BitFlip: A Bit-Flipping Scheme for Reducing Read Latency and Improving Reliability of Flash Memory

Suzhen Wu *, Sijie Lan *, Jindong Zhou *, Hong Jiang †, Zhirong Shen *

* School of Informatics, Xiamen University, China

† Department of Computer Science and Engineering, The University of Texas at Arlington, USA

suzhen@xmu.edu.cn, sijielan@gmail.com, zhoujindddd@qq.com, hong.jiang@uta.edu, shenzr@xmu.edu.cn

***Abstract*—LDPC codes provide stronger error correction capability for flash memory, but at the expense of high decoding latency that leads to poor read performance. In this paper, we demonstrate via preliminary analysis that the four states of a MLC cell in flash memory differ substantially in error proneness and proportion, which opens up new opportunities for reducing the read latency. We therefore design BitFlip, a lightweight yet effective bit-flipping scheme for flash memory. BitFlip carefully examines the bits at the proper granularity and looks for opportunities to flip the error-prone data to make them more stable against retention errors, thereby reducing the decoding time. In-depth analysis and extensive experiments are conducted to show that BitFlip can reduce 25.9%-34.2% of the read latency and prolong 2.9%-33.3% of the lifespan for flash memory, while adding negligible impact on the write latency.**

***Index Terms*—LDPC Codes, Bit Flips, Read Performance, Reliability, NAND Flash Memory**

I. Introduction

Over the past few years, NAND flash memory [1] (also called "flash memory" for brevity) has been replacing hard disk drives as the primary storage device in many storage systems (e.g., mobiles and laptops), mainly due to its superior traits, such as low access latency, low power consumption, and high shock resistance. With the per-bit cost of the flash memory continuing to drop, the flash memory is poised to be deployed in a wide range of applications (e.g., storage servers and data centers).

However, flash memory has a potential defect in that it can only tolerate a certain number of *program/erase* (P/E) cycles, thereby aggravating the concern about the reliability of the data being stored in the flash memory. For example, the SLC flash, which stores one bit per cell, can only survive under 10 k P/E cycles [2]. More specifically, each cell, which is the basic storage unit in flash memory, is constructed of floating gate transistors. When writing data into a flash cell (also called "programming"), a certain number of electrons are injected into the floating gate to ensure that the resulting *threshold voltage* is properly set, such that the subsequent reads can obtain the right data by transforming the sensed threshold voltage. To re-program a flash cell, an *erase* operation has to be performed in advance to eject all the electrons that are currently captured in the floating gate. Frequently performing P/E cycles will damage the structure of the floating gate and weaken its capability to trap electrons. Consequently, the electrons are prone to escaping from the floating gate over time, resulting in flash errors. More seriously, as the flash memory continuously becomes denser, the number of P/E cycles that it can tolerate sharply declines. For example, the MLC (multi-level cell) flash memory, while capable of storing two bits in a cell, can merely endure about 3 k P/E cycles for 30-40nm technology generations [2].

The reliability weakness of flash memory calls for advanced error correction codes with stronger correction capability. To this end, the *low-density parity-check* (LDPC) code has emerged to be such a code [3]–[7]. Generally, LDPC code adopts two stages of decoding processes, namely, *hard-decision decoding* and *soft-decision decoding*, to increase the probability of data correction. LDPC code first tries hard-decision decoding to correct the arised errors. If the hard-decision decoding fails, then LDPC code will resort to the soft-decision decoding, which comprises several *decoding levels*. The decoding levels in the soft-decision decoding are performed iteratively until the decoding process succeeds, where at each level the data will be re-sensed and re-transferred for conducting another decoding trial. As a result, though the decoding probability increases with the number of decoding levels used, the decoding time is also progressively accumulated. Therefore, when the *raw bit error rate* (RBER) grows, the decoding latency in LDPC code is detrimental to the read performance.

In this paper, we mainly focus on *retention errors* for the MLC flash memory, which are caused by charge leakage over time and have been demonstrated as a main source of the flash errors in many independent studies [2], [8]. Generally, there are four possible states exhibited by a MLC flash cell, namely, '00', '01', '10', and '11'. The retention errors of a MLC flash cell can be formally represented as 'AB'→'CD', where 'AB' and 'CD' are the states stored in the MLC flash cell before and after the occurrence of retention errors, respectively. Our observation is that the four possible states that a MLC flash cell exhibits are substantially different in both error proneness and proportion in practice. On one hand, the fractions of erroneous changes between any two possible state pairs are relatively unbalanced, where '00'→'01' and '01→10' are the most frequent retention errors and account for about 90% of all possible retention errors [2]. On the other hand, we find via analyzing the real-world files that the states '00' and '01' take up the majority of all the four possible states (69.0% on

Corresponding author: Zhirong Shen (shenzr@xmu.edu.cn)

average, see Section II-C).

Based on this observation and insight, we then design BitFlip, a bit-flipping technique from the ground up to make use of the differences of the four possible states to simultaneously improve the read performance and reliability of flash memory. BitFlip first carefully examines the state proportions of the stored data at a proper granularity (e.g., 512 Bytes). It then looks for opportunity to flip the bits, with the primary objective of reducing the number of the states '00' and '01' that are prone to retention errors. This design can bring forth two advantages. First, by reducing the number of the states '00' and '01', BitFlip can effectively decrease the probability of retention errors, thereby reducing the decoding latency in LDPC code and improving the read performance. Second, by reducing the occurrence of retention errors, BitFlip helps prolong the lifespan of flash memory. To the best of our knowledge, BitFlip is the **first work** that employs the bit-flipping technique to simultaneously improve the read performance and reliability of flash memory. In addition, BitFlip is also orthogonal to prior work [3], [4], [6], [7], [9]–[12] and therefore can complement existing studies in the literature of flash memory.

Our main contributions can be summarized as follows.

- We first analyze a wide range of real-world data files and identify that the proportions of the four MLC states are substantially unbalanced in practice.
- We design BitFlip, a bit-flipping technique to improve the read performance and reliability for flash memory. BitFlip first carefully analyzes the proportion of the states at a proper granularity. It then duly flips the bits to reduce the distribution of the two error-prone states '00' and '01', with the primary objective of suppressing retention errors.
- We implement BitFlip in the SSDsim simulator [13] and conduct extensive evaluation with real-world traces and files, showing that BitFlip can reduce 25.9%-34.2% of the read latency and prolong 2.9%-33.3%% of the lifespan for flash memory, while adding negligible impact on the write latency.

The rest of this paper proceeds as follows. Section II presents the background and motivation. We then elaborate the main idea of the BitFlip in Section III and evaluate its performance in Section IV. Finally, we review the related work in Section V and conclude the paper in Section VI.

II. Background and Motivations

In this section, we first introduce the basics of flash (Section II-A) and elaborate the workflow of error correction in LDPC code (Section II-B). We then present the analysis of the state proportions in real-world data files, which guides and motivates the design of BitFlip (Section II-C).

