after using selected comparison methods.

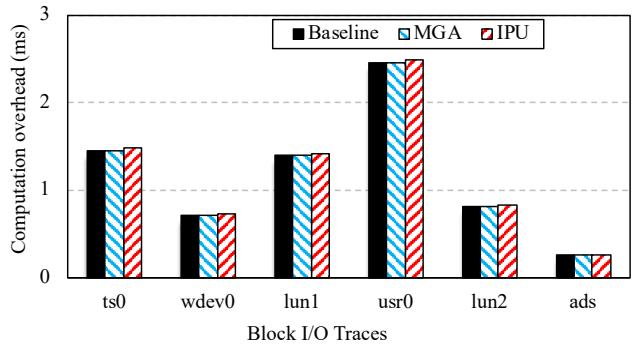

Figure 12: Computation overhead in GC processing after using selected comparison methods.

page number)), which results in an acceptable amount of memory space in SSDs.

4.4.2 Computational Overhead. The new GC policy results in computation overhead of traversing all the blocks in the target plane, whose computation overhead is similar to the greedy policy that employs in *Baseline*. Figure 12 shows the computation overhead when using GC policies of *Baseline* and *IPU*. The GC policy of *IPU* only expends more time by `1.2%`, in contrast to the GC policy of *Baseline*. That is, our GC policy needs less than `2.48ms` for searching the target GC blocks, which is acceptable.

4.5 Performance Case Study of P/E Cycles

This section analyzes the comparison of I/O latency and bit read error rate under four varied P/E Cycles. Figure 13 and 14 show the relevant results. As seen, both I/O latency and read error rate have the similar incremental tendency. This is because the SSD devices having the large number of P/E cycles have worse ability to fight against the bit error rate, and thus induce high I/O latency that needs more ECC decoding time. Nevertheless, our proposed method *IPU* unveils similar improvement on I/O latency and read error rate, compared with the related work *MGA*. This fact verifies the fine scalability of our proposal on varieties of SSD use stages.

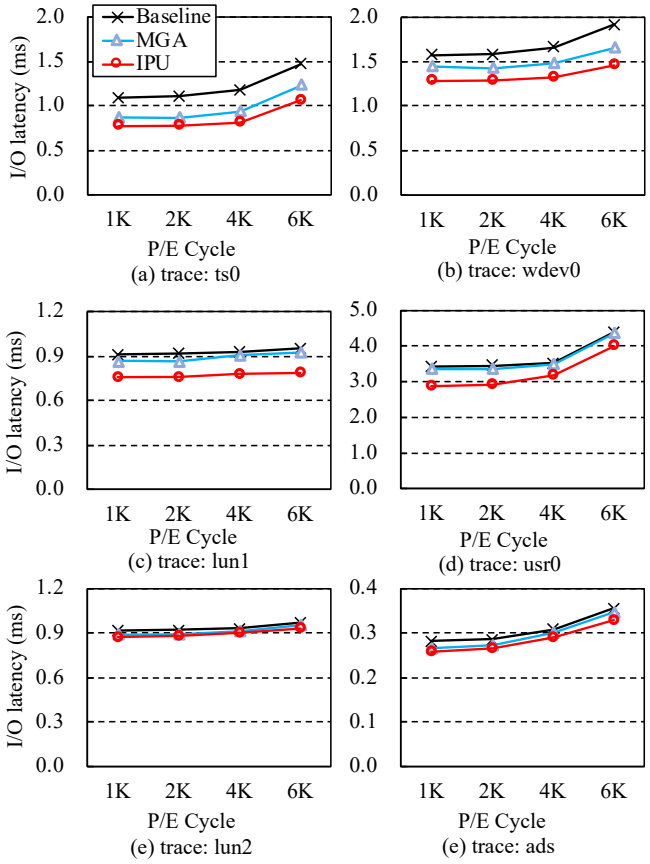

Figure 13: Analysis on I/O latency under varied P/E cycles after running the selected block I/O traces.

4.6 Summary

With respect to comparing the existing partial programming method, we emphasize the following two key observations. **First**, the proposed intra-page update with partial programming can minimize incremental bit errors induced by program disturb, which can contribute to less read latencies as the ECC correction time is reduced. **Second**, the proposed data movement policies in both SLC-mode cache management and GC processing can efficiently separate hot and cold update data, which can contribute to less write latencies as the hot update data are preferably buffered in the SLC-mode cache.

