IMPROVING RELIABILITY AND PERFORMANCE OF NAND FLASH BASED STORAGE SYSTEM

by

Jie Guo

B.S. in Electrical Engineering,

University of Electronic Science and Technology of China, China,

2005

M.S. in Electrical Engineering,

University of Electronic Science and Technology of China, China,

2008

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2016

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This dissertation was presented

by

Jie Guo

It was defended on

February 12th, 2016

and approved by

Yiran Chen, Ph.D., Associate Professor, Department of Electrical and Computer Engineering

Panos K. Chrysanthis, Ph.D., Professor, Department of Computer Science

Amro El-Jaroudi, Ph.D., Associate Professor, Department of Electrical and Computer Engineering

Hai Li, Ph.D., Associate Professor, Department of Electrical and Computer Engineering

Kartik Mohanram, Ph.D., Associate Professor, Department of Computer Science

Dissertation Director: Yiran Chen, Ph.D., Associate Professor, Department of Electrical and Computer Engineering

IMPROVING RELIABILITY AND PERFORMANCE OF NAND FLASH BASED STORAGE SYSTEM

Jie Guo, PhD

University of Pittsburgh, 2016

High seek and rotation overhead of magnetic hard disk drive (HDD) motivates development of storage devices, which can offer good random performance. As an alternative technology, NAND flash memory demonstrates low power consumption, microsecond-order access latency and good scalability. Thanks to these advantages, NAND flash based solid state disks (SSD) show many promising applications in enterprise servers. With multi-level cell (MLC) technique, the per-bit fabrication cost is reduced and low production cost enables NAND flash memory to extend its application to the consumer electronics.

Despite these advantages, limited memory endurance, long data protection latency and write amplification continue to be the major challenges in the designs of NAND flash storage systems. The limited memory endurance and long data protection latency issue derive from memory bit errors. High bit error rate (BER) severely impairs data integrity and reduces memory durance. The limited endurance is a major obstacle to apply NAND flash memory to the application with high reliability requirement. To protect data integrity, hard-decision error correction codes (ECC) such as Bose-Chaudhuri-Hocquenghem (BCH) are employed. However, the hardware cost becomes prohibitively with the increase of BER when the BCH ECC is employed to extend system lifetime. To extend system lifespan without high hardware cost, we has proposed data pattern aware (DPA) error prevention system design. DPA realizes BER reduction by minimizing the occurrence of data patterns vulnerable to high BER with simple linear feedback shift register circuits. Experimental results show that DPA can increase the system lifetime by up to 4× with marginal hardware cost.

With the technology node scaling down to 2Xnm, BER increases up to 10^{-2}. Hard-decision ECCs and DPA are no longer applicable to guarantee data integrity due to either prohibitively high hardware cost or high storage overhead. Soft-decision ECC, such as low-density parity-check (LDPC) code, has been introduced to provide more powerful error correction capability. However, LDPC code demands extra memory sensing operations, directly leading to long read latency. To reduce LDPC code induced read latency without adverse impact on system reliability, we has proposed FlexLevel NAND flash storage system design. The FlexLevel design reduces BER by broadening the noise margin via threshold voltage (V_{th}) level reduction. Under relatively low BER, no extra sensing level is required and therefore read performance can be improved. To balance V_{th} level reduction induced capacity loss and the read speedup, the FlexLevel design identifies the data with high LDPC overhead and only performs V_{th} reduction to these data. Experimental results show that compared with the best existing works, the proposed design achieves up to 11% read speedup with negligible capacity loss.

Write amplification is a major cause to performance and endurance degradation of the NAND flash based storage system. In the object-based NAND flash device (ONFD), write amplification partially results from onode partial update and cascading update. Onode partial update only over-writes partial data of a NAND flash page and incurs unnecessary data migration of the un-updated data. Cascading update is update to object metadata in a cascading manner due to object data update or migration. Even through only several bytes in the object metadata are updated, one or more page has to be re-written, significantly degrading write performance. To minimize write operations incurred by onode partial update and cascading update, we has proposed a Data Migration Minimizing (DMM) device design. The DMM device incorporates 1) the multi-level garbage collection technique to minimize the unnecessary data migration of onode partial update and 2) the virtual B+ tree and diff cache to reduce the write operations incurred by cascading update. The experiment results demonstrate that the DMM device can offer up to 20% write reduction compared with the best state-of-art works.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PREFACE

This dissertation is submitted in partial fulfillment of the requirements for Jie Guo's degree of Doctor of Philosophy in Electrical and Computer Engineering. It contains the works done from September 2011 to January 2016. My advisor is Yiran Chen, University of Pittsburgh, 2010 – present.

The work is original to the best of my knowledge, except where acknowledgement and reference are made to the previous work. There is no similar dissertation that has been submitted for any other degree at any other university.

Part of the work has been published in the following conferences:

1. **DAC2015**: **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," Design Automation Conference (DAC), Jun. 2015, pp. 1-6.
2. **ASP-DAC2014**: **J. Guo**, Z. Chen, D. Wang, Z. Shao and Y. Chen, "DPA: A Data Pattern Aware Error Prevention Technique for NAND Flash Lifetime Extension," 19th Asia and South Pacific Design Automation Conference (ASP-DAC), Jan. 2014, pp. 592 - 597.
3. **DATE2013**: **J. Guo**, J. Yang, Y. Zhang and Y. Chen, "Low Cost Power Failure Protection for MLC NAND Flash Storage Systems with PRAM/DRAM Hybrid Buffer," Design, Automation & Test in Europe (DATE), Mar. 2013, pp. 859 - 864.
4. **DATE2013**: **J. Guo**, W. Wen, S. Li, H. Li and Y. Chen, "DA-RAID-5: A Disturb Aware Data Protection Technique for NAND Flash Storage Systems," Design, Automation & Test in Europe (DATE), Mar. 2013, pp. 380-385.

Part of the work has been submitted to the journals and conferences:

1. **TCAD2016**: **J. Guo**, W. Wen, J. Hu, D. Wang, H. Li and Y. Chen, "FlexLevel NAND Flash Storage System Design to Reduce LDPC Latency," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), submitted on Jan. 31st.
2. **MSST2016**: **J. Guo**, C. Min, T. Cai and Y. Chen, "A Design to Reduce Write Amplification in Object-based NAND Flash Devices," International Conference on Massive Storage Systems and Technology (MSST), submitted on Feb. 12th.

ACKNOWLEDGEMENTS

I would like to acknowledge the support of my advisor, Yiran Chen, whose support made this work possible, and to National Science Foundation Project (NSF CCF-1217947, NSF CNS-1116171, NSF CNS-1342566) for directly providing much of the financial support. I'd like to thank Professor Yiran Chen and Professor Hai (Helen) Li for their excellent guidance during the research. Professor Yiran Chen gives me guidance of NAND flash memory designs from device modeling, circuit implementation to architecture simulations and validations. Special thanks go to Professor Panos K. Chrysanthis, Professor Amro El-Jaroudi, Professor Hai (Helen) Li and Professor Kartik Mohanram for being my committee members. I also would like to thank Professor Jingtong Hu from Oklahoma State University and Professor Tao Cai from Jiangsu University, for their guidance and encouragement during my Ph.D. study.

In addition, I'd like to express my gratitude to the members from Evolutional Intelligent (EI) lab at Swanson School of Engineering, especially Wujie Wen and Chuhan Min, for their consistent supports during my research. Finally, I'd like to thank my husband, Zhigang Wang, an associate professor in University of Electronic Science and Technology of China (UESTC) and my parents in China for their great encouragement during the whole Ph.D. research.

1.0 INTRODUCTION

1.1 MOTIVATION

Magnetic hard disk drives (HDD) have been the dominant second storage medium for several decades. However, time-consuming seek and rotation process rooted in HDD operating mechanism has been a major performance bottleneck for the random-write intensive applications. This performance concern motivates many research efforts on NAND flash memory technology. NAND flash memory can offer fast access time, good scalability and low power consumption. Thanks to these advantages, NAND flash memory is applied to various storage systems ranging from low-power embedded systems to high-end servers.

A NAND flash cell is a floating gate transistor with programmable threshold voltage (V_{th}). The stored values are represented by different V_{th} levels. For example, in a single-level cell (SLC), one bit is stored, which is represented by two V_{th} levels. In a multi-level cell (MLC), four V_{th} levels are employed to represent two bits. To store the data into the NAND flash cell, electrons are injected to the floating gate by the program operation. Before the cell is re-programmed, an erase operation has to be performed to remove the electrons from the floating gate. The program and erase operations cause the major reliability and performance problems in NAND flash based storage system which are summarized as follows:

1. **Limited device endurance.** The program/erase (P/E) operations wear out the NAND flash cells and introduce bit error rate (BER). BER gradually increases with P/E cycling and eventually results in cell failure. Due to high BER, NAND flash memory has short endurance: 5000 P/E cycles under 3Xns technology node [1]. Such a limited endurance cannot meet the reliability requirement of enterprise application.

2. **High data protection overhead.** To prevent data corruption under high BER, error correction codes (ECCs) are usually employed in NAND flash based storage systems. Typically, low-density parity-check (LDPC) code is adopted to provide powerful error correction capacity. However, the LDPC code incurs high read overhead, which hinders its application in the read-critical applications.
3. **Write amplification.** The erase-before-write (i.e., *out-of-place update*) memory characteristic introduces more data migration or write operations than requested, which is called write amplification. Write amplification shortens system endurance and incurs write performance degradation.

These three reliability and performance issues are discussed below in Sections 1.1.1, 1.1.2 and 1.1.3, respectively.

1.1.1 Challenge 1: Limited Device Endurance

The endurance issue of NAND flash memory is rooted from the intrinsic device noises. As mentioned above, the stored data is represented by different V_{th} levels. However, in reality, V_{th} levels are not clearly separated: Intrinsic device noise easily fluctuates V_{th}, resulting in overlapping of neighboring V_{th} levels. Previous works identify random telegraph noise (RTN), cell-to-cell interference and retention time limit as three major noise sources in NAND flash cells [2, 3, 4, 5]. RTN causes V_{th} fluctuation by electron capture and emission events at a charge trap site near interfaces. Cell-to-cell interference noise stems from capacitance-coupling. It results in V_{th} increase of one floating gate transistor when its neighboring cells are programmed. Retention time noise causes charges to leak away from the floating gate, leading to V_{th} decrease. Among these three type of noise, RTN and retention time noise rise up as P/E cycle increases, resulting in more severe V_{th} shift.

A direct result of the device noise is bit errors. Due to noise augmentation, these bit errors increase with P/E cycling. To prevent data corruption from bit errors, error correction code (ECC), such as Bose-Chaudhuri-Hocquenghem (BCH) code is usually deployed in NAND flash based storage system. The ECC controls the uncorrectable bit error rate (UBER) of the storage system under an acceptable level, e.g., 10^{-14}. The acceptable P/E

cycles of NAND flash memory, i.e., device endurance, is limited by maximum allowed UBER. Under 3Xnm technology node, the MLC NAND flash memory endurance is reduced to 5000 P/E cycles [1]. The limited endurance of MLC NAND flash memory is acceptable for consumer application but cannot meet the requirement of highly reliable enterprise application. Therefore, increasing device endurance is critical to expand the MLC NAND flash memory to the application with high reliability requirement. One way to extend device endurance is to increase the error correction capacity of BCH ECC. However, enhancement of BCH ECC error correction capacity incurs prohibitively high hardware cost [6].