A. Basics of Flash

A NAND flash cell is a floating gate metal oxide semiconductor field effect transistor (FGMOSFET), which stores the injected electrons to represent data information through the exhibited *threshold voltage* [1]. Figure 1 shows a basic flash

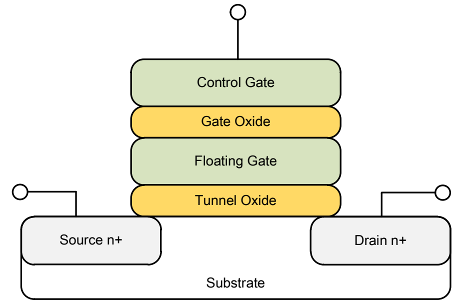
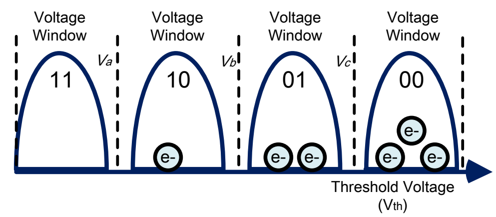
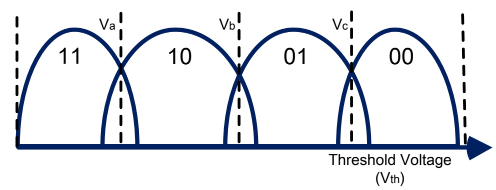


Fig. 1. Schematic diagram of a flash memory cell.



(a) The four voltage windows are non-overlapped.



(b) The boundaries of the four voltage windows blur with the increase of P/E cycles.

Fig. 2. The four states in a MLC flash cell.

memory cell, in which two insulation layers are located at the upper and lower ends of the floating gate, respectively. When the flash memory is powered off, the electrons in the floating gate are still trapped, thereby producing the non-volatile nature of the NAND flash memory.

The evolving of microelectronics allows a flash cell to store more information. Based on the number of bits that a cell can store, we can classify the general flash memory into the following branches: SLC (i.e., one bit per cell), MLC (i.e., two bits per cell), TLC (i.e., three bits per cell), and QLC (i.e., four bits per cell). With the number of bits stored in a flash cell increases, the number of P/E cycles that it can endure dramatically drops. For example, the SLC flash memory can tolerate 10 k P/E cycles, while the QLC flash memory can survive under merely 1.5 k P/E cycles [14]. In this paper, we mainly focus on the MLC flash memory, as the MLC flash memory continues dominating the global flash memory market [15].

A flash cell uses threshold voltage to represent the data stored, where the threshold voltage of each cell heavily depends on the number of electrons stored in the floating gate. Consequently, the write and read operations to a flash cell can be realized by setting and sensing the threshold voltage, respectively [1], [2]. Specifically, to store n bits in a flash cell, the corresponding voltage range will be partitioned into 2^n *voltage windows*. Figure 2 shows an example of the four voltage windows of the MLC flash cell, which is supposed to arrange two bits (i.e., $n = 2$). When writing data to a flash cell (also called the "program" operation), the threshold voltage is set by injecting a certain number of electrons into the floating gate, ensuring that the resulting threshold voltage is properly in the range of the corresponding voltage window [3], [16]. Generally, the higher threshold voltage requires more charge injected. When reading the data in a flash cell, its threshold voltage will be sensed and then compared with the voltage window to identify the stored information.

A flash cell cannot be re-programmed directly. Before re-programming a flash cell, we have to perform an *erase* operation, which ejects all the electrons stored in the corresponding floating gate [1]. However, the frequently program/erase (P/E) cycles will damage the structure of the floating gate transistor and hence weaken the capability for the floating gate to trap electrons. As a consequence, after enduring enough P/E cycles, the threshold voltage is prone to shift and the noise margin between voltage windows gradually blurs (see Figure 2(b)).

In this paper, we mainly focus on the *retention errors* in the MLC flash memory, which are caused by the charge leakage over time and becomes the dominant source of flash memory errors [17]. Previous studies have uncovered that the distribution of retention errors closely relates to the states [2]. More particularly, because of the electron leakage in retention errors, the state with more electrons is much easier to shift to the left adjacent state with less electrons. To demonstrate, among the four states of the MLC flash cell, the most common retention errors are '00'→'01' (i.e., the original information '00' will change to '01' if the retention error occurs) and '01'→'10' (i.e., the original information '01' will shift to '10' if the retention error occurs), which take up 46% and 44% among all possible information changes under retention errors [2], respectively. It rarely happens that a state transits to another non-adjacent state (e.g., '00'→'11'). As the states '00' and '01' are more vulnerable to incur retention errors, we call them *error-prone states* throughout the paper.

B. Error Correction in LDPC

Because of the stronger error correction capability, LDPC code [18] is now extensively adopted in flash memory [3], [4], [6], [7], [19]. LDPC code corrects errors using the following two consecutive steps, namely *hard-decision decoding* and *soft-decision decoding* [20], where Figure 3 summarizes the workflow of the data decoding in the LDPC code. Specifically, it first uses the optimal reference voltage to read the data (represented in the binary form) and performs the hard-decision decoding. If the hard-decision decoding fails, it then resorts to the soft-decision decoding, which comprises several *decoding levels* [1] by default [12]. The soft-decision decoding is performed level-by-level and the error correction capability is

[1]Some papers also call them "sensing levels" [3] or "read levels" [12].

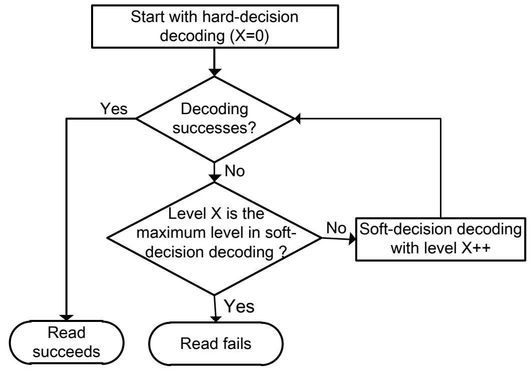


Fig. 3. The workflow of the data decoding in LDPC code.

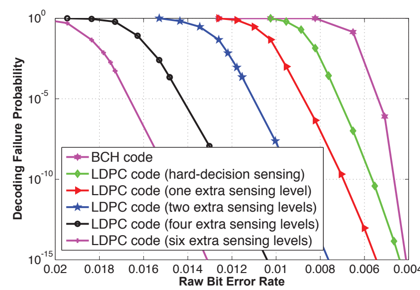


Fig. 4. Decoding failure probability versus the RBER for different LDPC and BCH codes [3].

stronger when the level of the soft-decision decoding increases [12]. Specifically, when using more levels in the soft-decision decoding, the LDPC decoder can re-sense more accurate information for decoding, thereby increasing the probability for data correction. Figure 4 shows the decoding probability versus the RBER for different decoding levels, indicating that for a given RBER, the LDPC code with more sensing levels has a lower decoding failure probability. For example, when the RBER is 0.016, the soft-decision decoding with six extra decoding levels (i.e., the LDPC code with six extra sensing levels in Figure 4) has a much smaller decoding failure probability than that with four extra decoding levels.

While LDPC code has stronger error correction capability, it severely deteriorates the read performance of flash memory. The reason lies in that the iterative decoding process in the hard-decision and soft-decision decodings of LDPC code accumulates the read latency [10]. For each trial, the system has to re-sense the stored data and re-transfer it to the LDPC decoder. Table I gives the latencies spent in the hard-decision decoding and different decoding levels of the soft-decision decoding, indicating that the decoding latency progressively grows with the number of the decoding levels used.

C. Analysis and Motivation

Analysis: To probe the real proportions of the four states in the MLC flash memory, we first carry out preliminary analysis based on the real-world files. The same as the previous work

TABLE I
RBERS AND THE CORRESPONDING DECODING LATENCIES.

Level	RBER	Read Latency
Hard-decision decoding	<0.005	$85us$
One extra decoding level	[0.005,0.006)	$109us$
Two extra decoding levels	[0.006,0.008)	$133us$
Three extra decoding levels	[0.008,0.009)	$157us$
Four extra decoding levels	[0.009,0.01)	$181us$
Five extra decoding levels	[0.01,0.012)	$205us$
Six extra decoding levels	[0.012,0.013]	$229us$

TABLE II
FILE SELECTED FOR ANALYSIS.

