Invalid Data-Aware Coding to Enhance the Read Performance of High-Density Flash Memories

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

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

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

***Abstract*—High bit-density flash memories such as Multi-Level Cell (MLC) and Triple-Level Cell (TLC) flash have become a norm, since doubling the cell bit-density can increase the storage capacity by 2× using the same number of cells. However, these high bit-density flash memories suffer from the read variation problem – e.g., the three kinds of bits (i.e., lower, middle, and upper bits) in a TLC cell have different read latencies; reading an upper bit takes a longer time than reading a middle bit, and reading a lower bit takes the minimum time. In this paper, we note that, in the conventional coding, the long read latencies of the middle and upper bits are not reduced even after the lower bit value is invalidated (i.e., no longer used). Motivated by this problem with the traditional coding, we propose a new coding technique, called Invalid Data-Aware (IDA) coding, which reduces the upper and middle bit read latencies close to the lower bit read latency when the lower bit becomes invalid. The main strategy the IDA coding employs is to merge the duplicated voltage states coming from the bit invalidation and reduce the number of (read) trials to identify the voltage state of a cell. To hide the performance and reliability degradation caused by the application of the IDA coding, we also propose to implement it as a part of the data refresh function, which is a fundamental operation in modern SSDs to keep its data safer and longer. With an extensive analysis of a TLC-based SSD using a variety of read-intensive workloads, we report that our IDA coding improves the read response times by 28%, on average; it is also quite effective in devices with different bit densities and timing parameters.**

I. Introduction

As NAND flash technology has become mature, Solid State Drives (SSDs) based on it have rapidly started to be pervasive in various computing domains ranging from personal electronics to enterprise infrastructures. The sharp increase in the bit-per-cost of NAND flash (and SSD) is one of the most critical elements of its success. Increasing storage capacity can be realized by (i) decreasing the feature sizes (e.g., up to below ten nm), (ii) stacking a planar memory array towards 3D architecture (e.g., up to several tens of layers), or (iii) increasing the bit density of each memory cell (e.g., up to four bits-per-cell). Especially, the latter strategy is indispensable, since the storage capacity can increase two-fold as the number of bits per cell doubles under a given number of memory cells (i.e., when none of the physical parameters such as the feature size, the horizontal memory array size, and the number of vertical layers changes).

As a result, a majority of modern flash memories today have high bit-density for each cell. Compared to Single-Level Cell (SLC) flash that allows each cell to maintain a 1-bit data, Multi-Level Cell (MLC) flash can store 2-bit information in each cell and Triple-Level Cell (TLC) flash is able to accommodate 3-bit data in a cell. Note that SSDs using these high bit-density flash (MLC or TLC) are the mainstream now, due to their cost benefits; furthermore, a few vendors [1] have recently debut even Quad-Level Cell (QLC) flash – 4 bits capacity for each cell – in the market.

Unfortunately, these high bit-density flash devices have what can be called a *read performance variation problem:* the read latencies of the different bits in a cell are asymmetric. For example, TLC flash provides three types of bits for each cell, Least-Significant-Bit (LSB), Center-Significant-Bit (CSB), and Most-Significant-Bit (MSB); and their read latencies are all *different*. Specifically, reading CSB takes a longer time than reading LSB, but, shorter than reading MSB, and in fact, the MSB read latency is 2× (or more) longer than the LSB read latency [2], [3]. This is because reading the three bits needs a different number of memory accesses in the conventional TLC coding. For example, in the most widely-used TLC coding, the number of memory accesses to read LSB, CSB, and MSB bits are 1, 2, and 4, respectively.

The presence of longer latencies (e.g., for the CSB or MSB reads in a TLC flash) against shorter latencies (e.g., for the LSB read) in a high bit-density flash can prevent one from maximizing the storage read performance for read-intensive applications. Also, the variation in flash read latencies can have a negative impact on the service of time-critical applications. It should be noted that high bit-density flash exhibits non-uniformity in write latencies as well, which has received a lot of attention in the past [4]–[6]. Specifically, for write-intensive applications (or applications whose write I/Os are not a few), it is critical to optimize write I/Os, since (i) flash write latencies are generally 10× longer than flash read latencies and (ii) garbage collection (GC) caused by write I/Os imposes a significant performance burden. On the other hand, our focus in this paper is on *read-dominant workloads* where enhancing flash read latencies is critical in the SSD performance.

In this paper, we point out a significant problem with the conventional flash coding mechanism: for example, in a TLC flash, even after the LSB value becomes invalid, reading the CSB or MSB continues to access the memory array multiple times (i.e., 2 or 4). *Why can't one reduce the number of memory accesses (and the latencies) for CSB and MSB reads, when the value of the corresponding LSB is not needed (invalidated)?* Motivated by this question, we present a new coding scheme, called **Invalid Data-Aware (IDA)** coding, which brings the long MSB and CSB read latencies close to the short LSB read latency, when the associated LSB is invalid. Note that while we use TLC as our baseline, our scheme can be generalized to any high bit-density flash such as MLC and QLC. In this new approach, one begins with the traditional

978-1-5386-6240-3/18/$31.00 ©2018 IEEE
DOI 10.1109/MICRO.2018.00046

coding to write data into a new block; after multiple page data in the block are invalidated, one can apply our IDA coding to that block and the reprogrammed block provides enhanced read latencies. The principle mechanism behind our proposed coding technique is *voltage adjustment*, that is, merging the duplicated voltage states (coming from the invalidation of the associated bits) and reducing the number of memory accesses needed to read the remaining bits.

Applying the IDA coding to a block might raise performance and reliability issues. That is, reprogramming a cell with the IDA coding needs to adjust the voltage level of the cell, which (i) consumes storage-internal bandwidth and (ii) can generate errors in the neighboring cells. To effectively hide such overheads, we propose to implement our IDA coding as part of the *data refresh*, which is a regularly executed flash function to safely maintain the data for a long time by (1) reading the data from a target block, (2) correcting errors therein using Error Correcting Code (ECC), and (3) writing it into a new block. Specifically, the time needed for adjusting the voltage levels in the target block can replace a part of step (3) in the data refresh; so, one can avoid the potential performance overhead. Also, since we secure the error-free data from steps (1) and (2), one can be free from data loss, as a result of the voltage adjustment. Note that data refresh (also known as *data scrub*) is a fundamental operation in modern SSDs, as evidenced by [7]–[9].

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

- We propose a new flash coding technique, called Invalid Data-Aware (IDA) coding, which brings the MSB or CSB read latencies close to the LSB read latencies, when the associated LSB or CSB is invalid. By changing the voltage level of a cell, our IDA coding limits the number of memory accesses to only one or two for an MSB read; and only one for a CSB read.
- Applying our IDA coding requires voltage adjustment in target blocks, which can drop the storage throughput as it occupies flash resources and affect the data integrity by interfering with neighboring data. To address this potential problem, we propose to leverage the (existing) data refresh operation; implementing the voltage adjustment procedure as a part of the data refresh can allow us to effectively hide the potential performance and reliability overheads.
- Our extensive experimental evaluations based on TLC flash reveal that the proposed IDA coding can improve the read response times by 28% over the conventional coding. Also, a detailed analysis on the data refresh confirms that the overheads (both the performance and reliability) brought by our IDA coding are successfully hidden. Furthermore, our sensitivity experiments indicate that the IDA coding is also effective for MLC devices (an improvement of 14.5%) and devices with various memory timing parameters.

II. Preliminaries

A. SSD and Flash Memory

Figure 1 illustrates the major components of an SSD and flash organization, which are relevant to our proposal.

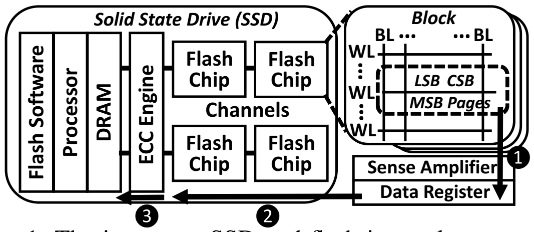


Fig. 1: The important SSD and flash internal components.