1.1.2 Challenge 2: High Data Protection Overhead

As mentioned in the previous section above, ECC is employed in the NAND flash based storage system to protect data integrity. The selection of ECC is based on error correction capability. The qualified ECC should guarantee that the UBER of the storage system is under an acceptable level. A widely applied ECC in the NAND flash based storage system is BCH ECC. BCH ECC is hard-decision in nature. It can fast correct bit errors by decoding binary information. However, as the technology node scales down to 2Xnm, BER can reach up to 10^{-2} [7]. Under such a high BER, BCH ECC is no longer applicable since realization of stronger error correction capability requires prohibitively high hardware cost.

To offer more powerful error correction capacity, low-density parity-check (LDPC) code is adopted. LDPC code adopts a sparse M×N parity-check matrix. The matrix is represented by a bipartite graph with N variable nodes and M check nodes. Error correction is realized by iteratively computing error messages, which are exchanged between variable nodes and check nodes [8]. Under low BER, LDPC can work in the similar manner as hard-decision ECC without introducing extra overhead. When BER is high, LDPC needs to work in a soft-decision fashion, which demands log-likelihood-ratio (LLR) information [9] to achieve better error correction capability. In NAND flash memories, the LLR information can only be acquired by extra fine-grain memory sensing operations. More memory sensing levels offer more accurate LLR information and therefore, provide more powerful error correction performance. However, more memory sensing levels cause longer read latency, which severely degraded the system read performance.

1.1.3 Challenge 3: Write Amplification

Write amplification results from the out-of-place update of NAND flash memory [10][11]. With more data being written than requested, write amplification directly results in system endurance reduction and write performance degradation. As the technology node scales down, the memory performance and endurance are continuously degrading: The program latency under 2Xns technology node increases to 3ms [12]; under the sub-20ns technology node, rated memory endurance decreases to 3000 P/E cycles [13]. Hence, reducing write amplification become critical to improve system performance and reliability.

One way to minimize write amplification is to reduce data migration of garbage collection by optimizing the layout of hot and cold data [14][15]. Unfortunately, under the existing block-based storage model, hot and cold data cannot be accurately identified. As shown in Fig. 1, in the block-based model, the file system manages logical blocks and allocates them to the stored file data. The logical block address (LBA) is the only data identifier in the underlying NAND flash device. Unaware of block allocation policies at the file system layer, LBA cannot accurately identify data access patterns [16]. In addition, due to out-of-place update, an indirection table is needed in the NAND flash device to map LBAs to physical addresses, which significantly increases memory consumption [17].

To eliminate this architectural limitation, an object-based storage model is proposed [18]. In this model, the storage management layer is offloaded to the underlying object-based NAND flash device (ONFD) [18, 19]. The ONFD manages data by unit of object instead of by logical blocks. Understanding the object semantics, the ONFD can utilize the object attributes to improve accuracy of data pattern identification [19, 20]. In addition, the ONFD eliminates adoption of the indirection table, simplifying system design and reducing memory consumption.

With object semantics, two causes to write amplification in the ONFD have been identified. One cause is onode partial update. Onode is a data structure in the ONFD to store object attributes [19]. Due to small size, more than one onode is stored in a physical page to reduce internal fragmentation [19]. Update to an onode incurs partial update to a physical page, i.e., onode partial update. Due to the page-unit write in the ONFD [10], onode partial

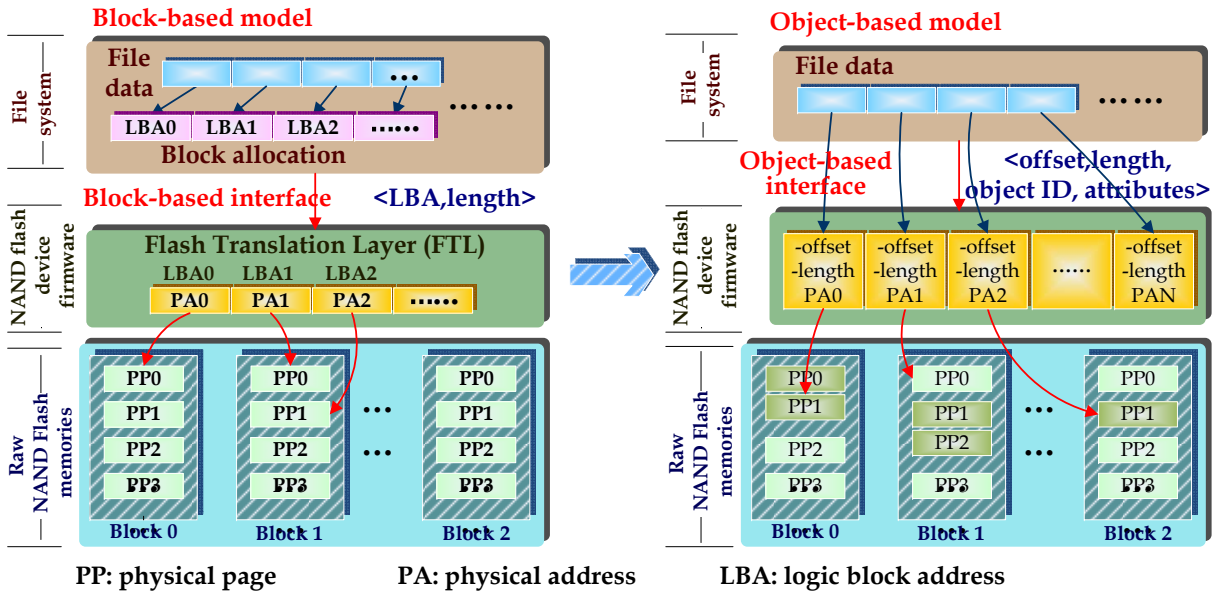


Figure 1: The block-based storage v.s. the object-based storage.

update invokes unnecessary migration of the un-updated data. Another write amplification cause is cascading update. In the ONFD, the physical addresses of object data are maintained in per-object indices; the address of the per-object index root node page is stored in the onode [19]. The per-object index is implemented with extent-based B+ tree [21]. When object data is updated, due to out-of-place update, the wandering tree issue causes the cascading update within a per-object index [22]. In addition, the cascading update changes the address of per-object index root node page. Hence, the corresponding onode are also updated. Despite several bytes update to the object metadata, one or more page have to migrate entirely, causing significant write amplification.

1.2 DISSERTATION CONTRIBUTION AND OUTLINE

In this dissertation, we propose three technologies to handle the design challenges from limited endurance, high data protection overhead and write amplification. The proposed technologies are decoupled into three main research scopes: 1) A system-level solution to

achieve system lifetime extension, 2) a novel hardware and software co-design to minimize LDPC incurred read overhead and 3) an architectural solution at the firmware level to reduce write amplification in ONFD.

For Research Scope 1, we propose a novel technique to extend lifetime of MLC NAND flash based storage system. Previous works reveal that BER heavily depends on V_{th} levels. For example, [2, 4] identify that most retention time errors occur in V_{th} level 2 and V_{th} level 3. [23] reveals that programming to V_{th} level 1 and level 3 incurs most cell-to-cell interference bit errors. If the vulnerable V_{th} levels can be avoided, BER can be effectively reduced and therefore the device endurance can be extended. Inspired by this idea, we propose Data Pattern Aware (DPA) error protection technique which utilizes a data pattern unbalancing technique combining the existing BCH ECC to protect data integrity [24]. Our contribution of this scope is summarized as follows:

- We propose Pattern Probability Unbalance (**DPA-PPU**) to reduce the probability of data patterns that are sensitive to bit errors. DPA-PPU identifies the data correlation and adopts de-correlation and scrambling accordingly to unbalance the number of 1's and 0's in the stored data. By increasing the ratios of 1's, fewer cells are placed on the vulnerable V_{th} levels and therefore the BER of NAND flash memories is effectively reduced.
- We propose Data-Redundancy Management (**DPA-DRM**) to mitigate DPA-PPU induced performance degradation and provide protection to the redundant bits. DPA-PPU with different unbalancing efficiencies is adopted at different P/E cycle counts to avoid unnecessary redundancy write. Due to random data patterns, we also provide stronger protection to the redundant bits.

The simulation result shows that DPA technique can increase NAND flash storage system lifetime by 4× with marginal hardware and power overhead, offering a complementing solution to other NAND flash lifetime enhancement techniques like wear-leveling.

For Research Scope 2, we propose a novel design FlexLevel to achieve read speed-up in the LDPC code applied NAND flash storage system [25]. The FlexLevel is motivated by the fact that LDPC read overhead heavily depends on BER and that BER is partially

determined by the noise margin. Hence, LDPC read overhead reduction can be realized by noise margin increase. In FlexLevel, to diminish BER, we increase the noise margin of the NAND flash cell by V_{th} level reduction. Thereby, adoption of soft-decision LDPC can be avoided and read performance is improved. The contribution of this scope is summarized as follows:

- We propose **LevelAdjust** technique to reduce BER at the device level. It first decreases the number of V_{th} levels in the NAND flash cell to extend the noise margin of each V_{th} level. To further inhibit BER increase in the post cycling stage, we adopt NUNMA technique to increase retention time noise margin. By minimizing device BER, no extra memory sensing is needed for LDPC code and therefore read latency can be reduced.

- LevelAdjust technique enhances device reliability at the cost of memory capacity loss. To balance storage space reduction and performance improvement, we propose FlexLevel-AccessEval scheme (referred to **AccessEval** hereafter) at the system level. To maximize the efficiency of LevelAdjust technique, we only apply LevelAdjust techniques to the stored data with high soft-decision cost. Here, soft-decision cost denotes soft-decision LDPC induced performance degradation. By adopting AccessEval scheme, our FlexLevel design can achieve 11% read speedup with only 6.25% density loss.

For Research Scope 3, we propose a novel architectural design to reduce write amplification in ONFD. As mentioned in Section 1.1, two major causes to write amplification in ONFD are onode partial update and cascading update. To minimize onode partial update and cascading update induced data migration, the Data Migration Minimization (DMM) device design is proposed. Our contribution of this scope is summarized as follows:

- A multi-level garbage collection (MLGC) technique is proposed to reduce writes of onode partial update. MLGC adopts both page-level and byte-level garbage collection: When onode partial update occurs, instead of moving the un-updated bytes immediately, MLGC records the information of invalid bytes temporarily. By grouping invalid bytes incurred by multiple onode partial updates, the amount of moved data can be reduced.

- A virtual B+ tree is proposed to reduce cascading update within the per-object index. Each node page of the virtual B+ tree is assigned with a virtual address. The parent node page records the virtual addresses of the child node pages. A page table is used to map the virtual addresses to the physical addresses. Migration of the child node pages is only reflected in the page table without update to the parent node page. As such, cascading update induced data migration can be effectively reduced.
- The diff cache is proposed to further minimize cascading update. The diff cache leverages DRAM to reduce the writes to NAND flash memory. Due to the limited size, the diff cache selectively buffers data depending on the data type. To maximize the cache utilization, a diff cache replacement policy is proposed.