File Type	**File information**
Game files	`Game 1∼10`: Kerbal Space Program, Onigiri, Paunch, Robocraft, Sniper Fury, War Thunder, Destiny 2, Dark Deception, Star Conflict, Kika Raid
Image files	`Image 1∼10`: Linux (version: 1.1.13, 1.2.12, 1.3.12, 2.0.10, 2.2.21, 2.3.13, 2.4.19, 2.6.12, 3.0.11, 5.0.7)
Multimedia files	`Photos_1∼3`: New York Times [21]–[23] `Mp4_1∼3`: Vienna New Year Concert (2017∼2019) (Bit rate: 192 Kbps) `Mp3_1∼4`: Partita No. 2 in C Minor, Prelude and Fugue No. 9 in E Major, Prelude and Fugue No. 16 in G Minor, Prelude and Fugue No. 19 in A Major, (By *Johann Sebastian Bach*, Bit rate:192 Kbps)
Executable files	Executable files in Linux Kernel (5.3.0)

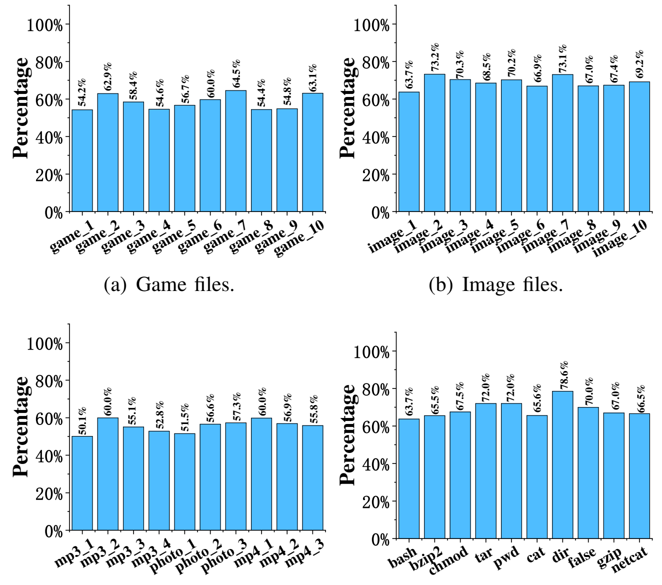
[24], we select a bunch of files from a wide range of file types, including game files, image files, multimedia files, and executable files. Table II lists the files and the corresponding file types that are chosen in this analysis. For each file, we analyze its content and measure the proportion of the *error-prone states* (i.e., states '00' and '01'). For example, based on ASCII, the character 'a' can be represented by the eight bits "01 00 00 01", thereby occupying four MLC flash cells (each MLC cell can store two bits, see Section II-A) to exhibit two '00' states and another two '01' states.

Figure 5 first presents the proportion of the error-prone states in different files across the four file types. We can observe that the error-prone states take up the majority among the four states in all the files. Specifically, the proportion of the error-prone states ranges from 50.1% (`MP3_1` in Figure 5(c)) to 78.6% (`dir` in Figure 5(d)).

Figure 6 then gives the average proportions of the four states in different file types. The results show that the proportion of the state '00' (resp '01') is obviously larger than that of the state '11' (resp. '10') across all file types. This finding will be utilized to demonstrate that the reliability of the stored data is further strengthened under the proposed BitFlip (Section III-D).

As the error-prone states predominate in the state proportions, it raises a critical concern about the reliability of the stored data, especially when RBER dramatically climbs with the aging process of the flash memory.

Motivation: Our motivation is that we can convert the error-prone states to other states that are relatively stable (e.g., the states '10' and '11') via bit flips and therefore reduce the



(a) Game files.

(b) Image files.

(c) Multimedia files.

(d) Executable files.

Fig. 5. Analysis on the proportion of the error-prone states for different file types.

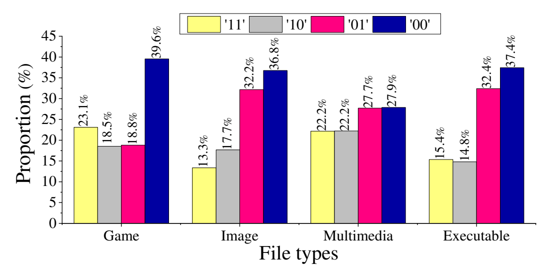


Fig. 6. Proportions of the four states across the four file types.

proportion of the error-prone states. For example, the original eight bits to represent the character 'a' is "01 00 00 01", which are represented via four error-prone states. By using the bit flips, we can change the eight bits to "10 11 11 10", thereby avoiding the error-prone states.

Reducing the number of the error-prone states can bring forth two advances. On one hand, as the error-prone states are the major cause of the RBER, we can suppress the RBER once the number of the error-prone states is reduced, thereby favoring the data reliability. On the other hand, we can decrease the number of decoding levels for data correction in LDPC decoding and hence reduce the read latency.

III. DESIGN OF BITFLIP

We now present the design of BitFlip, which is built on two modules: *bit counter module* and *reorganization module*. Figure 7 presents the architecture of BitFlip, which is realized in the *flash translation layer* (FTL) [25], [26].

We summarize the workflow of BitFlip as follows. Before writing data to the flash memory, the bit counter module and the reorganization module will examine the state proportions

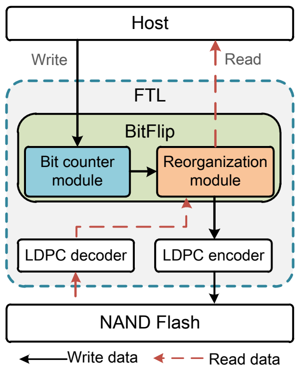


Fig. 7. The architecture of BitFlip with two modules: the bit counter module and the reorganization module.

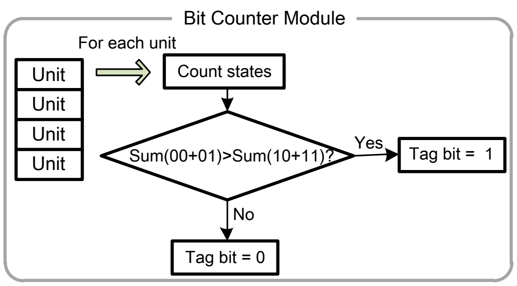


Fig. 8. The process of the bit counter module.

Algorithm 1: Write procedures in BitFlip.

```
Input  : Data from the host {D_1, D_2, ..., D_u}, where u is
         the number of units.
Output: Data flushed to the flash memory
         {R_1, R_2, ..., R_u}.
/* Functionality of the bit counter module:
   checking the error-prone states                    */
1  for i ← 1 to u do
2      Get the numbers of four states of D_i represented by
        N_00, N_01, N_10, and N_11
3      if N_00 + N_01 > N_10 + N_11 then
4          Set the tag bit t_i as 1
5      else
6          Set the tag bit t_i as 0
/* Functionality of the reorganization module:
   flipping the bits for the error-prone units
   */
7  for i ← 1 to u do
8      if t_i == 1 then
9          Generate a unit F_i with all '1's (i.e., {11···1})
10         R_i = D_i ⊕ F_i // ⊕ means XOR operation
11     else
12         R_i = D_i
13 Save tag bits {t_1, t_2, ..., t_u} in the OOB area
14 return {R_1, R_2, ..., R_u}
```

of the data (Section III-A) and flip the bits once the error-prone states in the data are dominant (Section III-B). When reading data from the flash memory, the reorganization module will restore the data and return them to the host (Section III-C). We finally provide in-depth analysis in terms of storage overhead and data reliability for BitFlip (Section III-D).