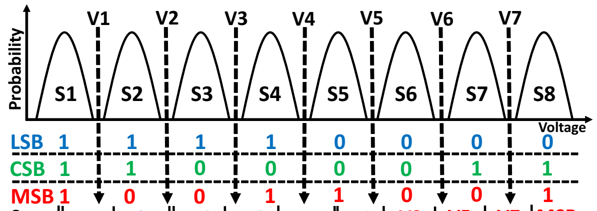
- **Flash Software:** This provides various functionalities to manage the underlying flash memories and the traditional storage interface to the host; it is called flash translation layer (FTL). It uses an SSD-internal processor and DRAM to execute its operations and maintain its data structures. Among various tasks of FTL, the *data refresh* is worthwhile to note; it relocates the data from one place to another in an attempt to prevent the data from being corrupted.
- **ECC Engine:** As the data stored in a flash memory can be polluted by various error sources, this engine plays an essential role of providing high data reliability. Using the ECC, data are encoded before being written into the memory, and when the data is read, it is decoded and its erroneous parts are corrected.
- **Channel:** This is a DDR bus [10] that connects multiple flash chips to the DRAM.
- **Flash Chip:** A flash chip includes thousands of blocks (a block is the erase unit). Each block is constructed as a memory array where up to hundreds of wordlines (WLs) and up to tens of thousands of bitlines (BLs) form cells. In a TLC flash where a cell can express 3-bit information, each WL provides three logical pages, namely, the LSB, CSB, and MSB pages. Note that a page is the basic read and write unit. The blocks share a data register which acts as a buffer for the page data read from a WL or to be written into a WL.

B. Conventional TLC Coding

A TLC cell can express its three bit values using the voltage level of the cell. Figure 2 illustrates the most widely-used TLC coding which provides 8 different voltage ranges (or states), each representing a combination of 3 different bit values. Using this conventional coding, programming (writing) three bit values into a cell forms a voltage level (called the threshold voltage) in one of the eight voltage ranges; so, if a cell is in the state S4, its LSB, CSB, and MSB bit values are "1", "0", and "1", respectively. Forming the desired threshold voltage in a cell can be achieved by the Incremental Step Pulse Programming (ISPP) [11], which repeatedly injects electrons to the cell until the threshold voltage of the cell reaches the desired voltage level. To write a new set of three bit values into a cell, the cell should be *erased* first, after which the voltage state of the cell becomes S1. Reading the three bits (identifying the threshold voltage of the cell) involves sensing the cell with some of read voltages (V1 to V7), which will be discussed in detail in the following section.

C. Flash Read Mechanism

Our interest in this paper is to enhance flash *read latencies*; here, we discuss the entire set of stages a page read goes



State	V4	LSB	V2	V6	CSB	V1	V3	V5	V7	MSB
S1	*on*	1	*on*	*on*	1	*on*	*on*	*on*	*on*	1
S2	*on*	1	*on*	*on*	1	*off*	*on*	*on*	*on*	0
S3	*on*	1	*off*	*on*	0	*off*	*on*	*on*	*on*	0
S4	*on*	1	*off*	*on*	0	*off*	*off*	*on*	*on*	1
S5	*off*	0	*off*	*on*	0	*off*	*off*	*on*	*on*	1
S6	*off*	0	*off*	*on*	0	*off*	*off*	*off*	*on*	0
S7	*off*	0	*off*	*off*	1	*off*	*off*	*off*	*on*	0
S8	*off*	0	*off*	*off*	1	*off*	*off*	*off*	*off*	1

Fig. 2: The conventional TLC coding with eight voltage states, each representing a combination of the LSB, CSB, and MSB bit values. Reading each bit needs memory sensing with a different set of read voltages.

through and focus in particular on the memory access stage.

Stages of a Read: The data path from a flash memory array to the SSD-internal DRAM can be divided into three stages, as also shown in Figure 1 (the interface between the SSD-internal DRAM and the host is outside our interest, as it is an independent module from flash memory). Given a page read request, ❶ the page data is read from the memory array by sensing the target WL and is temporarily stored in the data register, which is referred to as the memory access stage. Then, ❷ the page data is transferred out of the flash chip via the connected channel; this is called the data movement stage. Finally, ❸ the page data is decoded by the ECC engine to correct the erroneous bits in it (we call this the ECC decoding stage). To date, a range of efforts to improve the read latency have mainly targeted the data movement ❷ and ECC decoding ❸ stages. For the fast data movement, vendors increase the channel data rates and/or the number of channel pinouts [12], [13]. For example, a flash chip with NVDDR2 bus (333MT/s) and 8 pinouts takes at most 50us to transfer a 8KB page data. And, to improve the ECC decoding process, vendors employ highly-parallelized hardware ECC modules [14], [15]. For instance, an ultra-throughput (1000MB/s) ECC is able to decode a 8KB page data, whose raw bit error rate is 0.004, in at most 20us. In contrast, few works have paid attention to improve ❶ the memory access stage, even though this stage can be a performance bottleneck in *flash read*, depending on the type of the page being accessed. In our work, based on a real TLC device [16], we assume that the time taken by the memory access stage can range between 50us and 150us. Motivated by this, in this paper, we try to optimize this stage with a new coding technique to reduce read latencies.

Memory Access Stage: The latency taken by this stage is determined by the number of WL accesses. Specifically, the identification of the threshold voltage of a cell can be realized by applying one or more read voltages to the WL and each time by sensing whether the cell is on or off. But, the number of read voltages to apply to the WL varies depending on the page type, which leads to an "asymmetry" across the read latencies of high bit-density flash memory. In a TLC using the conventional coding shown in Figure 2, the mechanisms used for the LSB, CSB, and MSB reads are as follows:

- **LSB Read:** This needs to apply the read voltage *V4* only one time to the WL. If the cell is switched on, the LSB value is "1"; otherwise, it is "0".
- **CSB Read:** This uses two read voltages, *V2* and *V6*; so, the number of memory accesses is two. If both of the accesses switch the cell on or off, the CSB value is "1". If one is on and the other is off, the CSB value is "0".
- **MSB Read:** This needs a total of four memory accesses with four different read voltages, *V1*, *V3*, *V5*, and *V7*. If the cell is on or off for all the four accesses, the MSB value is "1"; it is also "1", when two are on and the other two are off. For all the other cases, the MSB value is "0".

III. Invalid Data-Aware Coding

Even though the conventional TLC coding technique (illustrated in Figure 2) can provide an efficient representation of a three bit information in a cell, it also increases the read latencies of CSB and MSB, compared to LSB. What is worse, in this coding, once a data is stored in the CSB or MSB of a cell, its long read latency does not change until the cell is erased or the data is relocated. In this paper, we pay special attention to scenarios where the associated LSB or CSB data is invalidated, but the cell is *not* yet erased; unfortunately, the valid CSB or MSB data continue to exhibit long read latencies. Thus, we attempt to reduce the read latencies of the CSB or MSB data by applying a new coding (called invalid data-aware coding) in such scenarios. In the next three subsections, we introduce our proposed coding technique by answering the following three questions:

1) Why do CSB/MSB reads need to access the memory multiple times, even when its associated LSB/CSB data are invalid? What is the fraction of such unnecessary memory accesses in real applications? (Section III-A)
2) How can we reduce the number of memory accesses for CSB/MSB reads, after its associated LSB/CSB data become invalid? What is the design concept behind our invalid data-aware coding? (Section III-B).
3) How can we implement the invalid data-aware coding with negligible overheads? (Section III-C)

A. Motivation: Unnecessary Memory Accesses

Motivational Question: As an application keeps executing, its stored data can be updated (i.e., the stored data is marked as invalid and the updated data is written into a new cell). As a result, one, two, or all of the LSB, CSB, and MSB bits programmed in a cell can become invalid. If only the MSB bit is invalid, the still-valid LSB and CSB bits can be read with one and two memory accesses, respectively, as usual (i.e., the case where all the three bits are valid). However, we pay attention to other scenarios; for example, only the LSB bit is invalid while the CSB and MSB bits are still valid. In such a scenario, we question whether reading the CSB and MSB

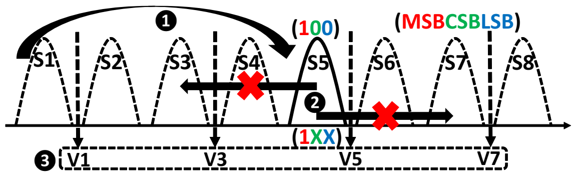


Fig. 3: An example scenario where (1) "100" is written and (2) then the LSB and CSB bits are invalidated. In the conventional coding, (3) the invalidation of the associated bits does not change the procedure followed for reading the other bits.

bits still requires two and four memory accesses, respectively. Our goal in this paper is to reduce the number of memory accesses for the CSB and MSB reads and in turn reduce their read latencies in such cases.

Limitation of Conventional Coding: In the conventional coding techniques, even if the LSB bit becomes invalid, the number of memory accesses to read CSB and MSB bits cannot be reduced. The reasons can be explained as follows:

• The threshold voltage of a cell is determined when the data is written into the cell. For example, ❶ if one writes "0" as LSB, "0" as CSB, and "1" as MSB into a cell, the threshold voltage of the cell becomes S5 (Figure 3).

• Even though some or all of the data are invalidated, the threshold voltage of the cell does not change, until the cell is erased (here, we consider the general error-free case where an error source does not affect the threshold voltage). In the above scenario, ❷ the programmed voltage state (S5) does not move, even though both the LSB and CSB become unavailable.