We evaluate the efficiency of the DMM device design under four workloads. The experimental results show that compared with the best art-of-state works, the DMM device can achieve up to 20% write reduction and extend the system lifetime by 76%. For future work directions, we will to explore the optimization space in the ONFD to improve wear-level efficiency.

The outline of this dissertation is summarized as follows: Chapter 1 presents the overall picture of this dissertation, including the research motivations, scopes and contributions; Chapter 2 introduces the proposed *data pattern aware* (DPA) data protection scheme and discusses its efficiency and overhead in detail; Chapter 3 analyzes NAND flash memory error patterns and describes the details of our proposed FlexLevel system design; Chapter 4 demonstrates the benefits of the proposed architectural solution – *data migration minimization* (DMM), in improving system performance and extending system lifetime of the ONFD. Chapter 5 finally summarizes the research work and presents the potential future research directions.

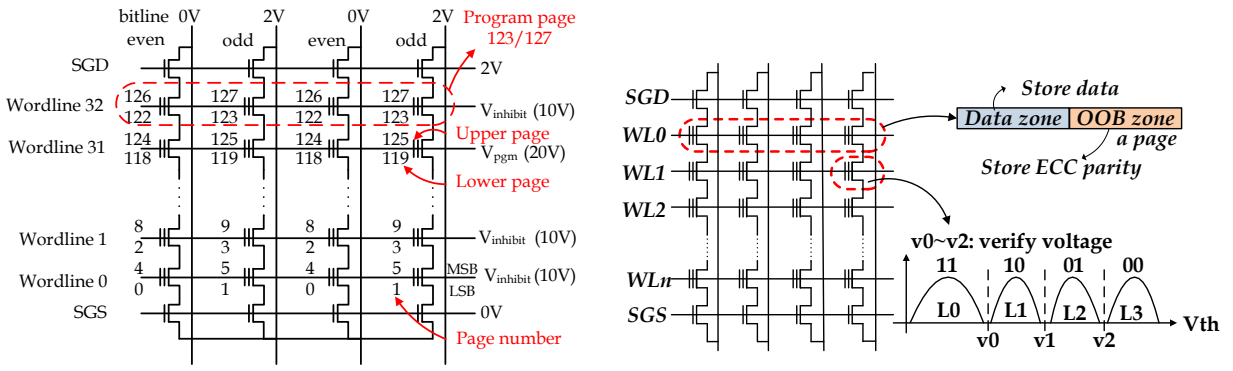
2.0 DPA: DATA PATTERN AWARE ERROR PREVENTION TECHNIQUE

In this chapter, we will present our Data Pattern Aware (DPA) error prevention technique. The structure of this chapter is organized as the follows: Section 2.1 presents the preliminary knowledge of NAND flash memories; Section 2.2 quantitatively characterizes bit error rate and error patterns of NAND flash memories; Section 2.3, 2.4 and 2.5 present DPA architecture and discuss DPA-PPU and DPA-DRM details, respectively; Section 2.6 presents the experimental results; Section 2.7 summarizes this chapter.

2.1 PRELIMINARY

2.1.1 MLC NAND Flash Basics

MLC NAND flash memory is composed of a number of blocks. A block is an array of NAND flash cells, or floating gate transistors, which can be sub-divided into a number of pages. Usually, a MLC NAND flash block has an even/odd bit-line structure, which is depicted in Fig. 2(a) [4]. Under the even/odd bit-line structure, a wordline stores two page groups, an even and an odd page group. Operations to each page group are realized by selecting the corresponding wordline and bitline. Each page group contains two pages: a lower page and an upper page. The most significant bit (MSB) and least significant bit (LSB) in the same cell belong to the lower page and the upper page in one page group, respectively. Each MLC NAND flash cell stores 2 bits by four V_{th} levels. L0 (the lowest level) $\sim$ L3 (the highest level) represent 11,10,01,00, respectively as shown in Fig. 2(b). Each page consists a data area and a Out-Of-Band (OOB) zone for ECC parity redundancy.



(a) MLC NAND flash even/odd bit-line structure.

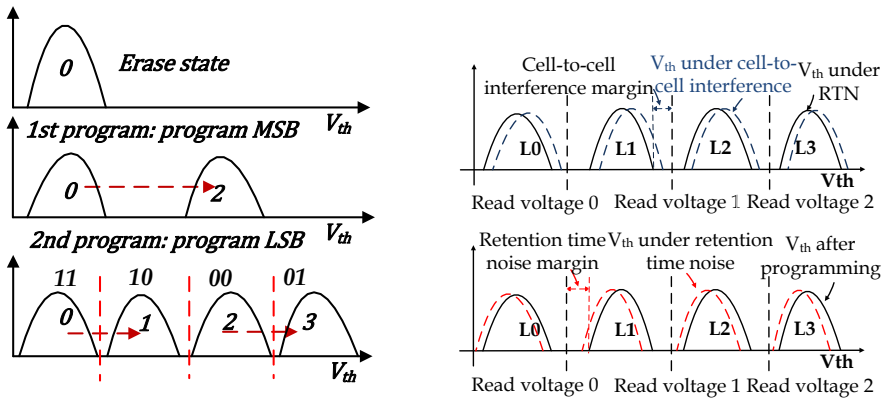
(b) MLC NAND flash device structure and the V_{th} distribution.

Figure 2: MLC NAND flash memory circuit structure.

MLC NAND flash supports three operations: program, read and erase. Program operation realizes injection of pre-defined amount of electrons to configure V_{th}. A two-step program operation can be performed to each cell as shown in Fig. 3(a) [2, 26]. The first program operation stores data in MSB and the second program operation stores LSB data. Program is performed by unit of page and all the pages within a block should be programmed sequentially. The logic bits are read out by comparing cell's V_{th} with a series of read reference voltages [4]. Erase operation removes electrons from floating gate to reduce NAND flash to a ready state (V_{th} level 0) for incoming program operations. Unfortunately, these operations cannot achieve ideal V_{th} distribution due to intrinsic noises. These noises lead to high BER and severely impair data integrity. There are three major contributors to V_{th} distortion which are identified by previous work: program disturb, retention time limit and read disturb [3, 4, 2].

2.1.2 Program Disturb

Program disturb stems from the combinative effect of random telegraph noise (RTN) and cell-to-cell interference. RTN develops from electrons capture and emission at charge trap sites. As shown in Fig. 3(b), RTN can either increase or reduce the V_{th} of the programmed



(a) 2-step program to MLC NAND flash memory.

(b) RTN-induced V_{th} distribution.

Figure 3: The program method and RTN noise of NAND flash memory.

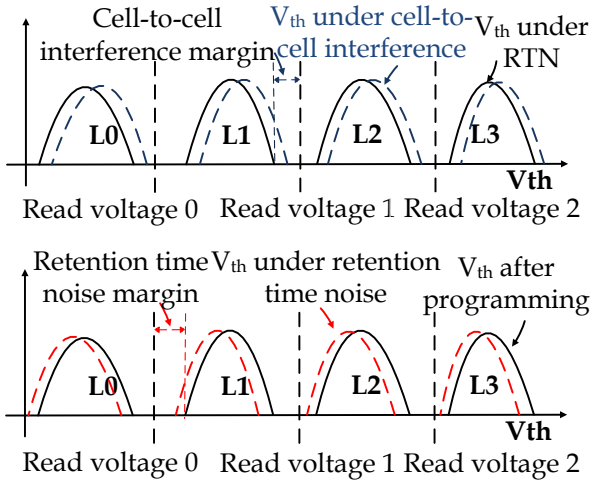
cell and results in wider threshold voltage distribution. [3] reveals that the effect of RTN is aggregated over P/E cycling. Assume that λ is the mean value of V_{th} shift. RTN induced V_{th} shift ΔV_{rtn} is modeled by [3]

$$p_{\Delta V_{rtn}} = \frac{1}{2\lambda} exp(-\frac{|\Delta V_{rtn}|}{\lambda}). \tag{2.1}$$

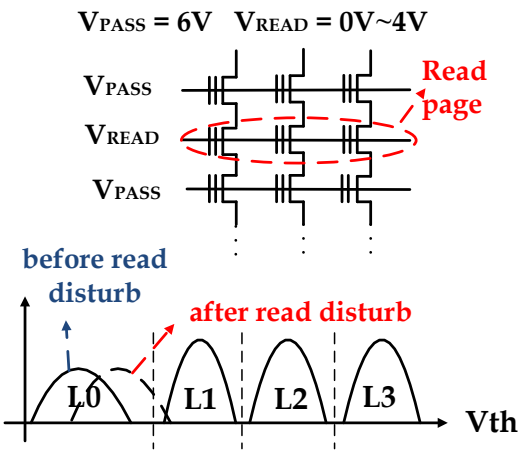
Programming one cell provokes the V_{th} shift of the neighboring cells via parasitic capacitance-coupling. This is cell-to-cell interference. The V_{th} shift of the victim cell ΔV_{c2c} can be modeled by [3]

$$\Delta V_{c2c} = \sum_{k=0} \Delta V_p^{(k)} \times \gamma^{(k)}. \tag{2.2}$$

$\Delta V_p^{(k)}$ and $\gamma^{(k)}$ denote the V_{th} shift of the interfering cell after programming and coupling ratio, respectively. In the even/odd bit structure, there exists coupling ratios on three directions: γ_y, γ_x and γ_{xy} [27]. Cell-to-cell interference noise margin is shown in Fig. 4(a). Each V_{th} level is confined by lower and/or upper read reference voltages (V_{rd-ref}). Since cell-to-cell interference noise incurs V_{th} increase, cell-to-cell interference noise margin is defined as the voltage difference between V_{th} after RTN and the upper read reference voltage. [23] shows that the effect of cell-to-cell interference is V_{th} level dependent. The largest V_{th} transition of the victim cell occurs when the interfering cell is programmed to level L1 and L3.



(a) Illustration of cell-to-cell interference.



(b) Description of read disturb. Both the read and unread cells are exposed to risk of read disturb.

Figure 4: The device noise in NAND flash memory.

2.1.3 Read Disturb

Read disturb is originated from both Fowler-Nordheim tunneling mechanism and stress induced leakage current (SILC). The read operation is shown in Fig. 4(b). The verify voltage V_{READ} (0∼4V) is applied to the wordline of the read page while V_{PASS} (6V) is applied to the unread pages in the same block. Both V_{PASS} and V_{READ} cause electrons to transmit to the floating gate, directly leading to V_{th} increase. The BER resulting from read disturb increases with P/E cycling and read count. The research of [28] shows the lowest level L0 is most susceptible to read disturb.

2.1.4 Retention Time

Retention time limitation results from electron detrapping and SILC. Electrons are trapped in transistor tunnel oxide through P/E cycling [3]. These trapped electrons gradually leak away and assist charges stored on floating gates to escape, leading to V_{th} decrease (Fig. 4(b)).