A. Bit Counter Module

The bit counter module is responsible for analyzing the incoming data and counting the number of four states for deciding whether to perform the bit flip operations. Specifically, to reduce the metadata overhead, the bit counter module suggests analyzing and flipping the data at the unit-level. In this paper, a *unit* is actually a piece of data information, whose size can be tunable to conform with the system requirement.

Figure 8 shows the workflow of the bit counter module and Algorithm 1 elaborates the detailed write procedures (steps 1-6). Given a data chunk to be written, the bit counter module first divides the data chunk into equal-sized units. For each unit, the bit counter module calculates the number of the four states (i.e., '00', '01', '10', and '11') (steps 1-2 in Algorithm 1). If the error-prone states occupy the mainstream position (i.e., more than the states '10' and '11' which are relatively stable) in this unit, we term this unit as an *error-prone unit* (step 3) and suggest flipping the bits in this unit. To facilitate the restoration of the flipped unit in future reads, the bit counter module generates a tag bit with the value of '1' for this unit, indicating that this unit should be flipped before writing to the flash memory (step 4). On the other hand, if the error-prone states are less than other states in a unit, the bit counter module will generate a tag bit with the value of '0' for this unit conversely, representing that the data of this unit should be unchanged in the write operation (step 5).

Consequently, the bit counter module can bring forth the following three advantages. First, by flipping the error-prone states for the error-prone units, the bit counter module can ensure that the proportion of the error-prone states in any unit will be no more than 50%. We also measure the number of error-prone states that can be reduced via bit flips in the experiment evaluation (Figure 11(b) in Section IV-A).

Second, the bit counter module merely needs eight tag bits for each 4-KB data chunk, if assuming that the unit size is 512 Bytes, to record the flip status of the data, thereby introducing marginal storage overhead (about 0.02% of the data to be stored in the flash memory). In realization, the additional tag bits can be stored in the *out-of-band* (OOB) area of the flash memory, and the unit size can be tuned as well to conform with the storage requirement of practical applications. We also uncover the intrinsic connection between the unit size and the introduced storage overhead in Section III-D (Figure 10).

Third, the bit flips can be easily implemented by using basic hardware circuits, demonstrating that BitFlip has good technical feasibility. The XOR operations in hardware can be quickly performed, thereby adding marginal impact on the write latency (Section IV-D).

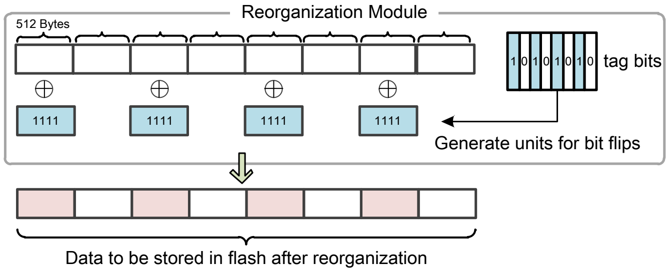


Fig. 9. An example about the workflow of the reorganization module.

Algorithm 2: Read procedures in BitFlip.

Input : Data stored in the flash memory $\{R_1, R_2, \cdots, R_u\}$, where u is the number of units.
Output: Data returned to the host $\{D_1, D_2, \cdots, D_u\}$.

1 Read tag bits $\{t_1, t_2, \cdots, t_u\}$ from the OOB area
2 Fetch the data $\{R_1, R_2, \cdots, R_u\}$ from the LDPC decoder
3 **for** $i \leftarrow 1$ *to* u **do**
4   **if** $t_i == 1$ **then**
5   Generate a unit F_i with all '1's (i.e., $\{11 \cdots 1\}$
6         $D_i = R_i \oplus F_i$
7   **else**
8         $D_i = R_i$
9 **return** $\{D_1, D_2, \cdots, D_u\}$

B. Reorganization Module

After checking the error proneness and establishing the tag bits for each unit, the reorganization module operates the units based on the corresponding tag bits before flushing them into the flash memory. Algorithm 1 elaborates the detailed procedures of the reorganization module (steps 7-14). Specifically, for each unit D_i (where $1 \leq i \leq u$ and u is the number of units), if the corresponding tag bit is '1' (i.e., D_i is an error-prone unit), the reorganization module generates a unit F_i with all '1's, and calculates the resulting unit R_i, where $R_i = D_i \oplus F_i$ (steps 7-10 in Algorithm 1). This operation guarantees that each bit in D_i is flipped. On the other hand, if the corresponding tag bit is '0' (i.e., D_i is not an error-prone unit), the reorganization module will generate $R_i = D_i$, indicating that all the bits in D_i is unchanged (step 12). Finally, the reorganization module writes all the tag bits to the OOB area and forwards $\{R_1, R_2, \cdots, R_u\}$ to the LDPC encoder for data encoding (steps 13-14). The data will be flushed into the flash memory after being encoded.

Figure 9 shows an example about the workflow of the reorganization module. We assume that the unit size is 512 Bytes and therefore a 4-KB data chunk can be partitioned into eight units. Suppose that the associated tag bits of the eight resulting units are "10101010", indicating that the 1st, the 3rd, the 5th, and the 7th units are all error-prone units. Therefore, the reorganization module flips the data of these four units and obtains the resulting data.

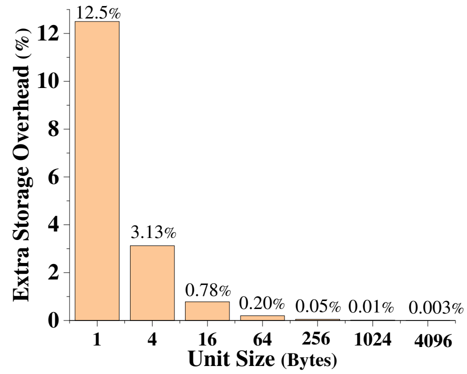


Fig. 10. Analysis on the storage overhead.

C. Procedures of Read Operations

Since the data stored in the flash memory are selectively changed by the reorganization module, BitFlip needs to restore the data when serving the read request. Figure 7 depicts the workflow to serve the read operation and Algorithm 2 elaborates the detailed read procedures. More specifically, when a read request arrives, the requested data will first be fetched from the flash memory for data decoding and delivered to the reorganization module subsequently. At the same time, the reorganization module also reads the corresponding tag bits from the OOB area (step 1 in Algorithm 2) and checks for each unit if the data were flipped before being written to the flash memory. If the data were flipped before, the reorganization module will restore the original data by flipping back the read data (steps 4-6); otherwise, the reorganization module will make the data unchanged (step 8). Finally, the reorganization module returns the requested data to the host (step 9).

We argue that the restoration adds negligible latency on the I/O flow, as the bit flip operations can be fast executed via hardware circuits. On the contrary, by decreasing the number of error-prone states, BitFlip can suppress the RBER and therefore reduce the decoding latency of LDPC code. The performance evaluation in Section IV-B demonstrates the read performance improvement gained by BitFlip.

D. Analysis

Storage overhead: We first analyze the storage overhead induced by BitFlip. Suppose that the unit size is x Bytes. As BitFlip requires to keep a tag bit for each unit, the storage overhead can be simply calculated as $\frac{1}{8 \cdot x}$.

Figure 10 depicts the simulation results of the additional storage overhead versus the unit size, indicating that the additional storage overhead induced by BitFlip vastly drops with the increase of the unit size. For example, when the unit size is 4 KB, the storage overhead is merely 0.003% of the stored data in the flash memory. The unit size can also be tunable to make the storage overhead meet the system requirement.