• The voltage state of a cell is not known before and after the data invalidation; so, it should be detected using the predefined memory access procedures for the LSB, CSB, and MSB reads. In the example (Figure 3), being unaware of the voltage state, ❸ one needs to access the memory four times using V1, V3, V5, and V7 to read the MSB bit.

Opportunities in Applications: To investigate how often a flash memory encounters the cases where LSB or CSB page is invalid but CSB or MSB page is still valid, we conducted an experiment (our detailed setup/configuration is given in Section IV). Figure 4 shows the distribution of reads across different page types and scenarios (whether the associated pages are valid or not), when executing different applications. Roughly speaking, all the read accesses can be divided into three groups: LSB, CSB, and MSB reads. In more detail, the CSB reads can be further classified into two groups, depending on whether the associated LSB page is valid or not; also, the MSB reads can be further categorized into two groups, based on whether both the associated LSB and CSB pages are valid or not. We now make the following important observations:

• The fraction of reads targeting the CSB or MSB pages is not small. Specifically, one third of total reads is for CSB page, and another one third of total reads targets MSB pages. Note that our interest is to reduce the CSB and MSB latencies (the number of memory accesses); the LSB read has no scope for optimization, as it originally needs only one memory access.

• For an average of 30% of all MSB reads, when the MSB

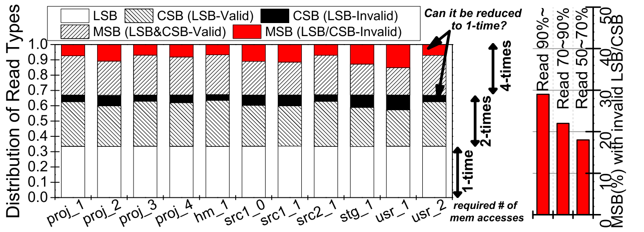


Fig. 4: Distribution of all read accesses across different page types in the tested 11 workloads (left): The LSB, CSB, and MSB reads are almost evenly distributed. On average, 18% of the CSB reads occur when the associated LSB is invalid, and 30% of the MSB reads occur when the associated LSB or/and CSB are invalid. For additional 9 workloads (right), categorized by their read ratios, the fraction of MSB reads where the associated LSB/CSB are invalid is not small.

page is read, the associated LSB or/and CSB pages are invalid. Our goal is to attack this group of reads and reduce the number of memory accesses from the original four to two or one.

• For an average of 18% of all CSB reads, when the CSB page is read, the associated LSB page is invalid. We are also interested in this group of reads; and we attempt to reduce the number of memory accesses from two to one.

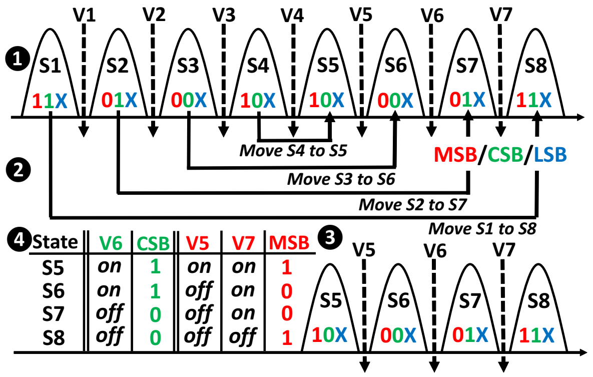
Overall, *the scenarios where we are interested in are quite common in a wide range of read-intensive applications*; in Figure 4 (right), we profiled 9 more workloads from [17], [18] and confirmed that the beneficial scenarios are abundant. (note that write-dominant workloads are out of our interest, where enhancing writes is more critical than reads for overall performance). Accordingly, one would reduce read latencies significantly, if the number of memory accesses for CSB and MSB reads could be reduced in the aforementioned scenarios.

B. Our Proposal: a New Coding Technique

In an attempt to enhance the CSB or MSB read latencies when the associated LSB is invalid, we propose a new coding technique, called *invalid data-aware* (IDA) coding. In our approach, one starts with the conventional coding to write data into cells; later, if the LSB or CSB bit of the cells becomes invalid, one can apply our IDA coding to the cells, which adjusts their threshold voltage levels. With the newly-programmed cells, one can read the CSB bit or MSB bit with fewer memory accesses than before.

Design Principles: Figure 5 uses an example scenario to illustrate how our IDA coding can be applied and how it can reduce the number of memory accesses. The design principles behind our IDA coding can be summarized as follows:

• In a cell programmed with the conventional coding, once its LSB or/and CSB bits are invalidated, different voltage states can represent the same bit values. Let us assume that the LSB bit becomes invalid while the CSB and MSB bits are still valid. ❶ The two voltage states S1 and S8 exhibit the same data value ("11X"), which indicates "1" for the MSB and "1" for the CSB; however, the LSB value is no longer used. Similarly, the voltage states S2 and S7, S3 and S6, and S4 and S5 have the same data values ("01X", "00X", and "10X", respectively).



State	V6	CSB	V5	V7	MSB
S5	on	1	on	on	1
S6	on	1	off	on	0
S7	off	0	off	on	0
S8	off	0	off	off	1

Fig. 5: An example scenario of the conventional coding where (1) the LSB bit is invalidated. (2) By merging two different voltage states with the same bit values, (3) our IDA coding reduces the number of memory accesses from 4 to 2 for reading MSB, and from 2 to 1 for reading CSB.

• Motivated by the duplicated voltage states after the invalidation of the associated bits, we propose to merge them and set the threshold voltage of the cell based on our IDA coding. In the above scenario, ❷ if the voltage state of the cell is S1, S2, S3, or S4, it can be moved to S8, S7, S6, or S5, respectively. If the voltage state is one of S5, S6, S7, or S8, it does not need to change. The relocation of the voltage states can be done through Incremental Step Pulse Programming (ISPP) [11], which is used for normal flash programming. Using ISPP, the electrons can be injected to the cell until the intended voltage level is achieved; thus, the voltage states can be moved to the right direction only. We call this part of applying our IDA coding *voltage adjustment*.

• As a result of the voltage adjustment, ❸ all the cells reprogrammed by the IDA coding will have one of the four voltage states (S5, S6, S7, and S8), instead of the original eight states provided by the conventional coding (it looks like the voltage distribution of an MLC cell where its two bits can be read with one or two memory accesses). In a reprogrammed cell, the read voltage levels should be adjusted as well; in this example, only V5, V6, and V7 are used to read the CSB and MSB bits. Now, ❹ reading the CSB bit can be done by accessing the memory with V6 only one time. Also, the MSB bit can be read by accessing the memory with V5 and V7.

By applying our IDA coding to the cells where the LSB bit is invalid, the number of memory accesses for the CSB and MSB reads can be reduced from two to one and from four to two, respectively. This, in turn, can lead to significant improvements in the TLC read latencies.

A few flash vendors employ a different TLC coding, which needs two, three, and two memory accesses for LSB, CSB, and MSB reads, respectively; here, the read performance variation is much less. However, in a higher density flash like QLC, such a TLC coding can also suffer from the read variation problem. (Figure 6 gives an example of applying our IDA coding to a QLC flash, which can bring a significant benefit). Note that our IDA coding is general, which can be combined with any coding scheme in any high bit density flash.

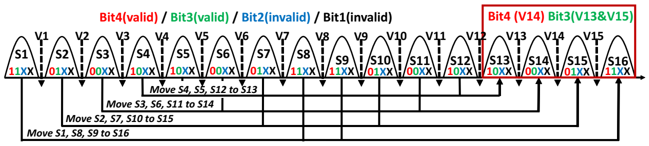


Fig. 6: An example scenario of QLC [19] with 4 bit data and 16 voltage stages, where the two lower bits, Bits 1 and 2, are invalidated. Our IDA coding reduces the number of memory accesses for Bits 4 and 3 from 8 and 4 to 2 and 1, respectively.

Block-level IDA Coding: Our IDA coding can be applied at the block granularity; as a result, there can be two types of blocks in an SSD: blocks programmed by the conventional coding and blocks reprogrammed by the IDA coding. When data are written into a clean (erased) block, the block always employs the conventional coding, which is called the "conventional block". After a certain period of time, some pages in the conventional block can become invalid; then, if beneficial, we re-program each WL of the block by applying the voltage adjustment, which is referred to as the "IDA block". Note that the voltage adjustment does not erase the block until two SSD operations, garbage collection and data refresh, reclaim the IDA blocks and make them clean again.

• The garbage collection (GC) selects a victim block with the least number of valid pages among all used blocks; all valid pages in the victim block are read and written into a clean (conventional) block. In our design, the victim block can be either a conventional block or an IDA block. Note however that, our interest is in *read-intensive applications* where writes are not frequent, and consequently GC invocations are rare.