It is shown that retention time error dominates the post-cycling error [2]. According to [3], V_{th} shift distribution due to retention time limitation is modeled by $N(\mu_d, \sigma_d^2)$. μ_d and σ_d^2 can be expressed by

$$\begin{cases} \mu_d = K_s(x - x_0)K_d N^{0.3} ln(1 + t/t_0), & (2.3) \\ \sigma_d^2 = K_s(x - x_0)K_m N^{0.4} ln(1 + t/t_0). & (2.4) \end{cases}$$

Here, K_s, K_d, K_m and t_0 are constants. N is P/E cycle count. x_0 is V_{th} of level 0. x is the initial V_{th} after programming and t is storage time. As shown in Fig. 4(a), retention time noise margin is defined as the voltage difference between the lower read reference voltage and the V_{th} after programming (under the effect of both RTN and cell-to-cell interference). Eq. 2.3 and 2.4 shows that higher initial V_{th} incurs larger threshold voltage shift. V_{th} level L3 is most vulnerable to retention time error.

2.2 MOTIVATIONS

2.2.1 Lifetime Model

BCH ECC code is prevalently employed in NAND flash based storage system to prevent data from bit error. A NAND flash block lifetime is defined as P/E cycle count when ECC failure rate reaches a threshold T_{uber}. ECC failure rate for a NAND flash block can be expressed by

$$F = max(f_{prog}, f_{rd}, f_{rt}). \quad (2.5)$$

f_{prog},f_{rd} and f_{rt} are error rate induced by program disturb, read disturb and retention time limitation, respectively. Assume n-bit BCH ECC (m,l,n) is performed to l-bit data block. m denotes the total codeword length. The yield $Y_{ber}(n)$ can be expressed by

$$uber(k) = \frac{1 - \sum_{i=0}^{k} C_m^i p_c^i (1 - p_c)^{(m-i)}}{n}. \quad (2.6)$$

Here, p_c is BER of a single NAND flash cell. Assume the probability that each flash cell be programmed to threshold voltage (V_{th}) level Li is p_i (i = 0, 1, 2, 3). The error rate of each level is represented by p_{li}. p_c can be calculated by

$$p_c = \sum_{i=0}^{3} p_{li} \times p_i. \tag{2.7}$$

p_i is determined by the ratio of 1's and 0's in the data stored in flash cells. $p(0)$ and $p(1)$ denote the probability of 1's and 0's. $p_0 \sim p_3$ can be represented by $p(1)p(1)$, $p(0)p(1)$, $p(1)p(0)$ and $p(0)p(0)$.

Based on Eq. 3.1,2.7, to estimate uber(k), we need to first calculate value of p_{li} as well as the probability of 1's and 0's $p(0)$ and $p(1)$. We employ reliability models in Section 2.1 to evaluate the error rate of each V_{th} level p_{li}. The flash model proposed in Eq. 2.1,2.2,2.3,2.4 is used to estimate the program disturb and retention time error rate. Based on the research of [29, 28, 30], we set V_{th} fluctuation led by read disturb as random variables with Gaussian distribution. The mean μ_{rd} and variation σ_{rd} are expressed by

$$\begin{cases} \mu_{rd} = \dfrac{1}{\gamma} In[1 + \gamma\beta t_s e^{\gamma(V_{CG} - V_{T,0})} N^{0.3} K_r] & (2.8) \\ \sigma_{rd} = \sqrt{\alpha(1 - e^{\gamma\mu_{rd}})} & (2.9) \end{cases}$$

γ, β, K_r and α are coefficients. V_{CG} and $V_{T,0}$ represent the voltage applied to the wordline and the voltage in floating gate. t_s and N denote read pulse duration and P/E cycle count, respectively.

To evaluate probability of 1's and 0's, we investigate 1's and 0's distribution in seven types of data listed in Table 1. Fig. 5(a) shows that 1's ratio in system metadata files is around 45% while others are approximately 50%. Hence, it is safe to assume that $p(0) = p(1) = 0.5$. For the evaluation purpose, we perform 8-bit BCH ECC to a 512B data block and T_{uber} is set 10^{-13} [2]. The L0 V_{th} is modeled by Gaussian distribution $N(1.1, 0.35)$. The ISPP verify voltages and the incremental program step voltage are 2.55, 3.15, 3.75 and 0.3, respectively [27]. λ is set $2.8 \times 10^{-4} N^{0.4}$ based on [31]. We adopt the emerging all-bit-line structure. The coupling ratios γ_y and γ_{xy} are set 0.08 and 0.0048 [27]. By fitting the data in [5], K_d and K_m are 4×10^{-5} and 3×10^{-6}. For the read disturb model, γ, β, and t_s are set 1.1, 8.8×10^5, 8×10^{-2} respectively. By fitting the data in [29, 28], α and K_r are set

Table 1. File data characteristics

file type	*number*	*file size*
mp3 file	2	7.5MB, 6.3MB
mp4 file	1	101MB
compressed file(tar.gz)	2	5.3MB,4.2MB
pictures (.jpg)	2	1.42MB, 1.37MB
pdf file	2	8.6MB,4.2MB
office files(.ppt)	2	797KB, 1.2MB
system matadata file	6	total 0.5MB

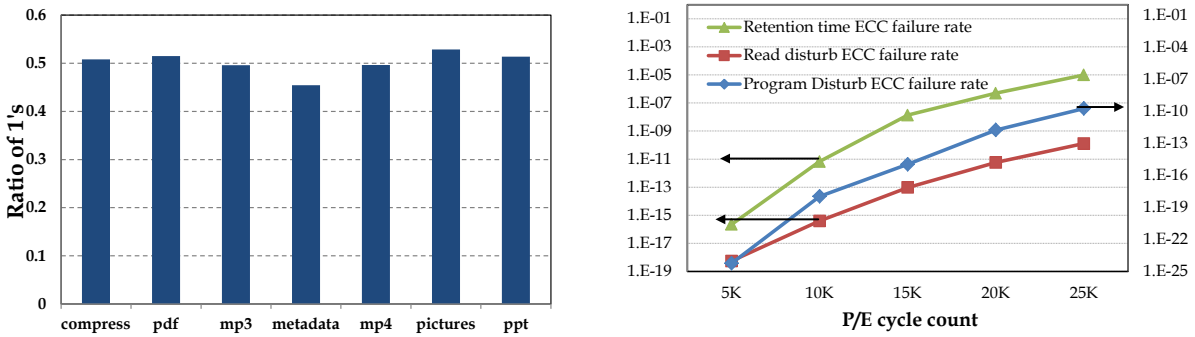
5.3×10^{-2} and 11.9. Read disturb is evaluated under 10K read count and retention time error rate is estimated with 1 year elapsed time. The ECC failure rate under different P/E cycle count is shown in Fig. 5(b). Data integrity cannot be guaranteed after 7.5K P/E cycling.

2.3 DPA OVERVIEW

Our DPA error prevention scheme aims to minimize the error rate of a single cell. Our scheme is based on the following pattern-dependent bit error features:

- Program disturb bit error is prone to occur when the interfering cell is programmed to V_{th} L1 and L3.
- It is highest probability that retention time error occur at V_{th} level L3.

To minimize the effect of program disturb and retention time noise, DPA scheme increases the probability of V_{th} level L0 by maximizing the ratio of 1's in the stored data. The DPA



(a) The distribution of 1's number in 8-bytes data block.

(b) The 8-bit ECC failure rate under different P/E cycling.

Figure 5: The ECC failure rate of NAND flash based storage system

error prevention system is illustrated in Fig. 6. DPA incorporates DPA-PPU and DPA-DRM. DPA-PPU implements data pattern conversion and DPA-DRM performs data management to mitigate DPA-PPU induced performance overhead. In our system, we employs a data buffer with simple LRU replacement cache policy. A array of flash chips are deployed to increase the access throughout. The write data from the host is first stored in the data buffer and data pattern conversion is performed to the evicted data before it is flushed to flash memory. For the read requests, the data is read out from flash and is recovered by DPA-PPU and sent back to the host.

2.4 DPA-PPU: PATTERN PROBABILITY UNBALANCE

To place more cell in V_{th} level L0, DPA-PPU performs data pattern conversion to increase the number of 1's. Since it is easier to decrease the number of 1's, we choose to reduce the number of 1's first and then inverts the whole data page. Our DPA-PPU investigates data correlation of the stored data. For the strong correlated data, de-correlator proposed in [32] is adopted to decrease the number of 1's. It performs simple XOR operation to neighboring data and reduction of 1's number can be realized by incurring negligible redundant overhead.

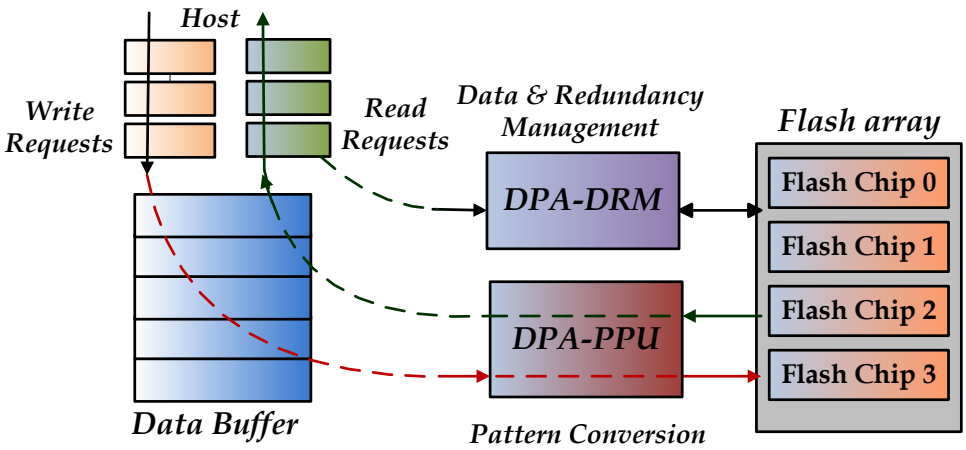


Figure 6: DPA Architecture Overview

Recently, many signal processing techniques are employed in the flash memory storage system [33, 34, 27, 35, 36]. In this work, we utilize a scheme based on scrambling coding to unbalance the number of 0's and 1's for the weakly correlated data. Basically, scrambling implements modulo 2 operation in Galois Field via cyclic XORing the data stream with a polynomial. If we can choose a appropriate polynomial whose pattern overlaps with the data to the maximum degree, 1-to-0 ratio can be effectively un-equalized. DPA-PPU performs data pattern conversion to increase the ratio of 1's in the stored data by performing de-correlation or scrambling, followed by bit inversion operation. If the data demonstrates a strong correlation, the de-correlation scheme in [32] can be adopted to decrease the number of 1's in the data by applying XOR operations between neighboring bytes. If the data is weakly correlated, we propose a scheme based on scrambling coding to skew the ratio of 1's and 0's. After the de-correlation or scrambling, all data bits are flipped to obtain the codeword of which the majority are 1's.