Reliability: We also demonstrate theoretically that BitFlip can indeed improve the reliability. As BitFlip only changes the

TABLE III
CONFIGURATIONS OF THE SSD SYSTEM.

Channels	18	Chips Per Channel	4
Dies Per Chips	2	Planes Per Die	2
Blocks Per Plane	4,096	Pages Per Block	64
Flash Page Size	4KB	Time of Writing a Page	660us

TABLE IV
SELECTED FILES FOR EVALUATION.

File Type	File information
Game	Kerbal Space Program, Onigiri, Paunch, Robocraft, Sniper Fury, War Thunder, Destiny 2, Dark Deception, Star Conflict, Kaki Raid
Image	Linux (version: 1.1.13, 1.2.12, 1.3.12, 2.0.10, 2.2.21, 2.3.13, 2.4.19, 2.6.12, 3.0.11, 5.0.7)
Multimedia	Photos: KITTI [27] MP4: Vienna New Year Concert (2017∼2019) (Bit rate: 128/192 Kbps)
Executable	Executable files (e.g., bash, tar) in the Linux kernel (5.3.0)

data content for the error-prone units, our objective is to show that the reliability of these error-prone units is improved under BitFlip. We use N_{ij} and P_{ij} to denote the number of the state 'ij' and the probability of suffering the retention errors for the state 'ij', respectively, where 'ij'∈{'00', '01', '10', '11'}. Therefore, the probability that the error-prone units do not have any retention error without BitFlip can be calculated as:

$$P = (1-P_{00})^{N_{00}} \cdot (1-P_{01})^{N_{01}} \cdot (1-P_{10})^{N_{10}} \cdot (1-P_{11})^{N_{11}}.$$

Under BitFlip, the data in the error-prone units are flipped. For example, the original state '00' will be converted to the state '11', and therefore, the number of the state '11' after bit flips is N_{00}. Hence, the new probability that the error-prone units do not induce any retention error under BitFlip can be given by:

$$P_{\text{BitFlip}} = (1-P_{11})^{N_{00}} \cdot (1-P_{10})^{N_{01}} \cdot (1-P_{01})^{N_{10}} \cdot (1-P_{00})^{N_{11}}.$$

Therefore, we can obtain the *reliability ratio* after using BitFlip as:

$$r = \frac{P_{\text{BitFlip}}}{P} = (\frac{1-P_{11}}{1-P_{00}})^{N_{00}-N_{11}} \cdot (\frac{1-P_{10}}{1-P_{01}})^{N_{01}-N_{10}}.$$

Existing studies [2], [19] have shown that the states '00' and '01' are much easier to suffer retention errors than the states '11' and '10', respectively. Then we can have $P_{00} > P_{11}$ and $P_{01} > P_{10}$. In addition, our analysis in Section II-C also unveils that conditions ($N_{00} > N_{11}$ and $N_{01} > N_{10}$) establish for all the file types (Figure 6). Therefore, we can deduce that $r > 1$, indicating that BitFlip can improve the reliability for the error-prone units to resist retention errors.

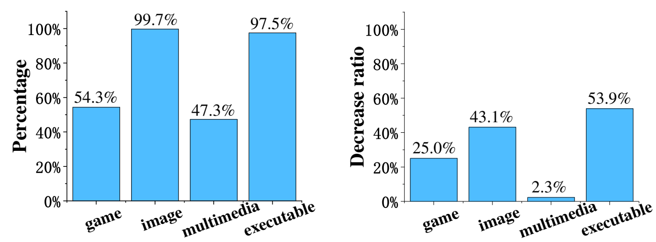
IV. PERFORMANCE EVALUATION

We carry out extensive experiments to evaluate the performance of BitFlip, with the expectation of looking for the answers of the following questions.

- How many error-prone units can be reduced by BitFlip? (Section IV-A)
- How much read performance can be improved by BitFlip? (Section IV-B)
- How much reliability improvement can be gained by BitFlip? (Section IV-C)
- Will BitFlip affect the write performance? (Section IV-D)

TABLE V
CHARACTERISTICS OF SELECTED TRACES.

Workload	Size of read data (GB)	Read ratio
proj_3	18.23	87.41%
web_1	3.81	85.45%
web_0	17.35	59.78%
hm_0	9.95	32.73%
mds_0	7.37	30.65%
wdev_0	7.15	27.80%
src2_0	1.37	12.76%
rsrch_0	1.39	12%



(a) Proportion of error-prone units. (b) Reduction proportion.

Fig. 11. Reduction on error-prone units.

Experimental setup: We implement BitFlip in the trace-driven flash simulator SSDsim [13]. The configurations are summarized in Table III. We first set up an SSD storage system with the capacity of 288 GB constructed over 18 channels. Inside each channel, there are 16 planes, where each plane is supposed to contain 4,096 blocks. Each block is composed of 64 pages, where the page size is set as 4 KB.

In the LDPC decoding, we set the latency of the hard-decision decoding as 85 *us*. We assume that the soft-decision decoding has six decoding levels and each level calls for the time of 24 *us*. These configurations are the same as those in the previous work [3]. Table I shows the accumulated latency of each level in the LDPC decoding.

Selection of test files: To demonstrate the practicality and generality of BitFlip, we conduct evaluation with four different file types, including game files, image files, multimedia files, and executable files. Table IV lists the file types as well as the corresponding files used throughout the evaluation. Specifically, we choose the installation files (including audio, video, and executable files) from ten famous games and the ten image files of the released Linux, whose versions vary from 1.1.13 to 5.0.7. The multimedia files selected include the photos and the MP4 files, and the executable files are the tools frequently used in the Linux kernel (version 5.3.0).

Selection of traces: To evaluate the read latency under BitFlip, we select eight real-world traces from MSR Cambridge Traces [28], which record various access characteristics from enter-

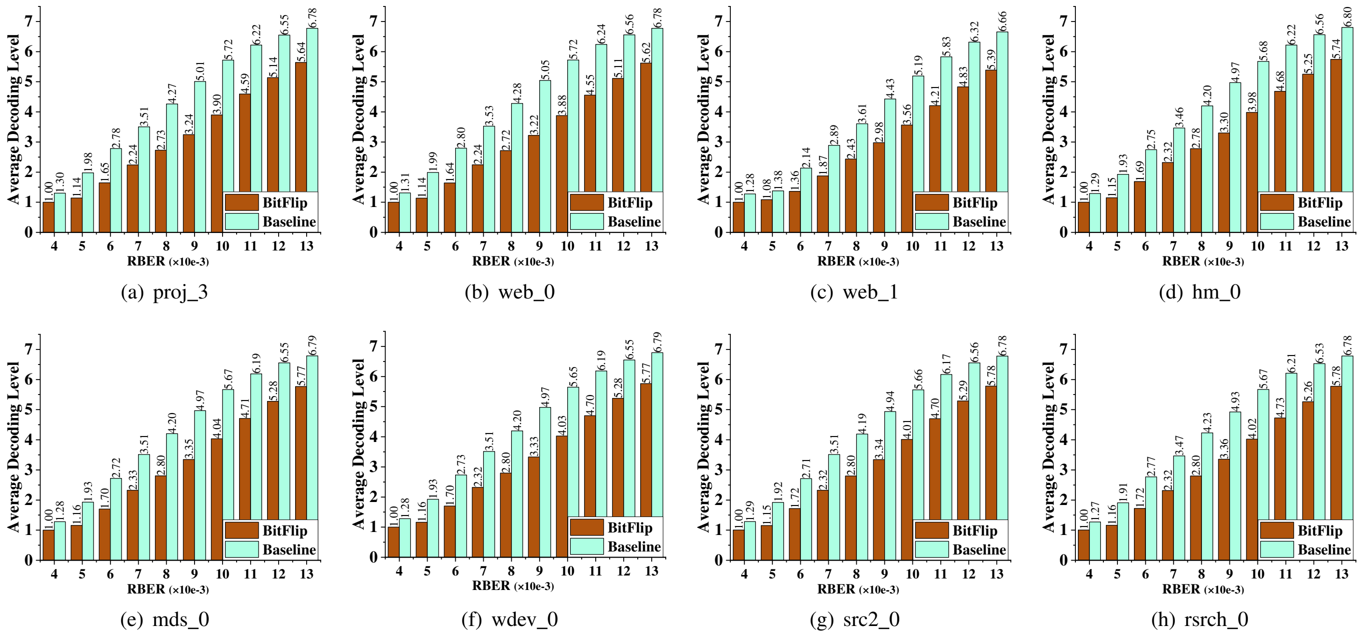


Fig. 12. Comparison on the decoding levels. The smaller value is better.