• The data refresh, on the other hand, is regularly invoked in our read-dominant environment. For every 3 days to 3 months (see our experimental configuration in Section IV), all the valid pages in the target block are read and written into a clean (conventional) block, after which the target block is erased by GC later. In our design, the target block can also be a conventional block or an IDA block.

Voltage Adjustment Feasibility: The voltage adjustment is a process of reprogramming NAND blocks using the existing ISPP controller, which is feasible in modern flash chips.

• The ISPP controller is a programmable logic that repeats to charge cells with the input-voltage till reaching the input-threshold. Actually, given a normal write command composed of input-voltage and input-threshold, the ISPP controller forms the threshold voltage, based on the conventional coding. In a similar fashion, our IDA coding has new commands consisting of predefined input-voltage and input threshold values, which can be programmed at design time.

• The voltage adjustment time is close to a half of the longest page write time (i.e., MSB page write latency). Specifically, the time taken by a voltage adjustment is proportional to the number of ISPP loop iterations, which is roughly determined by the voltage range the ISPP controller [11] has to explore. We can halve the voltage range by first placing the voltage states S1, S2, S3, and S4 to S5 (as the possible voltage states of an IDA-coded cell are S5, S6, S7, or S8), whereas the voltage range for an MSB page write is from S1 to S8. Note also that the voltage adjustment is performed once for each WL that includes three different pages (LSB, CSB, and MSB). In our evaluation, we conservatively set the voltage adjustment latency to the MSB write latency.

• Due to the cell interference, the voltage adjustment process can increase the number of erroneous bits in the block. However, our IDA coding is free from any data loss, since we keep error-free pages before performing the voltage adjustment; so, once a page becomes erroneous, we can discard it and use the error-free one (Section III-C will describe this strategy). Furthermore, the impact of the voltage adjustment is much less than that of direct page reprogramming such as [20]–[22] in two aspects. First, the voltage adjustment for a WL needs much fewer loop iterations than programming three pages in it, which in turn stresses each WL less. Second, there are many invalid pages and WLs in the neighbors, which have nothing to do with errors (see cases 4, 6 to 8 of Table I).

Flash Endurance Implication: Our IDA coding that forms the threshold voltage at higher voltage levels does not hurt the flash endurance. Actually, flash vendors provide an erase cycle limit for the lifetime of a flash chip, after which the integrity of the written data is not guaranteed. The advertised erase cycle limit for a flash chip is derived from the worst-case scenarios – e.g., programming cells with the highest voltage level and erasing them. Keeping this in mind, our IDA coding maximizes the cell utilization in each erase cycle, while preventing the experienced erase cycles from increasing.

C. Implementation: Exploiting Data Refresh

Our IDA coding should not bring a loss in data reliability or cause significant performance overheads. Keeping this in mind, below, we present a *light-weight* implementation of the proposed coding technique.

Two Problems with Voltage Adjustment: Adjusting the threshold voltage of a WL can potentially bring two problems:

• The voltage adjustment of a WL can have a write latency, which entails a high overhead in read-dominant applications. Note that the voltage states of cells in a WL can be adjusted by the ISPP mechanism, which is similar to writing a page data into a WL. Also, if the pages re-programmed by the IDA coding would be read only a few times more and invalidated soon, the new coding would degrade the overall performance.

• Since the ISPP mechanism repeatedly applies high voltage to the WL, the voltage adjustment of a WL can generate cell interference to the neighboring WLs, which can corrupt the page data stored in the WLs. Even though the new coding improves the performance, data loss cannot be allowed.

Motivation from Data Refresh: The data refresh [23] is the fundamental operation in read-intensive applications, since it prevents the stored data from being lost and extends data retention times. In implementing our IDA coding, we are motivated by the two aspects of data refresh.

• If the IDA coding is applied to some of the valid pages to be moved by the data refresh, one can save (avoid) the writes of those pages to new blocks; that is, for those pages, the writing times of the data refresh can be replaced by the voltage adjustment times of the IDA coding. In addition, instead of selecting arbitrary pages, a choice of the valid pages in the refresh target blocks as the target pages of the IDA coding can avoid the potential high overhead brought by the voltage adjustment for the pages to be invalidated soon. Note that the refresh target blocks include many valid page data that might be read more and more in the future, as they are not invalidated during the long refresh period.

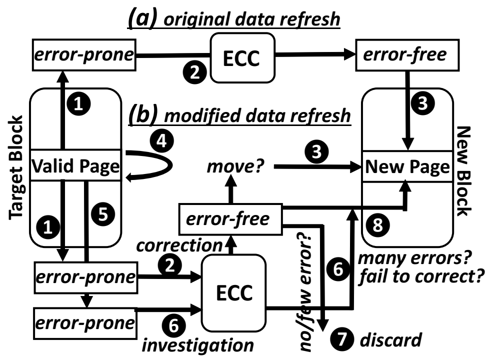


Fig. 7: (a) The original data refresh that migrates valid pages from a target block to a new block, and (b) the modified data refresh that accommodates our IDA coding; some pages are migrated while others are reprogrammed by our IDA coding. For the corrupted pages due to the voltage adjustment, the identical (error-free) pages are written into a new block.

• One does not need to worry about the data loss during the voltage adjustment, since the error-free data are maintained during the data refresh. Note that a refresh operation reads *all* valid pages to the memory-outside, corrects their errors using the ECC, and writes them to the memory. Once the pages become erroneous in a block where the IDA coding is applied, those (error-free) pages can be written into a new block.

Implementing IDA Coding on top of Data Refresh: Taking an advantage of the performance and reliability benefits coming from the data refresh, we propose to implement our IDA coding as a part of the data refresh operation. Figure 7 illustrates the flow of the conventional data refresh and that of our IDA coding-modified data refresh. In the original data refresh (Figure 7a), ❶ all valid pages in the target block are read, ❷ all the pages are decoded by the ECC to correct their errors, and ❸ the error-free pages are written into a new block. Our modified data refresh (Figure 7b) also begins with ❶ reading all valid pages and ❷ decoding them using the ECC as in the original process. Among all the error-free valid pages, ❸ the selected pages that will be reprogrammed by the IDA coding are kept while all other pages that cannot get benefit from the IDA coding are written into a new block. Then, ❹ the threshold voltages of pages (WLs) in the target block are adjusted to apply the IDA coding. Once the voltage adjustment for the WLs is done, ❺ all the valid pages in the block are read to examine whether they are corrupted or not (due to the voltage adjustment of the neighboring WLs). ❻ Those read pages are decoded by ECC, which can lead to two possible results: (i) the page is error-free or includes small errors and (ii) the page has many errors or fails to be decoded. Depending on the decoding result, (i) ❼ the error-free version of the page (kept in ❸) is discarded as it can stay in the target block, whereas (ii) ❽ the error-free version of the page is written into the new block as it is polluted by the voltage adjustment of the neighboring WLs.

After the Data Refresh: After the conventional refresh (Figure 7a), the target block includes no valid pages, as all its valid pages are moved to the new block; so, the target block is ready to be erased and is reclaimed by GC (later). In contrast, after our modified refresh, the target block is reprogrammed by the IDA coding, and the new block is newly programmed by the conventional coding; both the blocks continue to be used. Consequently, any read from the IDA block can experience reduced latencies, while the conventional block exhibits its original latencies. It is worthwhile to note the following points.

• Our IDA coding increases the number of in-use blocks in an SSD, since the refresh target blocks that are supposed to be erased (by GC later) are not erased. To investigate the impact of an increase in the in-use block count, we analyze the block usage behavior of the tested workloads (Section IV). First, the increases are not significant (i.e., 2$\sim$4%), compared to our baseline 512GB SSD capacity (i.e., 350,208 blocks). Specifically, our IDA coding uses 14$\sim$30% (25%, on average) more storage capacity, compared to the original workload sizes (20$\sim$110GB), which still takes a small portion of the total SSD size. Second, the number of in-use blocks does not continue to increase. This is because (i) our design forces IDA-blocks to be reclaimed in the next refresh cycle and (ii) IDA-blocks are more likely to be reclaimed by GC as they include relatively small number of valid pages. Consequently, in the worst-case scenario, the GC cost can increase by 2 to 4%, which reduces the lifetime and throughput a bit (as a negative side effect of 28% read performance improvement). To confirm this, assuming that the 512GB user space is fully utilized, with an additional 15% [24] over-provisioned space (i.e., 75GB), first we execute the tested workloads using our IDA coding; and then execute write-intensive workloads from [25]. In this scenario, the GC invocation and the number of block erases increase by up to 3%, compared to the case without our IDA coding. Note that this overhead decreases as we continue to execute write-intensive workloads, since IDA blocks are reclaimed by GC. These results indicate that the impact of our optimization (which targets read-intensive applications) on write-intensive applications is negligible (if both types of applications share the same SSD space).