The architecture of DPA-PPU is presented in Fig. 7. A data page is divided into multiple data chunks. The data chunks are processed by de-correlation and scrambling circuits in parallel. Scrambling circuit performs modulo 2 division, i.e., cyclic XOR, on the data by using a polynomial as divider. If the pattern of the polynomial overlaps with the data much, the number of 1's can be effectively reduced. DPA-PPU implements n different scrambling

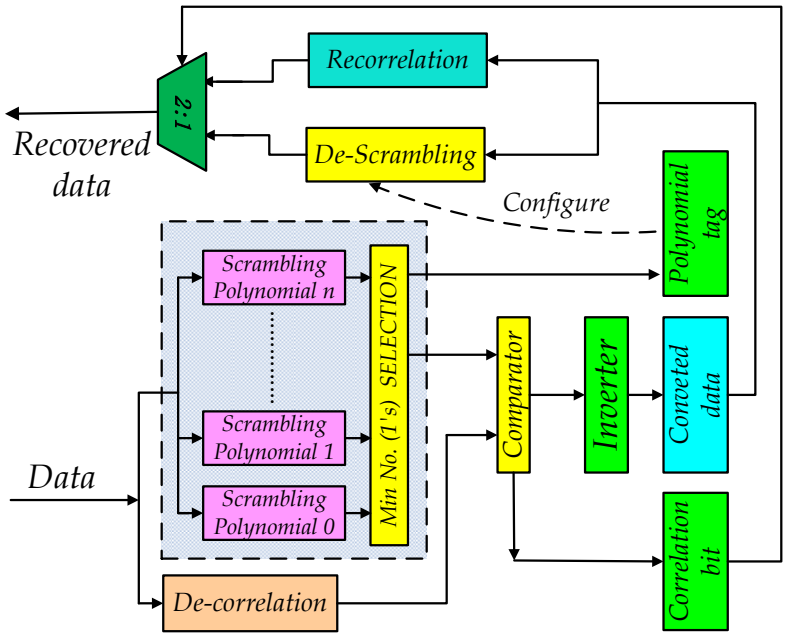


Figure 7: Architecture of DPA-PPU.

circuits which are differentiated by the polynomial tags from 0 to $n-1$. The data is scrambled by different polynomials and the result that has the fewest 1's will be selected as the output of scrambling circuit. Then the number of 1's in the output data from de-correlation and scrambling circuits are checked by a comparator. The one with the fewer 1's is selected and inverted before being flushed into the flash chips. If the de-correlated data is selected then the correlation bit is set to 1. Otherwise, the correlation bit is set to 0. The corresponding polynomial tag (if applicable) and the correlation bit must be also stored in the flash chip as they are required when re-correlation or descrambling is performed for data recovery at read operations.

The scrambling circuit efficiency is determined by the polynomial pattern, the data chunk size, the number of polynomials and the polynomial order. For example, the number of 1's can be effectively reduced by maximizing the overlapping bits between the data and the polynomial. In normal applications, the occurrence probabilities of 1's and 0's in the stored data are approximately equal [32]. Hence, the polynomials with a 1-to-0 ratio of 1:1 are generally

preferred. The increases of data chunk size diversifies the data pattern and thus potentially decreases the overlapping area between the data chunk and the polynomial. Hence, the scrambling efficiency degrades when the data chunk size rises. Similarly, increasing the number of polynomials may enhance the scrambling efficiency by maximizing pattern overlapping probability. Polynomial order itself does not directly affect the scrambling efficiency. However, the polynomial with a higher order can offer more available polynomials which may improve the pattern overlapping probability.

The scrambling-descrambling circuits can be implemented with simple linear feedback shift registers (LFSRs) with very marginal hardware cost. At 65nm technology node, the power of the scrambling-descrambling circuit is around 9mW, which is negligible compared to the large power consumption of the read and write operations of NAND flash based storage system (which is at $\geq$ 5W for a 256GB system). Since only one correlation bit is required by one data page, the incurred hardware cost is negligible. Similarly, the hardware overhead of polynomial tags is also marginal: assume the data chunk size is 8B and total 64 16-order polynomials are included in the scrambling-descrambling circuits, the extra space required to store the polynomial tags in a 256GB NAND flash based storage system is only 16GB (6%) because one third of the tags can be stored in OOB zone.

2.5 DPA-DRM: DATA-REDUNDANCY MANAGEMENT

DPA-PPU converts the data page into the pattern with more 1's to reduce the error rate resulting from retention time limitation and program disturb. To reduce DPA-PPU induced redundancy overhead, we employ a chunk-size-adaptive DPA-PPU scheme. The error rate resulting from retention time increases with P/E cycle count. Therefore, at the early post cycling stage, we adopt a large chunk to achieve a moderate un-equalization rate due to relatively low error rate. As the error rate increases with P/E cycling, a small chunk size is adopted to maximize the probability the flash cell be placed on level L0. However, smaller chunk size incurs more redundancy. There are two problems with the redundant bits: (1)

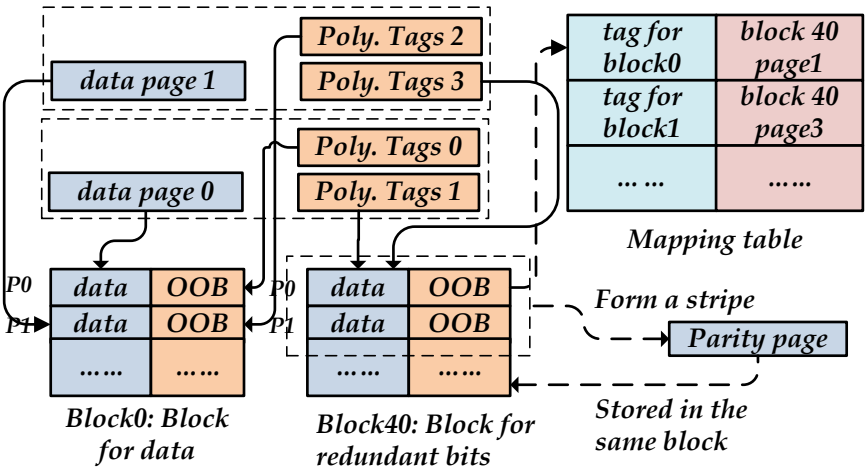


Figure 8: Redundant pages & data pages

They are characteristics of more random patterns than the converted data; (2)They introduce extra write operations and therefore accelerate the wear-out of flash memory and incur performance degradation.

Due to the random patterns, the redundant page is vulnerable to retention time error than the converted data. Therefore, we differentially deal with the converted data and the redundant bits. Each data page consists of multiple data chunks with multiple polynomial tags. We store a part of the polynomial tags in the OOB area of the data page and the rest is stored in a **redundant page**. Redundant pages and data pages are stored in different blocks. A mapping table is utilized to track the redundant pages. Since the redundant page is vulnerable to retention time error, we perform RAID-5 to the redundant pages. Instead of grouping the stripe by logic page number of conventional RAID-5 [37], we form the stripe by the physical page number and the parity page is stored in the same block to eliminate the necessity of extra mapping table. An example is shown in Fig. 8. Assume data page 0 have two polynomial tags: tag 0 and tag 1. OOB space can hold only one tag and ECC parity. The data page 0 and data page 1 store the tag 0 and tag 2 in the OOB space of the data page with the tag 1 and tag 3 in a redundant page. Redundant pages are stored in flash block 40 and 2 redundant pages form a stripe. A parity page is calculated and stored back to block 40.

To minimize redundancy-induced performance degradation, we adopt a delay write similar to [37]. When a redundant page is ready, it stays in the data buffer and is flushed to NAND flash only due to system idle time. For the read operation, the redundant page are read out with the data page to configure the de-scrambling circuit for data recovery. Therefore, to reduce read response time, we store the redundant pages and the data page to two different chip so that data page and the redundant page can be accessed in parallel. The performance overhead of DPA-DRM is evaluated in Section 2.6. Our DPA-DRM introduces a mapping table. A block of data pages only consume one or two redundant pages and therefore, mapping table is small. For 256GB storage system, it only consumes several mega bytes. The mapping table resides in data buffer for fast access. DPA-PPU places more cells on V_{th} level L0 and exposes them to higher risk of read disturb. However, increase of the cell V_{th} level L0 also mitigates the cell-to-cell interference and provide larger noise margin to read disturb. The effect of DPA scheme on read disturb will be evaluated in Section 2.6.

2.6 EXPERIMENTAL RESULTS

Flashsim [38] is adopted as our simulation platform. We modify the simulator by adding multi-chip access capability and incorporate our DPA error prevention scheme in it. The benchmarks representing five applications are selected to evaluating our scheme. The workload characteristics are listed in Table 2. The specification of MLC NAND flash memory used for our experiment is summarized in Table 3. The storage system capacity is set 256GB. A 8-bit BCH ECC (4200,4096,104) is applied to every 512-byte data block. We first study the efficiency of the de-correlation and scrambling circuits on reducing the ratio of 1's in the data (or reducing the ratio of 0's in the data after the bit inversion in Fig. 7) for the seven data types listed in TABLE 1, Fig. 9 shows the ratio of 1's in the data before and after processed by de-correlation circuit. Among all data types, only *system metadata* exhibits good correlation where the ratio of 1's decreases from 0.45 to 0.27. Other data types, however, display a more random pattern and the reduction of the ratio of 1's is very small after the de-correlation.

Table 2. Workload characteristics.

Disk trace	Write ratio	Seq.wr.	Application
WIN 7	42%	15.2%	p2p, office and web serfing
RHEL	93%	2.3%	Server access
TPC-C	99%	0.9%	OLTP application
financial [39]	98%	1.9%	OLTP application
web search [39]	0.02%	0	Access to search engines

Table 3. The parameters of MLC NAND flash

Capacity	Block Size	Block Number	Page Size
	512KB	4096	4KB+218Bytes
Timing	Program Latency	Read Latency	Erase Latency
	900 μs	50μs	3.5ms

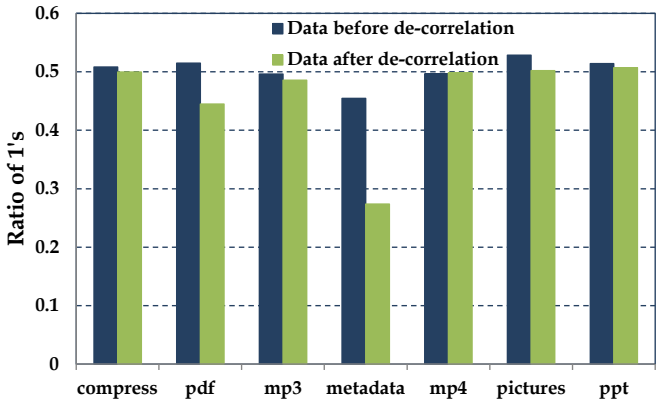


Figure 9: The ratio of 1's before and after de-correlation.