5 CONCLUSION

We have proposed and evaluated an intra-page update scheme with partial programming for the SLC-mode cache in high density SSDs that mitigates the negative impact caused by progressive partial programming. To this end, it flushes the updated data into free parts of the page that holds old version of data, and then utilizes three levels of blocks to shift the hot update data. Furthermore, it adopts a new GC policy, which selects the block having largest invalid subpage number, and migrates cold data to the low-level blocks till the data is not in the SLC-mode cache. Experimental results show that our proposal decreases I/O latency by between

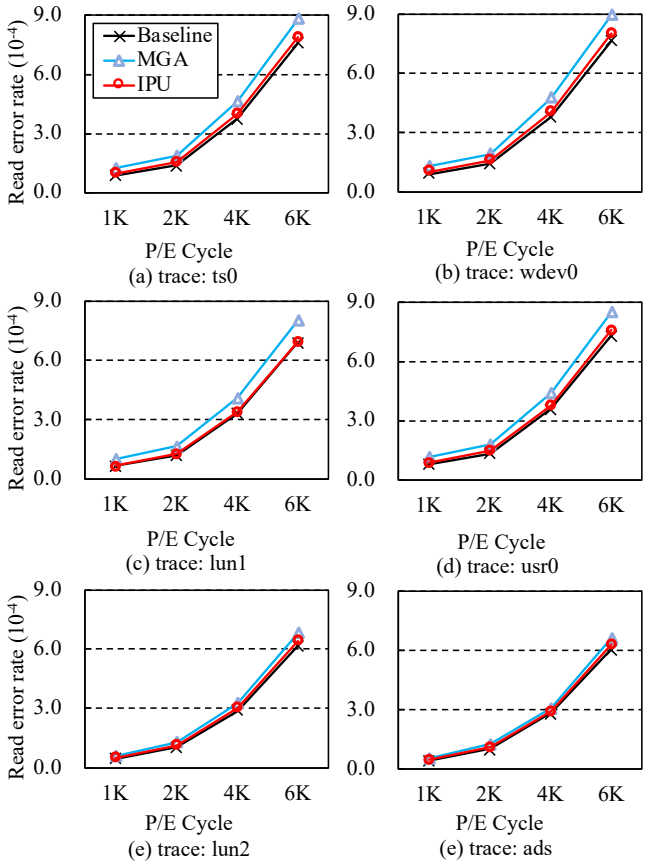

Figure 14: Analysis on bit error rate under varied P/E cycles after running the selected block I/O traces.

2.4% and 14.9%, and decreases read error rate by 9.2% on average, in contrast with state-of-the-art approaches. In this work, in order to eliminate the in-page disturb induced by partial programming, only the updated data utilizes the partial programming in the same page, leading to page utilization reduction. In the future, we will study improving the page utilization without a noticeable error increase, by adaptively combining infrequent data and saving them in the same page.

ACKNOWLEDGMENTS

This work was partially supported by "National Natural Science Foundation of China (No. 61872299, No. 62032019)", "Chongqing Talent (Youth, No. CQYC202005094)", "the Opening Project of State Key Laboratory for and Novel Software Technology (No. KFKT2021B06)", and "National Key Technologies R&D Program of China (No. 2018AAA0102102)".

REFERENCES

[1] Bryan S. Kim, Jongmoo Choi and Sang Lyul Min. Design tradeoffs for SSD reliability. In *USENIX Conference on File and Storage Technologies (FAST)*, 2019: 281–294.

[2] Lorenzo Zuolo, Cristian Zambelli, Rino Micheloni, and Piero Olivo. Solid-State Drives: Memory Driven Design Methodologies for Optimal Performance. In *Proceeding of the IEEE*, 2017: 105(9): 1589-1608. DOI: https://doi.org/10.1109/JPROC.2017.2733621

[3] Seiichi Aritome. NAND flash memory technologies. John Wiley & Sons, 2015.

[4] Yixin Luo, Saugata Ghose, Yu Cai, Erich F. Haratsch, and Onur Mutlu. Improving 3D NAND flash memory lifetime by tolerating early retention loss and process variation. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)*, 2018: 37:1-37:48. DOI: https://doi.org/10.1145/3224432

[5] Rino Micheloni. Solid-State Drive (SSD): a nonvolatile storage system. *Proceedings of the IEEE*, 2017, 105(4): 583-588. DOI: https://doi.org/10.1109/JPROC.2017.2678018

[6] Sangjin Yoo and Dongkun Shin. Reinforcement Learning-Based SLC Cache Technique for Enhancing SSD Write Performance. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2020.

[7] Mengying Zhao, Lei Jiang, Youtao Zhang, and Chun Jason Xue. SLC-enabled Wear Leveling for MLC PCM Considering Process Variation. In *Annual Design Automation Conference (DAC)*, 2014: 1-6. DOI: https://doi.org/10.1145/2593069.2593217

[8] Duo Liu, Lei Yao, Linbo Long, Zili Shao, and Yong Guan. A workload-aware flash translation layer enhancing performance and lifespan of TLC/SLC dual-mode flash memory in embedded systems. *Microprocessors and Microsystems*, 2017, 52: 343-354. DOI: https://doi.org/10.1016/j.micpro.2016.12.009

[9] Ahmed Izzat Alsalibi, Sparsh Mittal, Mohammed Azmi Al-Betar, and Putra Bin Sumari. A survey of techniques for architecting SLC/MLC/TLC hybrid Flash memory–based SSDs. *Concurrency and Computation: Practice and Experience*. 2018, 30(13): e4420. DOI: https://doi.org/10.1002/cpe.4420

[10] Jung-Hoon Kim, Sang-Hoon Kim, and Jin-Soo Kim. Subpage programming for extending the lifetime of NAND flash memory. In *Design, Automation and Test in Europe (DATE)*, 2015: 555-560.