• Writing back error-free data into new blocks due to the data corruption during the voltage adjustment (❽ in Figure 7) does hurt the flash lifetime; note that those data writes are not extra. In the baseline refresh, all the valid data are supposed to be written into a new block. In contrast, our modified refresh writes only non-beneficial data and erroneous data after the voltage adjustment, instead of all valid data. Compared to the baseline, the total write count decreases a little in our proposal.

• One might propose to simply move the valid page data of the (refresh) target blocks to fast LSB pages of new blocks; doing so can reduce the read latencies of the valid pages. However, this approach does not improve overall read performance. It is because (i) the number of fast LSB pages in a block is limited and (ii) if those LSB pages of new blocks are occupied by the valid MSB/CSB data of target blocks, the LSB page data of the target blocks and new I/O data will be placed into the slow MSB/CSB pages of new blocks. In contrast, out IDA coding enhances slow MSB/CSB data of the target blocks, and uses the limited fast LSB pages of the new blocks for LSB data.

TABLE I: Eight possible scenarios of a TLC WL, depending on the validation status of the LSB, CSB, and MSB pages. In our current implementation, the IDA coding is applied to the WLs for cases 1 to 4.

Bit	**Case 1**	**Case 2**	**Case 3**	**Case 4**
LSB	*Valid*	*Invalid*	*Valid*	*Invalid*
CSB	*Valid*	*Valid*	*Invalid*	*Invalid*
MSB	*Valid*	*Valid*	*Valid*	*Valid*
Action	Move LSB Adjust Voltage for CSB/MSB		Move LSB Adjust Voltage for MSB	
Bit	**Case 5**	**Case 6**	**Case 7**	**Case 8**
LSB	*Valid*	*Invalid*	*Valid*	*Invalid*
CSB	*Valid*	*Valid*	*Invalid*	*Invalid*
MSB	*Invalid*	*Invalid*	*Invalid*	*Invalid*
Action	Move LSB/CSB	Move CSB	Move LSB	Nothing To Do

Selecting Pages to Apply IDA Coding: During our modified data refresh, ❸ for each valid page in the target block, one needs to decide to move it to the new block or re-program it using the IDA coding in the target block. Table I lists the possible scenarios of WLs in the target block where the data refresh is executed. In the current implementation, we apply the IDA coding to the WLs in cases 1–4, whereas the pages of the WLs in cases 5–7 are moved to the new block as the original data refresh does; there is nothing to do for the WLs in case 8. More specifically, cases 2 and 4 are the best targets for our new coding, which can significantly reduce the CSB and MSB read latencies. Furthermore, we can convert cases 1 and 3 to cases 2 and 4, respectively, by moving the valid LSB page to the new block. In contrast, cases 5 to 8 get no or little benefit from our IDA coding; as a result, we move them to the new block as the original data refresh does. Note that, in the original (baseline) refresh, all the valid pages in the target block are moved to the new block.

Performance Overheads: Implementing the IDA coding increases the execution time of data refresh, since it incurs additional reads and writes. We present its overhead analysis in terms of the number of reads and writes. The total numbers of reads and writes in the original data refresh operation are N_{valid} and N_{valid}, where N_{valid} is the number of valid pages in the target block, as moving each page involves a read and a write. In contrast, in our modified data refresh, the number of reads is $N_{valid} + N_{target}$ due to the additional reads after the voltage adjustment (❺ in Figure 7); here, N_{target} is the number of pages reprogrammed by the IDA coding in the target block. The number of writes is $N_{target} + (N_{valid} - N_{target}) + N_{error} = N_{valid} + N_{error}$, where N_{error} is the number of pages corrupted during the voltage adjustment and so written into the new block, due to ❹ the voltage adjustment for N_{target} pages, ❸ the relocation for the remaining $N_{valid} - N_{target}$ pages, and ❽ the write-back for the corrupted N_{error} pages, as shown in Figure 7. Accordingly, the additional read and write counts in our modified refresh, compared to the original data refresh, are N_{target} and N_{error}, respectively. Considering that (i) a write latency is much longer than a read latency and (ii) the number of additional writes (N_{error}) is much smaller than the number of original writes (N_{valid}), we believe that the overheads brought by the application of our IDA coding would be successfully hidden by the (originally) high overhead data refresh. Section V-B evaluates the benefits and overheads of our IDA coding by varying the error rate (N_{error} value).

Hardware/Software Overheads: Our new coding brings small modifications to the flash controller and data structures. Specifically, the FTL needs an additional bit (per block) to indicate whether each block is the conventional block or IDA block, and another bit (per WL) to identify the reprogrammed code – each reprogrammed WL can accommodate both the CSB and MSB (cases 1 and 2 of Table I) or the MSB only (cases 3 and 4 of Table I). Since the two reprogrammed codes use different sets of read voltages, the flash controller also needs to be designed to switch the read voltages, depending on the values of the two added bits. Finally, the refresh function is modified to accommodate the voltage adjustment.

Critical Points: We want to highlight a few critical points in our design as follows:

• The data refresh is a key operation for read-intensive applications where the GC is rarely invoked. In the current implementation, as simulated in [23], we set the refresh period from 3 days to 3 months across our applications (Section IV); we do *not* intentionally reduce the refresh period.

• Our proposal does *not* trade off the storage lifetime with read performance. As discussed above, each cell is fully utilized (towards the highest voltage level) while keeping the erase count unchanged. Also, our refresh-based voltage adjustment does not incur additional writes. Our proposal does *not* sacrifice the storage capacity either; the number of in-use blocks can increase, but all IDA blocks are reclaimed later.

• *No* additional mechanism that tracks the validation status of a page is required. We use the existing "block status table" that captures the validation status of each page of each block.

• Under our approach, there is *no* change in ECC encoding and decoding algorithms, since the IDA coding does *not* change the stored data. Note that our coding scheme changes how data are stored in and read from the flash memory.

• There is *no* need of additional batteries in case of a power failure. First, DRAM usage/size does not increase, compared to the general SSDs supporting a block refresh. Specifically, as our IDA coding is implemented as part of a refresh operation, the number of valid pages (including metadata) temporarily kept in the DRAM does not change during the voltage adjustment. Second, we do not have to protect ongoing voltage adjustment process from a power failure, since we maintain error-free page data (and their metadata) in the DRAM and write them into new blocks (and out of band area therein), as the conventional refresh operation does.

TABLE II: Configuration of the baseline SSD and flash.

SSD	512GB capacity, sixteen 32GB flash chips [16], 4 channels, 4 flash chips/channel, PCIe 3.0 4-lane host interface (12GB/s)
Flash chip [16]	Triple-Level-Cell (3 bits/cell), 2 dies/chip, 2 planes/die, 5472 blocks/plane, 192 pages/block, 8KB page
Timing values [16]	50us (LSB), 100us (CSB), 150us (MSB) for page reads, 2.3ms for page write, 3ms for block erase, 333MT/sec data transfer rate (48us/8KB page), 20us for ECC decoding [15]
FTL	Remapping-based refresh [23] – 3 days to 3 months refresh period across applications, CWDP allocation [26], read-first scheduling, greedy (wear-aware) garbage collection [27]

IV. Evaluation Setup and Methodology

A. Simulation Framework

To model an SSD employing our proposed technique, we used the DiskSim simulator [28] with the SSD extensions [29]. DiskSim is highly modularized and parameterized; so, one can easily customize it. In the DiskSim+SSD simulator, we modified one existing module (*flash timing*) and added two new modules (*data refresh* and *IDA coding*), as follows:

• The *flash timing* model is modified to have different values based on the page types (i.e., reading the LSB, CSB, and MSB pages exhibit different latencies); the default simulator uses an average value for the flash read latency.

• The *data refresh* function implements the refresh mechanism, which moves valid pages from the target block to a new block, after a threshold timeout.

• The *IDA coding* function implements the voltage adjustment to the target block (as part of the data refresh function), which changes the CSB and MSB read latencies.

B. System Configuration

To mimic a modern flash-based storage system, we modeled a baseline system as a 512GB SSD with sixteen 32GB TLC flash chips, whose configuration is given in Table II. Specifically, there are 4 memory channels, each combining 4 flash chips; also, the host interface is a PCIe with 12GB/s bandwidth [30]. For the flash details including the timing information and internal organization, we used a modern TLC flash from Micron Technology [16]. In this device, there are 2 dies per chip, 2 planes per die, 5472 blocks per plane, and 192 8KB pages per block. The device read latencies (memory access times) are 50us, 100us, and 150us for the LSB, CSB, and MSB pages, respectively. We also assume that the FTL employs a CWDP static allocation [26] (Channel first, Chip second, Die third, Plane last) as its page mapping strategy, and prioritizes read I/Os over write I/Os during scheduling. Two other important functionalities of the FTL are data refresh and garbage collection (GC): the data refresh mechanism moves valid page data from old blocks to new blocks periodically. More specifically, we fix the refresh period from 3 days to 3 months depending on the workload. We employ a detailed refresh simulation methodology from [23]. The GC selects target blocks using the GREEDY algorithm [27].