In the evaluations of different scrambling schemes, the default values of data chunk size, polynomial number and polynomial order are set to 8B, 64 and 16, respectively. The 64 polynomials we use are listed in Table 4. We only change the value of one parameter each time to evaluate its impact on scrambling efficiency. We show the probability of each scrambling polynomials to reduce the number of 1's in Table 5. The scrambling is inefficient to reduce 1's number in the system metadata file: The total probability of the sixty-four polynomials to reduce 1's is only 10%. This is because the system metadata file has fewer 1's than 0's. Scrambling (or OXRing) these data with the polynomials with equal 1's and 0's generates more 1's. For other type of data, the scrambling demonstrates good efficiency: Each polynomial has approximately 50% probability to reduce 1's numbers.

Table 4. Sixty-four polynomials employed for scrambling

poly. no	**polynomial**	**poly. no**	**polynomial**
0	$x^{16}+x^{15}+x^{14}+x^7+x^5+x^4+1$	1	$x^{16}+x^{12}+x^{11}+x^9+x^2+x^1+1$
2	$x^{16}+x^{10}+x^8+x^7+x^6+x^5+x^3+x^1+1$	3	$x^{16}+x^{11}+x^7+x^6+x^4+x^3+x^2+x^1+1$
4	$x^{16}+x^{12}+x^6+x^5+x^4+x^3+x^2+x^1+1$	5	$x^{16}+x^{12}+x^{10}+x^9+x^7+x^4+x^3+x^1+1$
6	$x^{16}+x^{12}+x^{10}+x^9+x^8+x^6+x^2+x^1+1$	7	$x^{16}+x^{13}+x^{12}+x^9+x^8+x^7+x^6+x^4+1$
8	$x^{16}+x^{12}+x^{11}+x^8+x^6+x^5+x^3+x^1+1$	9	$x^{16}+x^{15}+x^{14}+x^9+x^8+x^6+x^5+x^4+1$
10	$x^{16}+x^{13}+x^{12}+x^11+x^10+x^9+x^8+x^7+1$	11	$x^{16}+x^{15}+x^{14}+x^{11}+x^{10}+x^9+x^8+x^5+1$
12	$x^{16}+x^{15}+x^{12}+x^11+x^10+x^9+x^6+x^5+1$	13	$x^{16}+x^{15}+x^{13}+x^{12}+x^8+x^7+x^6+x^5+1$
14	$x^{16}+x^{12}+x^8+x^7+x^6+x^4+x^3+x^1+1$	15	$x^{16}+x^{15}+x^{14}+x^{12}+x^{11}+x^{10}+x^7+x^4+1$
16	$x^{16}+x^{12}+x^{11}+x^{10}+x^7+x^2+1$	17	$x^{16}+x^{13}+x^{11}+x^6+x^5+x^4+1$
18	$x^{16}+x^{13}+x^8+x^7+x^5+x^4+1$	19	$x^{16}+x^{13}+x^9+x^7+x^5+x^4+1$
20	$x^{16}+x^8+x^7+x^5+x^4+x^3+x^2+x^1+1$	21	$x^{16}+x^9+x^8+x^7+x^6+x^5+x^4+x^3+1$
22	$x^{16}+x^{14}+x^{11}+x^8+x^5+x^4+1$	23	$x^{16}+x^{15}+x^{11}+x^8+x^5+x^4+1$
24	$x^{16}+x^{12}+x^{11}+x^8+x^5+x^1+1$	25	$x^{16}+x^{12}+x^{11}+x^7+x^4+x^1+1$
26	$x^{16}+x^{12}+x^{11}+x^7+x^3+x^2+1$	27	$x^{16}+x^{15}+x^{10}+x^7+x^6+x^4+1$
28	$x^{16}+x^{12}+x^{10}+x^9+x^6+x^1+1$	29	$x^{16}+x^{12}+x^{11}+x^7+x^4+x^1+1$

Table 4 (continued)

poly. no	polynomial	poly. no	polynomial
30	$x^{16}+x^{15}+x^{10}+x^7+x^6+x^4+1$	31	$x^{16}+x^{15}+x^{14}+x^8+x^6+x^4+1$
32	$x^{16}+x^5+x^3+x^2+1$	33	$x^{16}+x^{14}+x^{13}+x^{11}+1$
34	$x^{16}+x^5+x^4+x^3+1$	35	$x^{16}+x^{13}+x^{12}+x^{11}+1$
36	$x^{16}+x^6+x^4+x^1+1$	37	$x^{16}+x^{15}+x^{12}+x^{10}+1$
38	$x^{16}+x^8+x^7+x^5+1$	39	$x^{16}+x^{11}+x^9+x^8+1$
40	$x^{16}+x^9+x^4+x^2+1$	41	$x^{16}+x^{14}+x^{12}+x^7+1$
42	$x^{16}+x^9+x^4+x^3+1$	43	$x^{16}+x^{13}+x^{12}+x^7+1$
44	$x^{16}+x^9+x^5+x^2+1$	45	$x^{16}+x^{14}+x^{11}+x^7+1$
46	$x^{16}+x^9+x^7+x^2+1$	47	$x^{16}+x^{14}+x^9+x^7+1$
48	$x^{16}+x^{12}+x^{11}+x^{10}+x^8+$ $x^7+x^6+x^5+x^4+x^3+x^2+x^1+1$	49	$x^{16}+x^{15}+x^{14}+x^{13}+x^{12}+x^{11}+$ $x^{10}+x^9+x^8+x^6+x^5+x^4+1$
50	$x^{16}+x^{12}+x^{11}+x^{10}+x^9+$ $x^8+x^7+x^5+x^4+x^3+x^2+x^1+1$	51	$x^{16}+x^{15}+x^{14}+x^{13}+x^{12}+$ $x^{11}+x^9+x^8+x^7+x^6+x^5+x^4+1$
52	$x^{16}+x^{13}+x^{11}+x^{10}+x^9+$ $x^7+x^6+x^5+x^4+x^3+x^2+x^1+1$	53	$x^{16}+x^{15}+x^{14}+x^{13}+x^{12}+x^{11}+$ $x^{10}+x^9+x^7+x^6+x^5+x^3+1$
54	$x^{16}+x^{13}+x^{12}+x^{11}+x^8+$ $x^7+x^6+x^5+x^4+x^3+x^2+x^1+1$	55	$x^{16}+x^{15}+x^{14}+x^{13}+x^{12}+x^{11}+$ $x^{10}+x^9+x^8+x^5+x^4+x^3+1$
56	$x^{16}+x^{13}+x^{12}+x^{11}+x^9+$ $x^8+x^7+x^6+x^5+x^3+x^2+x^1+1$	57	$x^{16}+x^{15}+x^{14}+x^{13}+x^{11}+$ $x^{10}+x^9+x^8+x^7+x^5+x^4+x^3+1$
58	$x^{16}+x^{13}+x^{12}+x^{11}+x^{10}+$ $x^9+x^7+x^6+x^4+x^3+x^2+x^1+1$	59	$x^{16}+x^{15}+x^{14}+x^{13}+x^{12}+x^{10}+$ $x^9+x^7+x^6+x^5+x^4+x^3+1$
60	$x^{16}+x^{13}+x^{12}+x^{11}+x^{10}+$ $x^9+x^7+x^6+x^5+x^4+x^3+x^1+1$	61	$x^{16}+x^{15}+x^{13}+x^{12}+x^{11}+x^{10}+$ $x^9+x^7+x^6+x^5+x^4+x^3+1$
62	$x^{16}+x^{14}+x^{11}+x^9+x^8+$ $x^7+x^6+x^5+x^4+x^3+x^2+x^1+1$	63	$x^{16}+x^{15}+x^{14}+x^{13}+x^{12}+x^{11}+$ $x^{10}+x^9+x^8+x^7+x^5+x^2+1$

Table 5. The probability of each polynomial to reduce 1's number

poly. no	**metadata**	**pdf**	**compress**	**ppt**	**mp3**	**mp4**	**picture**
0	0.000015	0.438711	0.471996	0.417557	0.440888	0.440549	0.534002
1	0	0.441995	0.472435	0.421484	0.448484	0.442699	0.541311
2	0	0.441939	0.475722	0.421859	0.447419	0.444703	0.542909
3	0.000021	0.440033	0.472821	0.419434	0.441828	0.441283	0.535894
4	0.000498	0.438511	0.471682	0.416916	0.439944	0.439712	0.530911
5	0.003138	0.443568	0.476516	0.423597	0.446503	0.445241	0.548343
6	0.00036	0.440135	0.474084	0.421106	0.445015	0.442773	0.546743
7	0.009584	0.439273	0.47067	0.419279	0.450382	0.441269	0.535987
8	0.000904	0.440829	0.475366	0.422825	0.44839	0.444916	0.54392
9	0.000072	0.440239	0.471414	0.420736	0.448248	0.441094	0.537634
10	0	0.433379	0.465882	0.415668	0.444976	0.438262	0.52978
11	0.000082	0.438736	0.469896	0.418377	0.448772	0.440774	0.53904
12	0.00396	0.438495	0.469067	0.420166	0.444459	0.440136	0.537302
13	0	0.439279	0.469607	0.418782	0.44554	0.440156	0.536442
14	0.020626	0.444134	0.476009	0.422674	0.443483	0.44471	0.544833
15	0	0.438562	0.471534	0.420231	0.452096	0.441536	0.534617
16	0.004699	0.440247	0.473177	0.420474	0.448235	0.442807	0.545585
17	0.02232	0.432267	0.460016	0.413064	0.440436	0.433806	0.51866
18	0.001125	0.442318	0.473832	0.422917	0.448515	0.443491	0.542439
19	0	0.435023	0.46326	0.41667	0.444034	0.436332	0.521316
20	0.000144	0.442425	0.476081	0.420478	0.444676	0.443124	0.539964
21	0.000005	0.436087	0.463277	0.413615	0.443698	0.436131	0.523366
22	0.00001	0.44069	0.473147	0.420055	0.442139	0.442136	0.542116
23	0	0.437115	0.464452	0.416242	0.446398	0.437421	0.525261
24	0	0.440651	0.473223	0.421484	0.448845	0.442935	0.54197

Table 5 (continued)