prise storage systems. Specifically, the MSR Cambridge Traces collect data access operations from 36 volumes constructed over 179 disks of 13 servers for one week. Each trace records the attributes of the access operations, including the I/O type (e.g., read or write), the starting position of the I/O operation, as well as the requested size. The eight traces selected have dramatically different *read ratios* (calculated by dividing the size of the read data by that of all the accessed data) and therefore can well probe the overall performance of BitFlip.

Evaluation methods: In the evaluation, we mainly compare BitFlip with a baseline approach, which is the state-of-the-art LDPC implementation in flash-based SSDs [3] without considering bit flips. Given a trace, the simulator consecutively extracts every access request from it and pinpoints the physical pages that are about to be accessed in this request. We range the RBER from 4×10^{-3} to 13×10^{-3}, ensuring that all the decoding levels in both hard-decision and soft-decision decodings can be tested. We then measure the elapsed latency to complete an access request, from the time of issuing the request to the time of receiving the response after completion.

A. Reduction on Error-Prone States

In this evaluation, we measure the reduction of the error-prone states gained by BitFlip for different file types. We first analyze the four file types and examine the proportion of the error-prone units that can be found by BitFlip. We set the unit size as 512 KB and show the results in Figure 11(a). We can observe that the error-prone units widely exist in real-world files and occupy the majority of the units in most of the file types. For all the four file types, the proportions of the error-prone units vary from 47.3% to 99.7%.

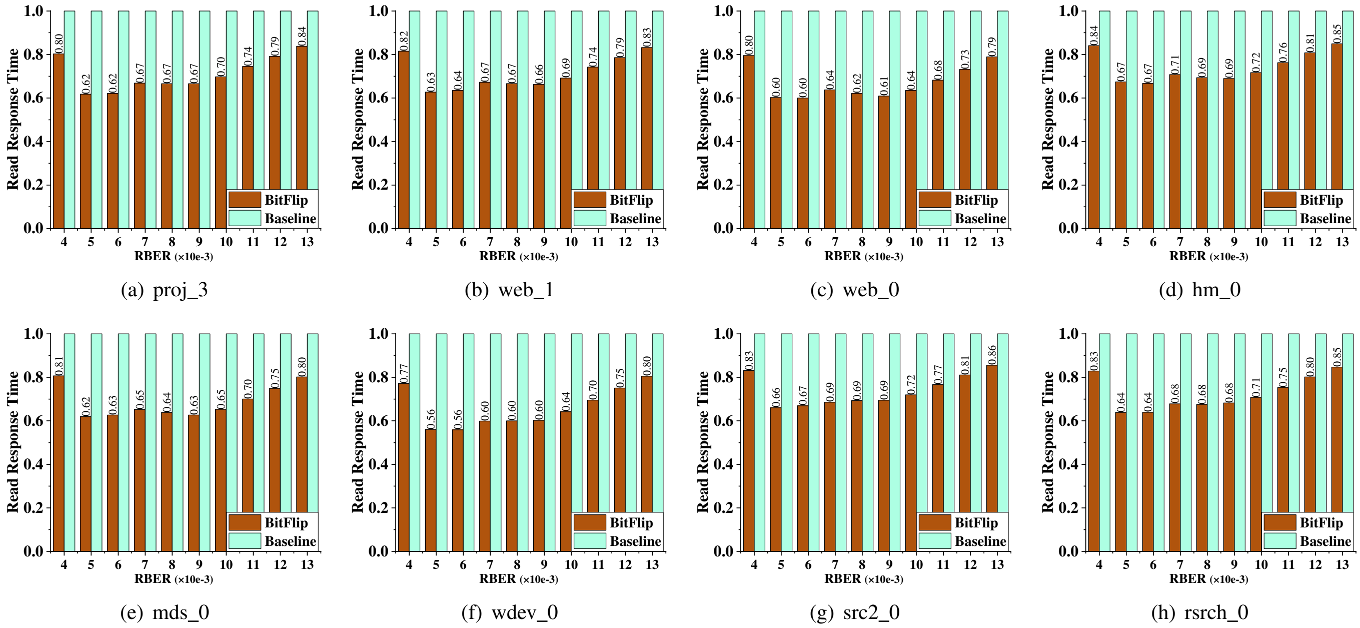
We can identify that the proportions of the error-prone states in different file types vary dramatically. More particular, the proportions of Game files and Multimedia files close to 50% (i.e., 54.3%, and 47.3%, respectively), as both file types are compressed through some special algorithms. However, the Image file type has a sizeable error-prone proportion, and we suspect the reason is that the Image files are not compressed.

We then measure the reduction of the error-prone states after applying BitFlip and show the results in Figure 11(b). Overall, BitFlip can reduce about 2.3%-53.9% of the error-prone states for different file types, thereby demonstrating the effectiveness of BitFlip. Specifically, BitFlip can eliminate at most 53.9% of the error-prone states for the executable files. In addition, BitFlip achieves the minimum reduction (i.e., 2.3%) for the multimedia files. We surmise that the encoding method of the multimedia files makes the state proportions much more balanced even in the error-prone units. Therefore, even using bit flipping for the multimedia files, the proportion of the error-prone states does not change significantly.

B. Read Performance

We first compare BitFlip and the baseline approach on the read performance under different RBERs. We replay the read operations of each trace to the four file types, ensuring that each file type receives the same amount of the requested data. This evaluation can mimic the daily operation and therefore can evaluate the overall read performance of BitFlip and the baseline approach in daily life.

We then measure the read performance on the two metrics, i.e., the number of decoding levels (including the hard-decision decoding) needed in a read operation, and the corresponding read latency. The first metric can indirectly reflect the decoding



(a) proj_3 (b) web_1 (c) web_0 (d) hm_0

(e) mds_0 (f) wdev_0 (g) src2_0 (h) rsrch_0

Fig. 13. Comparison on the read latency. The smaller value is better.

time, which is closely related to the used decoding levels (Table I). The results are shown in Figure 12 and Figure 13, respectively.