C. Evaluated Systems

We evaluate and compare the following three systems:

1) **Baseline:** This has the SSD/flash configuration described in Table II. It schedules read I/Os first and (regularly) triggers the default refresh mechanism.

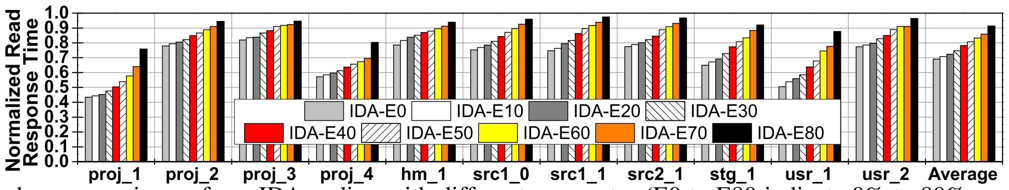


Fig. 8: The read response times of our IDA coding with different error rates (E0 to E80 indicate 0% to 80% error rates) during the voltage adjustment process, normalized to the baseline.

TABLE III: Important characteristics of our workloads. The second column indicates how read intensive each workload is.

Name	Read Ratio (%)	Read Size (KB)	Read Data Ratio (%)	Ratio of MSB Whose LSB/CSB Invalid (%)
proj_1	89.43	37.45	96.71	22.12
proj_2	87.61	41.64	85.77	32.47
proj_3	94.82	8.99	87.41	20.81
proj_4	98.52	23.72	99.3	24.63
hm_1	95.34	14.93	93.83	20.54
src1_0	56.43	36.47	47.42	33.31
src1_1	95.26	35.87	98	34.79
src2_0	97.86	60.32	99.51	21.27
stg_1	63.74	59.68	92.99	38.76
usr_1	91.48	52.72	97.37	45.44
usr_2	81.13	50.89	94.01	21.43

2) **IDA-Coding-E20:** This employs our IDA coding; whenever a refresh occurs, the controller adjusts the threshold voltage of each WL in the target block. This system assumes that 20% of the pages become erroneous during this process; so, those corrupted page data are written back into the new block. We also conduct a sensitivity test that varies the error rates from 10% to 80% (Section V-B).
3) **IDA-Coding-E0:** This system is similar to IDA-Coding-E20; however, it assumes that there is *no* error (disturbance) during the voltage adjustment. This ideal system can maximize chances for enhancing the CSB and MSB latencies, while minimizing the voltage adjustment overhead; so, it provides an "upper bound" for the performance benefits that could be coming from our IDA coding.

Unless stated otherwise, for comparison, the results of the evaluated systems are *normalized* to the baseline system.

D. Workloads

Recall that our proposed technique targets read-intensive workloads. Therefore, to evaluate its impact on improving the performance, we used 11 read-intensive applications from the MSR Cambridge suite [25]. Note that these benchmarks have been heavily used in similar flash-based studies (e.g., [31], [32]) that perform various read optimizations. Table III quantifies several important characteristics of our tested workloads: the *read request ratio*, *average read request size*, *read ratio in terms of the amount of data*, and *fraction of the MSB reads whose associated LSB or/and CSB is/are invalid*.

V. Evaluation Results

A. Read Response Time Enhancement

Figure 8 shows the read response times of our proposed systems, *normalized* to the baseline (see the two systems, IDA-E20 and IDA-E0, only). IDA-Coding-E20 and IDA-Coding-E0 improve the read response time by 28% and 31%, respectively. Most workloads see their read response times reduced by around 20%, while some workloads such proj_1 and usr_1 achieve significant gains (over 50%).

TABLE IV: The average overhead brought by voltage adjustment during the data refresh for a 192-page (64 WLs) block.

	Original Refresh Reads&Writes	Additional Operations for IDA	
Name	# Valid Pages / # Total Pages	# of Reads	# of Writes
proj_1	122.88 / 192	60.98	12.19
proj_2	122.21 / 192	60.47	12.09
proj_3	128.69 / 192	63.77	12.75
proj_4	114.87 / 192	56.41	11.28
hm_1	103.34 / 192	51.24	10.24
src1_0	130.26 / 192	64.29	12.86
src1_1	102.14 / 192	50.54	10.11
src2_0	116.36 / 192	57.53	11.51
stg_1	142.67 / 192	70.68	14.13
usr_1	98.58 / 192	48.61	9.72
usr_2	113.69 / 192	56.39	11.28

There are two sources of the improvement. After applying the IDA coding, reading such MSB page data takes the same time as an LSB read; and this directly improves the overall read response time. Furthermore, the enhanced MSB read latency results in a reduction in the I/O wait time (the I/O stall time due to I/Os being processed); this can indirectly improve the response times of the LSB, CSB, and MSB reads.

B. Benefits under Different Coding Error Rates

The effect of the program disturbance (while applying our IDA coding technique) can significantly vary depending on a wide range of parameters such as the technology node, the voltage applied to the WLs, and the neighboring data (the victim data) [33], [34]. Accordingly, here we evaluate the effectiveness of our IDA coding when the error rates due to the voltage adjustment are varied (from 0% to 80%). Figure 8 reports normalized read response times of our IDA coding with different error rates during the voltage adjustment. Here, IDA-Ex indicates that x% of the pages experience errors. We make the following important observations:

• **Still High Benefits in E50:** In general, as our IDA coding incurs more errors, its performance gain gradually decreases. However, even though applying the IDA coding generates 50% errors during the voltage adjustment (i.e., IDA-E50), it still improves the read response time by 20.2%, on average. Therefore, we believe that it is worthwhile to apply our proposed coding even in devices/cases where the voltage adjustment brings severe program interference.

• **Diminishing Benefits in High Error Rates:** When the error rate is 80% (i.e., IDA-E80), the enhancement of the read response time goes below 7%, on average. In extreme cases (error-prone devices), one needs to consider the magnitude of benefits and overheads brought by our IDA coding.

C. Refresh Overhead Analysis

In general, a refresh operation takes a long time to move (read and write) valid pages of the target block to a new block. Especially, a majority pages in a refresh target block are valid; so, it is typically a more time-consuming procedure than the

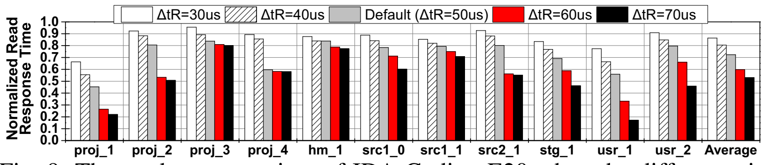


Fig. 9: The read response time of IDA-Coding-E20 when the difference in the read latencies (ΔtR) among LSB, CSB, and MSB varies from 30us to 70us, normalized to the baseline.

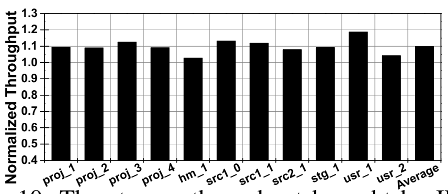


Fig. 10: The storage throughput brought by IDA-Coding-E20, normalized to the baseline.

GC. Our experiments confirm that most of the pages in the refresh target block are also valid, as shown in the second column of Table IV. Specifically, in our block setting (64 WLs x TLC = 192 pages), the number of valid pages is 113, on average (ranging from 98 to 130), and thus, each refresh operation (composed of reads and writes of 113 pages) needs a very long time – note that this is the already-existing (original) refresh cost, which is *not* related to our proposed technique.

To hide the overhead of applying our IDA coding, we implement our voltage adjustment process on top of the long-lasting refresh operation. As discussed in Section III-C, after reading all valid pages (as part of data refresh), the threshold voltages of the target WLs are adjusted. To check whether there is a loss of data integrity during the voltage adjustment, (i) all the pages reprogrammed by the IDA coding are read, which are the additional reads over the original refresh cost. For the corrupted pages disturbed by the voltage adjustment, (ii) the error-free version page data are written back to the new block – this also represents an additional overhead (writes).

• **Additional Reads:** The third column of Table IV provides the number of additional page reads. As a result of the voltage adjustment, an average of 58 pages are read to check whether they are corrupted or not. These additional reads can be regarded as a high overhead; but, such is not the case in practice, because (i) write latency is much longer than read latency, and a large number of writes are dominant in the total refresh cost, and (ii) all the pages (to be read) are programmed by the IDA coding; as a result, reading each of them needs only one or two accesses to the memory.