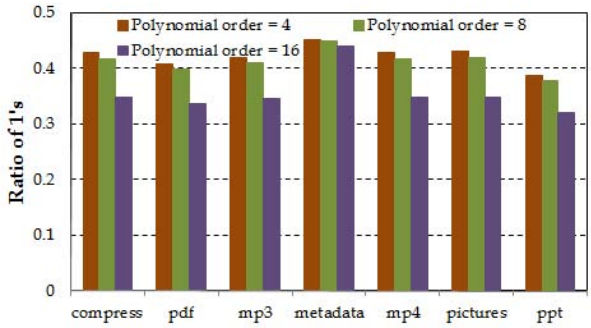
poly. no	metadata	pdf	compress	ppt	mp3	mp4	picture
25	0	0.436435	0.465094	0.415462	0.44855	0.437481	0.525322
26	0.003343	0.440164	0.47173	0.419896	0.441298	0.441608	0.539524
27	0.000185	0.43668	0.465181	0.414801	0.447648	0.437535	0.526578
28	0.000005	0.442247	0.474702	0.42222	0.448543	0.443607	0.54371
29	0	0.43625	0.464912	0.416166	0.450187	0.43754	0.527631
30	0.000257	0.443861	0.477076	0.421856	0.444209	0.443435	0.541014
31	0	0.436375	0.463546	0.415226	0.443077	0.437117	0.526386
32	0.00169	0.44151	0.473586	0.41996	0.447599	0.442491	0.542267
33	0.020215	0.433343	0.460023	0.412842	0.442338	0.433617	0.517418
34	0.009106	0.440225	0.470997	0.419154	0.448194	0.441919	0.539684
35	0.014971	0.433186	0.459926	0.412919	0.441133	0.433608	0.519018
36	0.000955	0.444899	0.475905	0.42331	0.448923	0.445061	0.546133
37	0	0.433029	0.461257	0.415673	0.440956	0.435036	0.521418
38	0.000272	0.439374	0.47022	0.418455	0.447588	0.440331	0.537422
39	0	0.435306	0.46454	0.4144	0.44497	0.43723	0.528459
40	0.000575	0.439338	0.470926	0.418815	0.446566	0.441134	0.541256
41	0	0.436811	0.46576	0.415142	0.443259	0.438342	0.529789
42	0.000021	0.439517	0.471394	0.421001	0.447655	0.442378	0.54276
43	0	0.438367	0.466656	0.414496	0.447086	0.438388	0.528789
44	0.000026	0.443264	0.47431	0.421401	0.44757	0.443815	0.543008
45	0	0.435302	0.465885	0.415209	0.441262	0.438547	0.528529
46	0	0.443567	0.476831	0.424812	0.449777	0.445766	0.544746
47	0	0.434661	0.465893	0.417268	0.440229	0.438003	0.530022
48	0.007658	0.43774	0.470313	0.415336	0.436354	0.43882	0.52801
49	0.00037	0.438954	0.471142	0.419517	0.448708	0.441169	0.53715
50	0.004001	0.439752	0.472526	0.419452	0.441584	0.441125	0.535459
51	0	0.439336	0.470785	0.418509	0.44948	0.441376	0.537395
52	0.000026	0.436671	0.471413	0.417003	0.438932	0.439507	0.529943

Table 5 (continued)

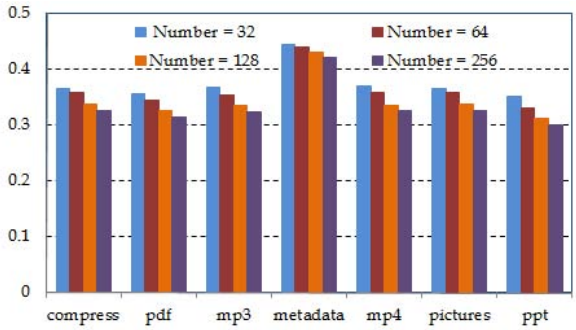
poly. no	metadata	pdf	compress	ppt	mp3	mp4	picture
53	0.009676	0.440834	0.471516	0.419007	0.448926	0.442098	0.540658
54	0.000021	0.435268	0.47043	0.414848	0.435516	0.438059	0.527065
55	0.000437	0.439448	0.471522	0.419285	0.446475	0.442177	0.539935
56	0.004746	0.441804	0.472875	0.419781	0.443671	0.44236	0.542425
57	0.009835	0.44219	0.471141	0.421075	0.448039	0.442399	0.539407
58	0	0.442474	0.472383	0.419839	0.441966	0.441578	0.536879
59	0	0.442562	0.471749	0.421147	0.448386	0.442328	0.540393
60	0.001705	0.443609	0.476583	0.421987	0.448994	0.445104	0.547865
61	0.002624	0.440573	0.471773	0.420737	0.448184	0.442375	0.540288
62	0.000914	0.435672	0.46928	0.414176	0.433102	0.437578	0.527946
63	0.003765	0.443383	0.4741	0.421855	0.447927	0.444083	0.543792

As shown in Fig. 10(a), the average ratio of 1's decreases from 0.42 to 0.34 in the scrambling scheme as the polynomial order rises from 4 to 16. Such improvement is due to increase of the available polynomials. Fig. 10(b) shows that the average ratio of 1's decreases by 11.5% as the polynomial number changes from 32 to 256. Fig. 10(c) shows that the ratio of 1's increases from 0.27 to 0.42 on average by increasing the chunk size from 4B to 32B due to the reduced overlap pattern between the data chunk and the polynomial.

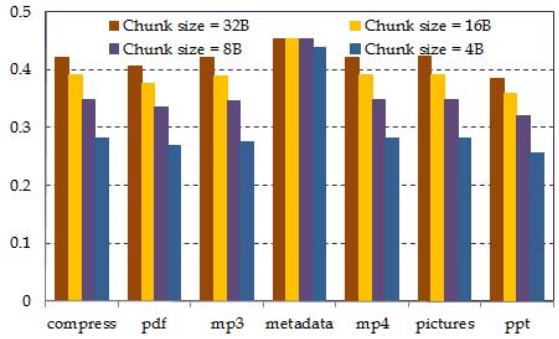
Finally, we compare the ratios of 0's in the outputs of the DPA-PPU's with a data chunk size of 4B and 8B, respectively. Note that the data from the DPA-PPU is the inversion of one output from the de-correlation or scrambling circuits. All other parameters are set to the default values. As shown in Fig. 10(d), the efficiency of DPA-PPU is generally better when the data chunk size is small. The only exception is *system metadata* under 8B data chunk size where the data is mainly processed by de-correlation circuit.



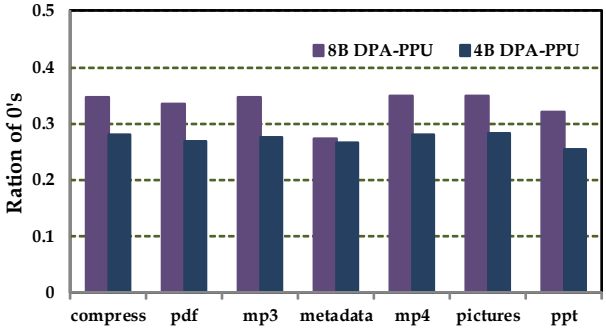
(a) The efficiency of scrambling under 4,8 and 16-order polynomials.



(b) The efficiency of scrambling when the polynomial number changes from 32 to 256.



(c) The efficiency of scrambling when the data chunk size changes from 4B to 32B.



(d) Ratio of 0's under DPA-PPU.

Figure 10: The efficiency of DPA-PPU to reduce 0's ratio

2.6.1 DPA Error Failure Rate

Fig. 11 shows the simulated V_{th} programming probability of NAND flash cells under WIN 7 workload before and after applying DPA-PPU. The results of different data chunk sizes, i.e., 4B and 8B, are also simulated. The probability of L0 increases from 24.5% to 48% when the DPA-PPU with a data chunk size of 8B is applied. When reducing the data chunk sizes down to 4B, the probability of L0 further raises to 59%.

We also simulated the ECC failure rates of NAND flash generating from program disturb, read disturb and retention time limit, respectively, by using the model in Section 2.2.1. DPA applies 8-bit BCH ECC (4312,4208,8) to every 526-byte data block. Fig. 12(a) shows that the ECC failure rate induced by program disturb keeps below 2×10^{-18} when the P/E cycle

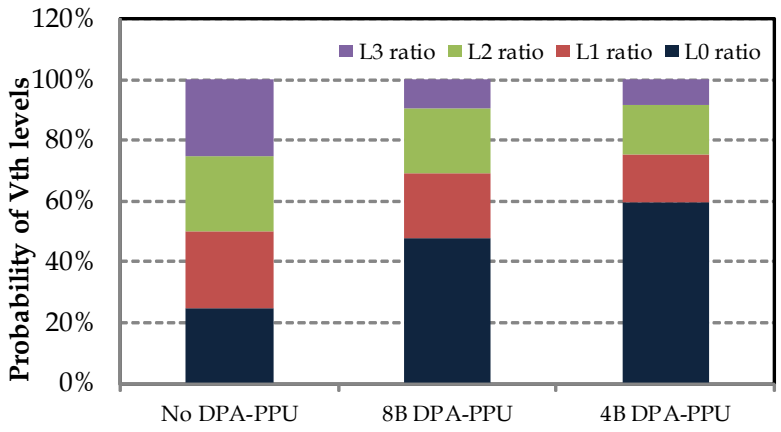
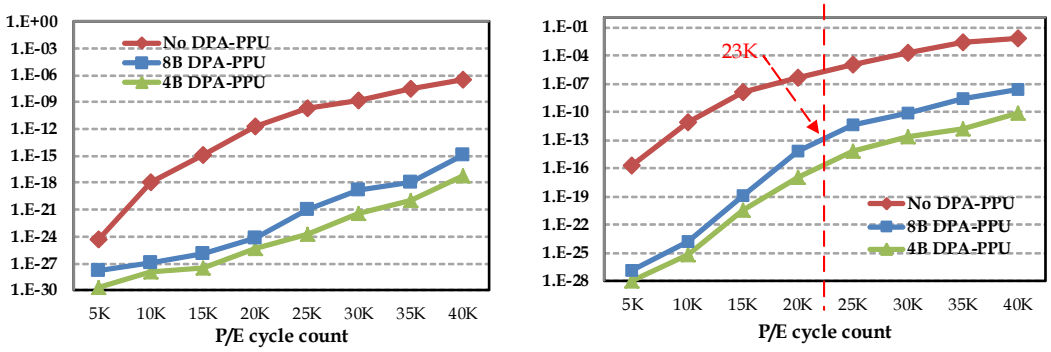


Figure 11: V_{th} distribution after DPA-PPU.

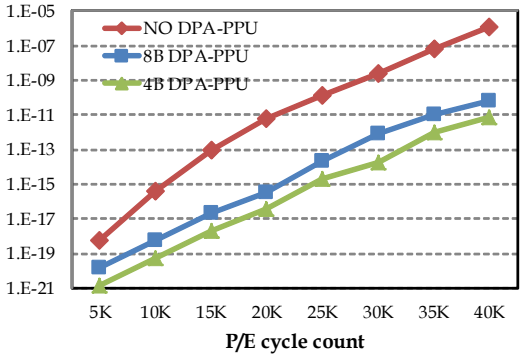
count increases from 5K to 40K as the cell-to-cell interference is significantly minimized by DPA-PPU. Similarly, Fig. 12(b) shows that the ECC failure rate induced by retention time limit is also substantially suppressed by DPA-PPU compared to the system without DPA-PPU. Fig. 12(c) depicts the ECC failure rate induced by read disturb at the read count of 5K. ECC failure rate reduction is still achieved by DPA-PPU even though more cells are placed on L0. It is because the minimization of cell-to-cell interference improves the cell noise margin and more reads can be tolerated. We note that the ECC failure incurred by retention time limit dominates all errors and primarily determines the NAND flash lifetime. When a 8B data chunk size is applied, ECC failure rate reaches the reliability threshold $T_{uber} = 10^{-13}$ when P/E cycle count = 23K; when a 4B data chunk size is applied, the ECC failure rate reaches the T_{uber} when P/E cycle count = 30K. Compared to the system without DPA-PPU where the maximum P/E cycle count = 7.5K (see Fig. 5(b)), the NAND flash lifetime is improved by 3× and 4×, respectively. In all scenarios, reducing the data chunk size always improves the NAND flash reliability.