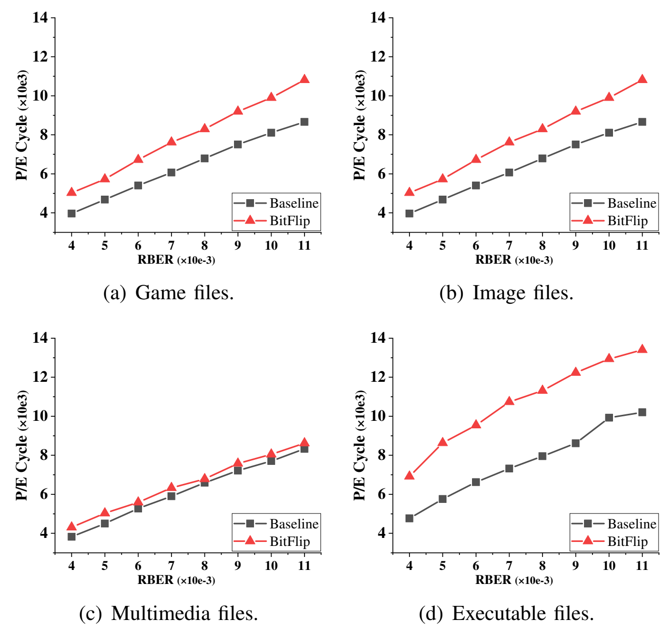
Decoding level: Figure 12 shows the average decoding levels required under different RBERs, where BitFlip can reduce 27.1%-31.6% of the decoding levels on average. The reason is that BitFlip is effective on suppressing the probability of the retention errors by reducing the error-prone states. Besides, with the RBER increases, the number of decoding levels reduced by BitFlip sharply increases at the beginning and then gradually decreased.

Read latency: Figure 13 shows the average read latencies under different RBERs. We make the following two findings. First, BitFlip can reduce the read latency by 25.9%-34.2% for each trace compared with the baseline approach. The reason is that by reducing the error-prone states, BitFlip can significantly reduce the decoding time needed in read operations.

Second, with the RBER increases, the reduction of the read latency gained by BitFlip first sharply increases and then gradually shrinks. More particularly, when the RBER ranges from 0.005 to 0.009, BitFlip can gain the most reduction on the read latency. The reason is that when the RBER climbs, it needs more decoding levels to read the data and therefore incurs higher accumulated decoding latency. As the number of decoding levels reduced by BitFlip shrinks when the RBER is extremely higher, the reduction of the read latency narrows as well.

C. Reliability

We also assess the reliability improvement gained by BitFlip through measuring the numbers of P/E cycles that BitFlip and



(a) Game files. (b) Image files.

(c) Multimedia files. (d) Executable files.

Fig. 14. Comparison on the number of P/E cycles that can be endured. The larger value is better.

the baseline approach can endure before emerging the same RBER, respectively. We replay the eight selected traces to the files of each file type, and recording the resulting RBERs. Given a RBER, we can speculate the P/E cycles endured via the results in a previous work (the one-week column of Table 4 in [9]), which unveils the intrinsic connections among the rate of the retention error, the retention time, and the P/E cycles. The results are shown in Figure 14.

From Figure 14, we can make two findings. First, BitFlip can markedly slow down the aging process of the flash memory, as the threshold number of P/E cycles to emerge the same RBER is significantly increased under BitFlip. More specifically, BitFlip can increase 2.9%-33.3% of P/E cycles that the flash memory can endure, indicating that BitFlip can to some extend prolong the lifespan of the flash memory when compared with the baseline approach. The rational lies in that BitFlip can effectively suppress the probability of raw bit errors by reducing the number of error-prone states, and hence slow down the rising trend of the RBER.

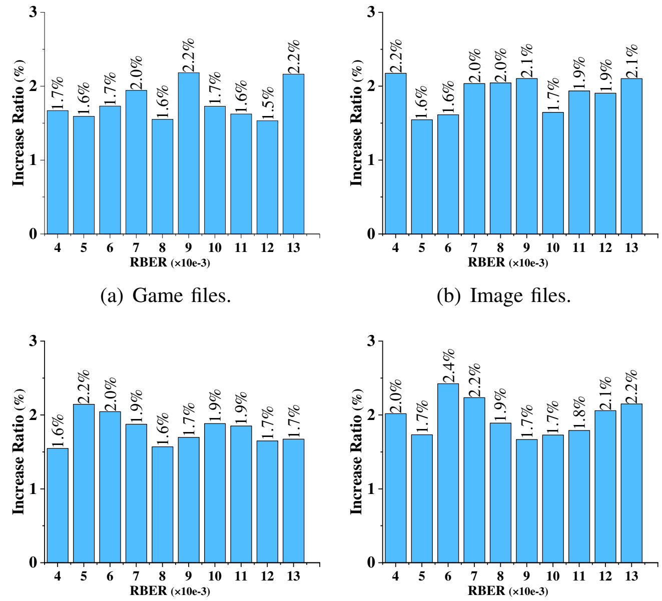
Second, the reliability improvement introduced by BitFlip varies across different file types. For example, BitFlip can increase 2.9% of the threshold number of P/E cycles on average for the multimedia files. The improvement increases to 29.2% on average for the executable files. The reason is that the proportions of error-prone states that can be reduced by BitFlip are dramatically varied due to the different characteristics across file types (see Figure 11(b)).

D. Write Performance

We finally evaluate the write performance of BitFlip. We extract the write operations from the eight selected traces and replay them to the data files of the four file types. We evaluate the write latencies of BitFlip and the baseline approach, which are denoted by w_{BitFlip} and w_{baseline}, respectively. We then calculate the *increase ratio* of the write latency under BitFlip as:

$$\text{increase ratio} = \frac{w_{\text{BitFlip}}}{w_{\text{baseline}}} - 1.$$

The results of the increase ratio evaluated are shown in Figure 15. We can observe that BitFlip adds negligible impact on the write latency, typically ranging from 1.6% to 2.4%. This is because BitFlip only requires to perform some additional XOR operations for bit flipping, which can be realized extremely fast by using hardware circuits. In summary, it is more appropriate to deploy BitFlip in the applications with the access characteristics of intensive reads and infrequent writes.



(a) Game files. (b) Image files.

(c) Multimedia files. (d) Executable files.

Fig. 15. The increase ratio of the write latency under BitFlip.

V. Related Work

We review the efforts to reduce the read latency from the following three aspects: i) improving the sensing accuracy, ii) utilizing the error patterns, and iii) reducing the sensing time.

Improving the sensing accuracy: Some studies seeked for improving the sensing accuracy of the raw data in flash, such that the read latency can be cut down by using fewer decoding levels in LDPC code. By caching detected errors, EC-Cache [6] corrected errors of the requested page before performing the LDPC decoding process, thereby increasing the sensing accuracy when the decoding starts. Li *et al.* [7] developed a read data placement scheme by exploiting the reliability characteristics of different blocks. There are also some techniques for refreshing data to improve the read performance. Lv *et al.* [29] put forward two refresh schemes for long access latency data to reduce the tail latency. LDR [4] aggressively corrects errors in the read-hot pages with long read latency and refreshes the corrected data in new pages to reduce the latency in subsequent reads. As a comparison, BitFlip suppresses the error proneness and improves the sensing accuracy via bit flips.

Utilizing the error patterns: Some studies proposed to design read-optimized techniques based on the error patterns of flash memory. By observing the different decoding latency caused by the unbalanced RBER between the least significant bit (LSB) and the most significant bit (MSB) pages, Zhang *et al.* [19] designed a cooperative error correction scheme that makes use of the decoding result of the LSB pages in the decoding process of the MSB pages to improve the read performance. Zhang *et al.* [30] noticed that the errors of LSB pages may result in the errors of MSB pages, and consequently proposed to supply promotion information in the decoding process to reduce the decoding latency. Different from previous work, BitFlip utilizes error proneness of the states in designing the bit-flipping technique.