[11] Micron 4Gb, 8Gb, and 16Gb x8 NAND Flash Memory Features, 2006. https://datasheetspdf.com/pdf/697309/Micron/MT29F8G08BAA/1.

[12] Yazhi Feng, Dan Feng, Chenye Yu, Wei Tong, and Jingning Liu. Mapping granularity adaptive ftl based on flash page re-programming. In *Design, Automation and Test in Europe (DATE)*, 2017. DOI: https://doi.org/10.23919/DATE.2017.7927019

[13] Chun-Yi Liu, Jagadish B. Kotra, Myoungsoo Jung, Mahmut T. Kandemir, and Chita R. Das. SOML Read: Rethinking the Read Operation Granularity of 3D NAND SSDs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019: 955-969. DOI: https://doi.org/10.1145/3297858.3304035

[14] Tseng-Yi Chen, Yuan-Hao Chang, Chien-Chung Ho, and Shuo-Han Chen: Enabling sub-blocks erase management to boost the performance of 3D NAND flash memory. In *ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016: 92:1-92:6. DOI: https://doi.org/10.1145/2897937.2898018

[15] Chun-Yi Liu, Jagadish Kotra, Myoungsoo Jung, and Mahmut T. Kandemir. PEN: Design and Evaluation of Partial-Erase for 3D NAND-Based High Density SSDs. In *USENIX Conference on File and Storage Technologies (FAST)*, 2018: 67-82

[16] Sheng Qiu and A. L. Narasimha Reddy. A hybrid file system for improving random write in nand-flash SSD. In *IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2013. DOI: https://doi.org/10.1109/MSST.2013.6558434

[17] Myungsuk Kim, Jaehoon Lee, Sungjin Lee, Jisung Park, and Jihong Kim. Improving performance and lifetime of large-page NAND storages using erase-free subpage programming. In *ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2017: 1-6. DOI: https://doi.org/10.1145/3061639.3062264

[18] Samsung K9F2G08U0C, 2010. https://datasheetspdf.com/datasheet/K9F2G08U0C.html.

[19] Xuebin Zhang and Jiangpeng Li and Hao Wang and Kai Zhao and Tong Zhang. Reducing solid-state storage device write stress through opportunistic in-place delta compression. In *USENIX Conference on File and Storage Technologies (FAST)*, 2016: 111-124.

[20] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage*, 2008, 4(3): 1-23. DOI: https://doi.org/10.1145/1416944.1416949

[21] Microsoft Production Server Traces. Retrieved from http://iotta.snia.org/traces/158.

[22] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In *ACM International Systems and Storage Conference (SYSTOR)*, 2017: 1-11. DOI: https://doi.org/10.1145/3078468.3078479

[23] Zujie Ren, Biao Xu, Weisong Shi, Yongjian Ren, Feng Cao, Jiangbin Lin, and Zheng Ye. iGen: A Realistic Request Generator for Cloud File Systems Benchmarking. In *IEEE International Conference on Cloud Computing (CLOUD)*, 2016: 343-350. DOI: https://10.1109/CLOUD.2016.0053

[24] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, Chao Ren. Exploring and exploiting the multilevel parallelism inside SSDs for improved performance and endurance. *IEEE Transactions on Computers*, 2013, 62(6): 1141-1155. DOI: https://doi.org/10.1109/TC.2012.60

[25] Congming Gao, Min Ye, Qiao Li, Chun Jason Xue, Youtao Zhang, Liang Shi, and Jun Yang. Constructing large, durable and fast ssd system via reprogramming 3D TLC flash memory. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019: 493-505. DOI: https://doi.org/10.1145/3352460.3358323

[26] R. Micheloni, R. Ravasio, A. Marelli, E. Alice, V. Altieri, A. Bovino, L. Crippa, E. Di Martino, L. D'Onofrio, A. Gambardella, E. Grillea, G. Guerra, D. Kim, C. Missiroli, I. Motta, A. Prisco, G. Ragone, M. Romano, M. Sangalli, P. Sauro, M. Scotti, and S. Won. A 4Gb 2b/cell NAND flash memory with embedded 5b BCH ECC for 36MB/s system read throughput. In *IEEE International Solid State Circuits Conference (ISSCC)*, 2006: 497-506. DOI: https://doi.org/10.1109/ISSCC.2006.1696082