• **Additional Writes:** The number of additional page writes is given in the fourth column of Table IV. The IDA-Coding-E20 needs an average of 11 pages to be written into the new block, which can vary depending on the error rates. Even though these extra writes directly increase the latency of each refresh operation, one can see that the IDA coding still brings significant benefits in the response time and throughput.

D. Throughput Improvement

Figure 10 shows the device throughput results, *normalized* to the baseline. All the tested workloads experience an increased storage throughput (10%, on average). There are two major parameters that shape the storage throughput:

• **Reduced Read Latency:** A decrease in the device latency leads to an increase in the throughput, since it can reduce the I/O wait times and process more I/O requests at a unit time. Our proposed coding technique reduces the MSB/CSB read latency to the LSB read latency, and this is a major contributor to the throughput improvement observed.

• **Data Refresh Overhead:** The additional reads and writes increase the latency of each refresh operation, which has a negative impact on the storage throughput. However, the benefits coming from the reduced read latency are overwhelming; so, it is worthwhile to put up with the increased refresh overheads.

E. Device Sensitivity Analysis

Our proposed coding technique is motivated by the fact that the LSB, CSB, and MSB pages have all different read latencies (tR). In our default configuration (see Table II), tR-LSB, tR-CSB, and tR-MSB are 50us, 100us, and 150us, respectively, and ΔtR is 50us (tR-MSB - tR-CSB = tR-CSB - tR-LSB = 50us). However, ΔtR is a device-specific value, and as a result, it can vary, depending on the device characteristics such as feature size and memory array size. Accordingly, here, we evaluate the effectiveness of our IDA coding under various device configurations (i.e., ΔtR). Figure 9 plots the read response time brought by IDA-Coding-E20, when varying ΔtR from 30us to 70us, *normalized* to the baseline system. We want to emphasize two points in this analysis:

• **More Effective in High ΔtR Devices:** As the difference in read latencies among the LSB, CSB, and MSB pages increases, our IDA coding becomes more beneficial. One can see that, when using a device whose ΔtR is 70us, IDA-Coding-E20 can improve the read response time by 49%, on average, and up to 83% for usr_1. Since our coding technique can reduce the long MSB/CSB latencies to the short LSB latency in many cases, a larger latency gap among pages leads to higher performance benefits.

• **Still Beneficial in Low ΔtR Devices:** Even if the target device has a small ΔtR (i.e., the read time difference among pages is not significant), our coding technique is still worthwhile to employ for high-performance storage systems. For example, if a device exhibits a ΔtR of 30us, IDA-Coding-E20 can still bring a 14% of improvement in read response times.

F. Results of the Read Retry Simulation

Our IDA coding would be more beneficial in later portions of the lifetime of an SSD, where the Raw Bit Error Rate (RBER) increases. Note that our proposed coding (that is designed for read-dominant applications and has a negligible impact on the SSD lifetime) can be applied in any portion (e.g., earlier, later, entire) of the SSD lifespan. When the RBER is high (hence, when a read is failed), modern LDPC-based ECCs [14], [15], [35]–[37] make the read successful by retrying the page read with different voltage values (i.e., accessing the memory multiple times by changing the read voltage levels). However, such a series of accesses to the

TABLE V: The read response time improvements brought by IDA-Coding-E20 over the baseline in an MLC-based SSD.

Name		proj_1	proj_2	proj_3	proj_4	hm_1
Resp. Time Imp.		30.8%	8.2%	16.3%	8.1%	7.8%
src1_0	**src1_1**	**src2_0**	**stg_1**	**usr_1**	**usr_2**	**Average**
18.3%	9.6%	3.4%	19.8%	31.8%	10.6%	14.9%

memory can significantly increase the response time of a single page read. Therefore, our proposed coding technique can be a solution that helps to relieve this high read overhead brought by read-retries. To evaluate IDA-Coding-E20 in the read-retry phase (i.e., late lifetime), we modeled ECC decoding failure probability in terms of the number of extra sensing [38]. Figure 11 shows the effectiveness of IDA-Coding-E20 in both the earlier and the later parts of the SSD lifetime. When the device goes through the later portions of its lifetime where read-retries occur more frequently, our proposed technique improves the read response time by 42.3%, on average; it brings far more benefits, compared to the earlier portions of the lifetime where there is no read-retry (28%).

G. Results with an MLC Device

The target of our proposed coding technique is the "asymmetry" in the read latencies of flash memories; so, it can be applied to MLC devices as well. For the simulation of an MLC-based SSD, we set the read latencies for the LSB and MSB pages to 65us and 115us, respectively, based on a device specification [39]. Table V presents the read response time improvements brought by IDA-Coding-E20 running on an MLC-based SSD. We see an average of 14.9% improvement over the baseline system. Although significant, this amount of improvement is relatively lower compared to the TLC case. This is because the difference between the LSB and MSB read latencies is less significant in MLC devices, whereas TLC devices provide three types of pages and exhibit huge latency gaps among them. We believe that our proposal will be more effective in QLC devices (when they become mainstream), where the "asymmetric latencies" among four types of pages will be more problematic; a detailed QLC device evaluation is left as a future work.

VI. Related Works

We categorize the representative SSD performance studies into three groups, based on the target problem they attack:

Hiding GC Overheads: Since the GC is a time-consuming task that occupies the SSD resources, it prevents normal I/O requests from being processed until it finishes. To hide this big GC overhead, Lee et al. [40] proposed to (i) suspend the ongoing GC, (ii) process the pending I/O requests first, and (iii) then resume the GC later. In contrast, the strategy proposed by Jung et al. [41] segmented the entire GC operation and distributed the resulting segments (smaller GCs) over non-critical I/O requests. Also, motivated by the tail latency problem, Yan et al. [42] provided a set of strategies to avoid the devastating impact of the GC on I/O requests. These works mainly target *write-intensive applications* where the GC is frequently invoked; however, our proposed coding is aimed for *read-dominant applications*.

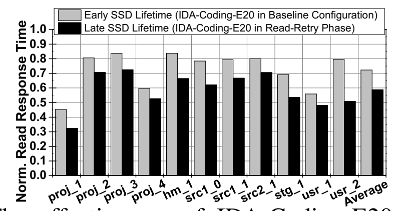


Fig. 11: The effectiveness of IDA-Coding-E20 in different portions of the SSD lifetime, normalized to the baseline.

Avoiding Resource Conflicts: As an SSD includes many shared resources, multiple ongoing I/O requests can conflict with one another, and this in turn increases the I/O service times. Wu and He [43] suggested to suspend the ongoing write I/Os, process read I/Os first, and resume the write I/Os, when the long-lasting write I/Os block the service of the read I/Os with short latencies. In comparison, the SSD controller proposed in [44] reschedules I/O requests by being aware of the resource utilization and I/O parallelism, with the goal of minimizing the I/O wait times. Furthermore, there are severe I/O contentions in a scenario where multiple users share an SSD; this problem has been the main target of a recent proposal by Huang et al. [45]. We want to emphasize that, all these prior techniques attack the SSD-level problem; but, our proposal is a flash-level optimization.

Exploiting ECC Characteristics: The proposals in this category [31], [46]–[49] can enhance the read latency by reducing the number of read retries in SSDs, with LDPC soft decoding ECC. To achieve this, some take advantage of SSD architectures and functions, while others utilize the unique flash error characteristics. Note that these techniques can further improve the read latency when used along with our IDA coding that enhances a single read latency.

VII. Conclusions

High density flash memories suffer from long read latencies. We observed from the conventional TLC coding that the long read latencies of the CSB and MSB pages are not reduced even when the associated LSB page has become invalid. We proposed a new coding technique (called the IDA coding), which can be applied to a used block programmed by the conventional coding. Our new coding can bring the read latencies of CSB and MSB pages close to the LSB page read latency. We also demonstrated that leveraging the data refresh to implement the IDA coding can prevent the potential data reliability loss and minimize the performance overheads. Our extensive evaluations using a variety of read-dominant workloads revealed that the IDA coding can improve read response times by 28%, on average.

Acknowledgements

This work is supported in part by NSF grants 1822923, 1439021, 1629915, 1626251, 1629129, 1763681, 1526750 and 1439057. Jung's research is in part supported by NRF 2016R1C1B2015312, DOE DEAC02-05CH11231, IITP-2018-2017-0-01015, NRF 2015M3C4A7065645, Yonsei Future Research Grant (2017-22-0105), and a grant from MemRay. M. Jung and M. Kandemir are the co-corresponding authors.

REFERENCES

[1] M. Webb, "3D NAND Status and Roadmap 2017," in *FMS*, 2017.

[2] L. M. Grupp, J. D. Davis, and S. Swanson, "The Bleak Future of NAND Flash Memory," in *USENIX FAST*, 2012.

[3] J. Zhang, G. Park, D. Donofrio, M. Shihab, J. Shalf, and M. Jung, "OpenNVM: An Open-Sourced FPGA-based NVM Controller for Low Level Memory Characterization," in *ICCD*, 2015.