Reducing the sensing time: Some studies proposed to improve the read performance by reducing the sensing time of the reference voltage. FlexLevel [9] reduces *bit error rate* (BER) via enlarging the noise margins by threshold voltage level reduction. By observing that the read level of a page may stay constant for a long time, LaLDPC [12] proposes to estimate the read voltage level based on the reference voltage used in the last read on the same page, so as to reduce the number of re-sensing operation. Instead of reading repeatedly, EP-LDPC scheme [10] reduces the read latency by estimating errors from a variety of factors, such as reference voltage, write/read cycles, data-retention time, and inter-cell coupling information. Du *et al.* [11] designed a Multi-Granularity LDPC read method, which applies multiple read-level-increment granularity to adapt the variations of layer RBERs, so as to reduce the unnecessary re-sensing trials in traditional single read-level-increment granularity. As an orthogonal study, BitFlip mainly focuses on lowering down the RBER in data decoding for improving the read performance.

VI. Conclusion

This paper explores the opportunities for utilizing the differences of the four states in both error-proneness and proportion to improve the read performance and reliability of flash memory. We design BitFlip, a novel bit-flipping technique. The core idea of BitFlip is to examine the state proportions of the data units and flip the bits for the error-prone units to make them more stable against retention errors. We conduct extensive experiments with real-world traces and files, showing that BitFlip can reduce 25.9%-34.2% of the read latency and prolong 2.9%-33.3% of the lifespan for flash memory, while adding negligible impact on the write latency.

VII. acknowledgement

This work was supported in part by the National Natural Science Foundation of China under Grant U1705261, Grant 61972325, and Grant 61872305, in part by U.S. NSF under Grant CCF-1704504 and Grant CCF-1629625.

Reference

[1] Roberto Bez, Emilio Camerlenghi, Alberto Modelli, and Angelo Visconti. Introduction to flash memory. *Proceedings of the IEEE*, 91(4):489–502, 2003.

[2] Yu Cai, Erich F Haratsch, Onur Mutlu, and Ken Mai. Error patterns in mlc nand flash memory: Measurement, characterization, and analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 521–526. EDA Consortium, 2012.

[3] Kai Zhao, Wenzhe Zhao, Hongbin Sun, Xiaodong Zhang, Nanning Zheng, and Tong Zhang. Ldpc-in-ssd: Making advanced error correction codes work effectively in solid state drives. In *the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 243–256, 2013.

[4] Yajuan Du, Qiao Li, Liang Shi, Deqing Zou, Hai Jin, and Chun Jason Xue. Reducing ldpc soft sensing latency by lightweight data refresh for flash read performance improvement. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2017.

[5] M. Zhang, F. Wu, Y. Du, C. Yang, C. Xie, and J. Wan. Cooecc: A cooperative error correction scheme to reduce ldpc decoding latency in nand flash. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 657–664, 2017.

[6] Ren-Shuo Liu, Meng-Yen Chuang, Chia-Lin Yang, Cheng-Hsuan Li, Kin-Chu Ho, and Hsiang-Pang Li. Ec-cache: Exploiting error locality to optimize ldpc in nand flash-based ssds. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2014.

[7] Qiao Li, Liang Shi, Yejia Di, Yajuan Du, Chun J Xue, and HM Edwin. Exploiting process variation for read performance improvement on ldpc based flash memory storage systems. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 681–684. IEEE, 2017.

[8] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A large-scale study of flash memory failures in the field. *ACM SIGMETRICS Performance Evaluation Review*, 43(1):177–190, 2015.

[9] Jie Guo, Wujie Wen, Jingtong Hu, Danghui Wang, Hai Li, and Yiran Chen. Flexlevel: a novel nand flash storage system design for ldpc latency reduction. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2015.

[10] Shuhei Tanakamaru, Yuki Yanagihara, and Ken Takeuchi. Error-prediction ldpc and error-recovery schemes for highly reliable solid-state drives (ssds). *IEEE Journal of Solid-State Circuits*, 48(11):2920–2933, 2013.

[11] Yajuan Du, Yao Zhou, Meng Zhang, Wei Liu, and Shengwu Xiong. Adapting layer rbers variations of 3d flash memories via multi-granularity progressive ldpc reading. In *Proceedings of the 56th Annual Design Automation Conference 2019*, page 37. ACM, 2019.

[12] Yajuan Du, Deqing Zou, Qiao Li, Liang Shi, Hai Jin, and Chun Jason Xue. Laldpc: Latency-aware ldpc for read performance improvement of solid state drives. In *MSST*, pages 1–13, 2017.

[13] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Shuping Zhang. Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of the international conference on Supercomputing*, pages 96–107. ACM, 2011.

[14] Fei Li, Youyou Lu, Zhongjie Wu, and Jiwu Shu. Ascache: An approximate ssd cache for error-tolerant applications. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[15] Guiqiang Dong, Shu Li, and Tong Zhang. Using data postcompensation and predistortion to tolerate cell-to-cell interference in mlc nand flash memory. *IEEE Transactions on Circuits and Systems*, 57(10):2718–2728, 2010.

[16] Yu Cai, Erich F Haratsch, Onur Mutlu, and Ken Mai. Threshold voltage distribution in mlc nand flash memory: Characterization, analysis, and modeling. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1285–1290. EDA Consortium, 2013.

[17] Neal Mielke, Todd Marquart, Ning Wu, Jeff Kessenich, Hanmant Belgal, Eric Schares, Falgun Trivedi, Evan Goodness, and Leland R Nevill. Bit error rate in nand flash memories. In *2008 IEEE International Reliability Physics Symposium*, pages 9–19. IEEE, 2008.

[18] Robert Gallager. Low-density parity-check codes. *IRE Transactions on information theory*, 8(1):21–28, 1962.

[19] Meng Zhang, Fei Wu, Yajuan Du, Chengmo Yang, Changsheng Xie, and Jiguang Wan. CooECC: A Cooperative Error Correction Scheme to Reduce LDPC Decoding Latency in NAND Flash. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 657–664. IEEE, 2017.

[20] Yu Cai, Saugata Ghose, Erich F Haratsch, Yixin Luo, and Onur Mutlu. Error characterization, mitigation, and recovery in flash-memory-based solid-state drives. *Proceedings of the IEEE*, 105(9):1666–1704, 2017.

[21] New York Times. The Supreme Court's Final Exam. https://www.nytimes.com/2019/12/19/opinion/sunday/trump-tax-supreme-court.html.

[22] New York Times. He Was One of Mexico's Deadliest Assassins. Then He Turned on His Cartel. https://www.nytimes.com/2019/12/14/world/americas/sicario-mexico-drug-cartels.html.

[23] New York Times. Trump Impeached for Abuse of Power and Obstruction of Congress. https://www.nytimes.com/2019/12/18/us/politics/trump-impeached.html.

[24] Sangyeun Cho and Hyunjin Lee. Flip-n-write: A simple deterministic technique to improve pram write performance, energy and endurance. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 347–357, 2009.

[25] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. A survey of flash translation layer. *Journal of Systems Architecture*, 55(5-6):332–343, 2009.

[26] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings. *Acm Sigplan Notices*, 44(3):229–240, 2009.

[27] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *The International Journal of Robotics Research*, 32(11):1231–1237, 2013.

[28] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):10, 2008.

[29] Yina Lv, Liang Shi, Qiao Li, Congming Gao, Chun Jason Xue, and Edwin Sha. Optimizing tail latency of ldpc based flash memory storage systems via smart refresh. In *2019 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–8. IEEE, 2019.

[30] M. Zhang, F. Wu, Y. Du, W. Liu, and C. Xie. Pair-bit errors aware ldpc decoding in mlc nand flash memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(12):2312–2320, Dec 2019.