[4] Y. Pan, G. Dong, Q. Wu, and T. Zhang, "Quasi-Nonvolatile SSD: Trading Flash Memory Nonvolatility to Improve Storage System Performance for Enterprise Applications," in *HPCA*, 2012.

[5] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom, "SFS: Random Write Considered Harmful in Solid State Drives," in *USENIX FAST*, 2012.

[6] R.-S. Liu, C.-L. Yang, and W. Wu, "Optimizing NAND Flash-Based SSDs via Retention Relaxation," in *USENIX FAST*, 2012.

[7] Toshiba, "SSD (Solid State Drives)," https://toshiba.semicon-storage.com/us/product/storage-products/trends-technology/ssd-0.html.

[8] R. Micheloni, L. Crippa, and A. Marelli, "Inside NAND Flash Memories," in *Springer*, 2010.

[9] D. Kearns, "Practical Guide to Endurance and Data Retention," in *Cypress – AN99121*, 2015.

[10] O. Workgroup, "Open NAND Flash Interface Specification Revision 4.0," 2014, http://www.onfi.org.

[11] K. Suh, B.-H. Suh, Y.-H. Lim, J.-K. Kim, Y.-J. Choi, Y.-N. Koh, S.-S. Lee, S.-C. Kwon, B.-S. Choi, J.-S. Yum, J.-H. Choi, J.-R. Kim, and H.-K. Lim, "A 3.3V 32Mb NAND Flash Memory with Incremental Step Pulse Programming Scheme," in *ISSCC*, 1995.

[12] M. Abraham, "Architectural Considerations for Optimizing SSDs," in *FMS*, 2014.

[13] K. Zhao, K. S. Venkataramant, X. Zhang, J. Li, N. Zheng, and T. Zhang, "Over-Clocked SSD: Safely Running Beyond Flash Memory Chip I/O Clock Specs," in *HPCA*, 2014.

[14] J. Zhu, "High-Throughput LDPC Solution for Reliable and High Performance SSD," in *FMS*, 2016.

[15] Y. Wang, "Ultra High Throughput LDPC Schemes for SSD," in *FMS*, 2016.

[16] Micron, "NAND Flash Memory, MT29F64G08EBAA[A/B], MT29F128G08EFAA[A/B], MT29F256G08EJAA[A/B]," https://www.micron.com/products/nand-flash.

[17] D. Arteaga and M. Zhao, "Client-side Flash Caching for Cloud Systems," in *SYSTOR*, 2014.

[18] M. Kwon, J. Zhang, G. Park, W. Choi, D. Donofrio, J. Shalf, and M. Kandemir, "TraceTracker: Hardware/Software Co-Evaluation for Large-Scale I/O Workload Reconstruction," in *IISWC*, 2017.

[19] S. Liu and X. Zou, "QLC NAND Study and Enhanced Gray Coding Methods for Sixteen-Level-Based Program Algorithms," in *Microelectronics Journal, Vol. 66*, 2017.

[20] G. Yadgar, E. Yaakobi, and A. Schuster, "Write Once, Get 50% Free: Saving SSD Erase Costs Using WOM Codes," in *USENIX FAST*, 2015.

[21] M. Fabio, G. Yadgar, E. Yaakobi, Y. Li, A. Schuster, and A. Brinkmann, "The Devil Is in the Details: Implementing Flash Page Reuse with WOM Codes," in *USENIX FAST*, 2016.

[22] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf, "Characterizing Flash Memory: Anomalies, Observations, and Applications," in *MICRO*, 2009.

[23] Y. Cai, G. Yalcin, O. Mutlu, E. F. Haratsch, A. Cristal, O. Unsal, and K. Mai, "Flash Correct-and-Refresh: Retention-Aware Error Management for Increased Flash Memory Lifetime," in *ICCD*, 2012.

[24] K. Smith, "Understanding SSD Over Provisioning," in *FMS*, 2013.

[25] D. Narayanan, A. Donnelly, and A. Rowstron, "Write Off-Loading: Practical Power Management for Enterprise Storage," in *USENIX FAST*, 2008, http://iotta.snia.org/traces/388.

[26] M. Jung and M. Kandemir, "An Evaluation of Different Page Allocation Strategies on High-Speed SSDs," in *USENIX HotStorage*, 2012.

[27] W. Bux and I. Iliadis, "Performance of Greedy Garbage Collection in Flash-based Solid-State Drives," in *Journal of Performance Evaluation, VOL. 67, Issue. 11*, 2010.

[28] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger, "The DiskSim Simulation Environment Version 4.0 Reference Manual," in *CMU-PDL-08-101*, 2008, http://www.pdl.cmu.edu/DiskSim/.

[29] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," in *USENIX ATC*, 2008, http://www.microsoft.com/en-us/download/details.aspx?id=52332.

[30] PCI-SIG, "PCI Express Base 3.0 Specification," 2012, http://pcisig.com.

[31] Y. Du, D. Zou, Q. Li, L. Shi, H. Jin, and C. J. Xue, "LaLDPC: Latency-aware LDPC for Read Performance Improvement of Solid State Drives," in *MSST*, 2017.

[32] G. Wu and X. He, "Reducing SSD Read Latency via NAND Flash Program and Erase Suspension," in *USENIX FAST*, 2012.

[33] Y. Cai, G. Yalcin, O. Mutlu, E. F. Haratsch, O. Unsal, A. Cristal, and K. Mai, "Neighbor-Cell Assisted Error Correction for MLC NAND Flash Memories," in *SIGMETRICS*, 2014.

[34] Y. Cai, O. Mutlu, E. F. Haratsch, and K. Mai, "Program Interference in MLC NAND Flash Memory: Characterization, Modeling, and Mitigation," in *ICCD*, 2013.

[35] R. G. Gallager, "Low-Density Parity-Check Codes," in *IRE Transactions on Information Theory*, 1962.

[36] O. Vahabzadeh, "ECC for NAND Flash," in *FMS*, 2017.

[37] A. Marelli, "False Decoding Probability (Detection) of BCH and LDPC Codes," in *FMS*, 2016.

[38] K. Zhao, W. Zhao, H. Sun, T. Zhang, X. Zhang, and N. Zheng, "LDPC-in-SSD: Making Advanced Error Correction Codes Work Effectively in Solid State Drives," in *USENIX FAST*, 2013.

[39] Micron, "NAND Flash Memory - MLC+, MT29F128G08CBCCB, MT29F256G08CECCB, MT29F512G08C[K/M]CCB, MT29F512G08CLCCB, MT29F1T08CQCCB, MT29F1T08CUCCB, MT29F2T08CTCCB, MT29F2T08CVCCB," https://www.micron.com/products/nand-flash.

[40] J. Lee, Y. Kim, G. M. Shipman, S. Oral, and J. Kim, "Preemptible I/O Scheduling of Garbage Collection for Solid State Drives," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 32, No. 2*, 2013.

[41] M. Jung, W. Choi, S. Srikantaiah, J. Yoo, and M. Kandemir, "HIOS: A Host Interface I/O Scheduler for Solid State Disks," in *ISCA*, 2014.

[42] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, P. Machines, A. A. Chien, and H. S. Gunawi, "Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs," in *USENIX FAST*, 2017.

[43] G. Wu and X. He, "Reducing SSD Read Latency via NAND Flash Program and Erase Suspension," in *USENIX FAST*, 2012.

[44] M. Jung and M. Kandemir, "Sprinkler: Maximizing Resource Utilization in Many-Chip Solid State Disks," in *HPCA*, 2014.

[45] J. Huang, A. Badam, L. Caulfield, S. Nath, S. Sengupta, B. Sharma, and M. K. Qureshi, "FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs," in *USENIX FAST*, 2017.

[46] Y. Du, Q. Li, L. Shi, D. Zou, H. Jin, and C. J. Xue, "Reducing LDPC Soft Sensing Latency by Lightweight Data Refresh for Flash Read Performance Improvement," in *DAC*, 2017.

[47] K. Zhao, W. Zhao, H. Sun, T. Zhang, X. Zhang, and N. Zheng, "LDPC-in-SSD: Making Advanced Error Correction Codes Work Effectively in Solid State Drives," in *USENIX FAST*, 2013.

[48] R.-S. Liu, M.-Y. Chuang, C.-L. Yang, C.-H. Li, K.-C. Ho, and H.-P. Li, "EC-Cache: Exploiting Error Locality to Optimize LDPC in NAND Flash-Based SSDs," in *DAC*, 2014.

[49] J. Guo, W. Wen, J. Hu, D. Wang, H. Li, and Y. Chen, "FlexLevel: a Novel NAND Flash Storage System Design for LDPC Latency Reduction," in *DAC*, 2015.