Fig. 13 demonstrates the relationship between the maximum tolerable read count and P/E cycle count when the T_{uber} is fixed, say, 10^{-13}. The maximum read count of the system without DPA-PPU quickly drops down to zero when the P/E cycle count raises above 18K. By applying DPA-PPU scheme, however, the working range of flash cells is dramatically expanded.



(a) DPA-PPU program disturb ECC failure rate.

(b) DPA-PPU retention time ECC failure rate.



(c) DPA-PPU read disturb ECC failure rate.

Figure 12: The ECC failure rates under different device noise.

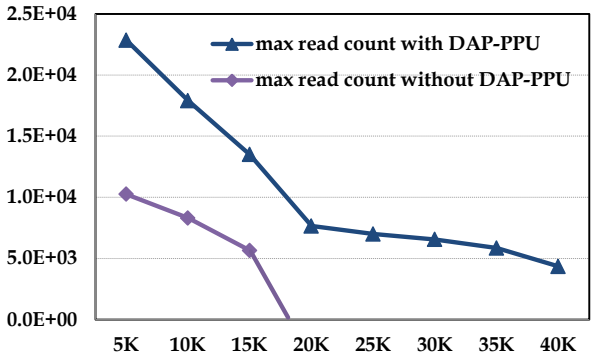


Figure 13: Tradeoff between read count and P/E cycle count.

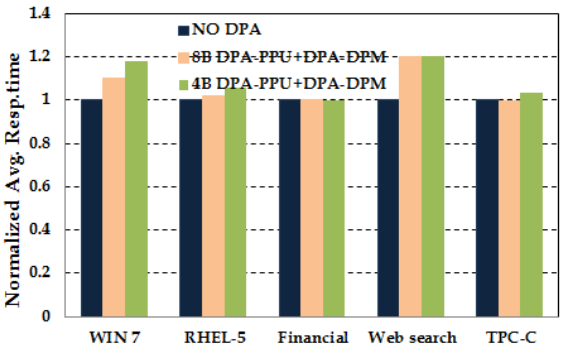
2.6.2 Overheads of DPA

We also evaluate the impact of DPA scheme on system performance under the data chunk size of 4B and 8B. The data chunk size is switched from 8B to 4B when the P/E cycle count reaches 23K (the reliability limit for 8B data chunk size). When the data chunk size is 8B, one third of the redundant bits can be stored in the OOB zone. When the data chunk size changes to 4B, the redundant bits increase and only one seventh of them can be stored in the OOB zone, inducing more page access overhead. 8-bit ECC is applied to each 170-byte data block in the redundant pages and four redundant pages form one RAID-5 stripe.

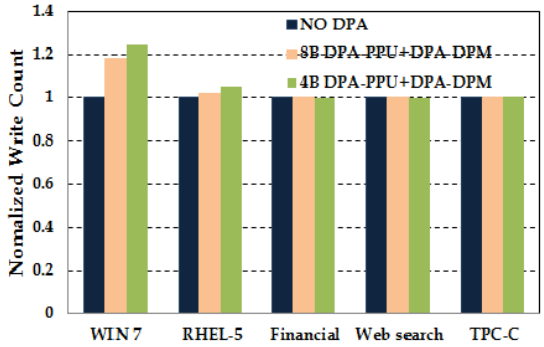
Fig. 14(a) shows that after 23K P/E cycle, compared with the system without DPA, the response time of DPA+DRM with 8B and 4B data chunk size degrades by $\sim 7\%$ and $\sim 10\%$ on average, respectively. The maximum performance degradation $\sim 20\%$ occurs at *Web* workload. One reason for the high overhead is that DPA incurs a considerable amount of redundant page reads. The other reason is that most reads in web workload are large-sized and sequential, where the hit rate of redundant page cache is low. Fig. 14(b) and Fig. 14(c) show the write counts and erase counts of two DPA schemes, respectively. Averagely, under 4B data chunk, DPA+DRM increases the write count and erase count by $\sim 6\%$ and $\sim 4\%$, respectively.

2.7 CHAPTER 2 SUMMARY

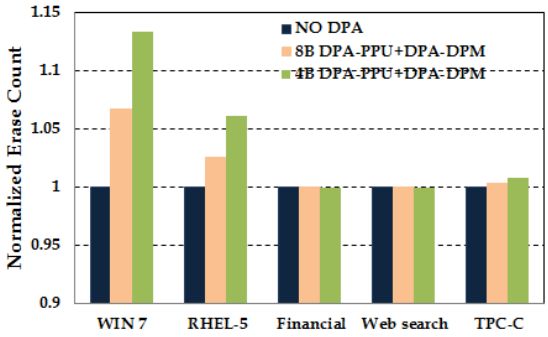
In this chpater, we propose a data pattern aware error prevention technique named DPA to extend the lifespan of NAND flash storage systems. We propose Pattern Probability Unbalance (DPA-PPU) scheme to skew the ratio of 1's and 0's in the stored data so as to place more cells on L0. We also employ Data Redundancy Management (DPA-DRM) scheme to mitigate the performance overhead induced by DPA-PPU. Experimental results show that DPA can prolong NAND flash lifetime by up to 3×.



(a) Write counts of DPA-DRM.



(b) The average response time of DPA-DRM.



(c) Erase counts of DPA-DRM.

Figure 14: The performance overhead of DPA-DRM.

3.0 FLEXLEVEL NAND FLASH STORAGE SYSTEM DESIGN TO REDUCE LDPC LATENCY

In this chapter, we will present FlexLevel technique which reduces Low Density Parity Check (LDPC) incurred read latency. The structure of this chapter is organized as the follows: Section 3.1 presents the preliminary knowledge of LDPC code and related works; Section 3.2 quantitatively estimates LDPC overhead with increase of P/E cycles; Section 3.3 illustrates FlexLevel system architecture. Section 3.4 and Section 3.5 present LevelAdjust at the device level and AccessEval design at the system level, respectively; Section 3.6 presents the experimental results; Section 3.7 summarizes this chapter.

3.1 LDPC CODE AND RELATIVE WORKS

With technology node scaling down, MLC NAND flash BER significantly increases. When technology node is down to 2Xnm, the MLC NAND flash BER reaches up to 10^{-2} [40]. Under such a high BER, traditional hard-decision ECC, e.g., BCH code, cannot meet system reliability requirement any more. Our DPA scheme cannot be applied under 10^{-2} BER either due to high storage overhead of the redundant pages. To enhance error correction capability with reasonable redundancy overhead, LDPC code is introduced. LDPC has a sparse $M \times N$ parity-check matrix. The matrix is represented by a bipartite graph with N variable nodes and M check nodes. Error correction is realized by belief-propagation (BP) algorithm [41]: the decoding messages are iteratively computed and exchanged between variable nodes and check nodes. There are two types of LDPC codes: hard-decision and soft-decision LDPC. Hard-decision LDPC uses binary bits as the decoding message while soft-decision LDPC

employs LLR information. Soft-decision LDPC can achieve better error correction strength than hard-decision LDPC and the error correction capability of the soft-decision LDPC heavily depends on accuracy of LLR information.

In NAND flash memory, only binary information is provided, which severely deteriorates LDPC performance. To enable adoption of soft-decision LDPC, read retry is employed. With this technique, LLR information can be collected by sensing V_{th} with extra reference voltages or soft sensing levels. More extra soft sensing levels generate more accurate LLR information. However, increasing soft sensing levels also increases memory sensing overhead and data transfer time, directly leading to longer read latency: read latency under soft-decision LDPC with extra six soft sensing levels is seven times that of hard-decision LDPC [40]. Long read latency is the major obstacle that excludes LDPC from high performance application.

To address the long read latency issue of LDPC code, several research works are proposed. From the perspective of signal processing, G. Dong et al. proposed entropy coding to reduce data transfer time [42]. However, entropy decoding induces high hardware cost. [41] adopted a nonuniform memory sensing strategy to reduce soft sensing levels. S. Tanakamaru proposed error prediction scheme which substitutes partial programming and partial erase for soft-decision LDPC code [40]. K. Zhao et al. proposed fine-grained progressive and look-ahead sensing scheme to reduce LDPC data transfer overhead [12]. Our work FlexLevel NAND flash storage system design can incorporate these previous works to further reduce LDPC overhead. Our work is inspired by [43], which minimizes BER in phase change memory by V_{th} level reduction. Due to different memory structure and device models, we device an encoding scheme and noise margin adjustment scheme different from [43] to minimize BER. Unlike [43], simple V_{th} level reduction at the device level incurs higher storage overhead. Therefore, we selectively apply V_{th} level reduction to the frequently read data with high BER to maximize the design efficiency.

3.2 MOTIVATIONS

In this section, simulations are conducted to show that soft-decision LDPC overhead is closely related to BER. Therefore, LDPC overhead will be reduced if BER can be minimized. Our

estimation of LDPC overhead is based on the reliability index uncorrectable bit error rate (UBER). Assume a rate-n/m ECC is employed in NAND flash storage system. n and m represent information length and total codeword length, respectively. UBER can be estimated by [2]:

$$uber(k) = \frac{1 - \sum_{i=0}^{k} C_m^i p_c^i (1-p_c)^{(m-i)}}{n}. \tag{3.1}$$

Here, p_c denotes BER of a single NAND flash cell. k is the correctable bit number. Here, p_c denotes BER of a single NAND flash cell. k is the correctable bit number. MLC NAND flash memory BER can be simulated based on the reliability models Eq. 2.1∼2.4 in by Monte Carlo simulation. The simulation is based on the mathematical models of RTN, cell-to-cell interference and retention time noises. These models are expressed in Eq. 2.1∼2.4. For RTN simulation, λ is calculated by $4.0 \times 10^{-4} N^{0.5}$ [31]. We adopt the even/odd bit-line structure to simulate the effect of cell-to-cell interference. In the even/odd bit structure NAND flash memory, capacitance coupling exists in three directions. Coupling ratios in the three directions can be denoted by γ_x, γ_y and γ_{xy}. They are set 0.07, 0.09 and 0.005, respectively [44]. The retention time noise parameters are obtained by fitting the data in [5]. K_s, K_d and K_m are 0.333, 4×10^{-4} and 2×10^{-6}, respectively. t_0 is set one hour. x_0 is V_{th} level 0 and is modeled by Gaussian distribution $N(1.1, 0.35)$ [27]. x is the initial V_{th} after programming and t is storage time. The ISPP verify voltages and the programming step voltages are set 2.55, 3.15, 3.75 and 0.15, respectively [27].

MLC NAND flash BER over P/E cycling is shown in Fig. 15. BER increases with both P/E cycling and storage time. BER immediate after program operation (referred as program BER) increases from 6.72×10^{-4} to 2.29×10^{-3} when P/E cycle count reaches 6000. Based on simulated NAND flash BER, we estimate overhead of qualified LDPC code. The targeted UBER is set 10^{-15} [45]. A rate-8/9 LDPC code is performed to each 4KB data block. According to LDPC performance in [12], we list the required LDPC extra soft memory sensing levels under specific P/E cycle and storage time in Table. 6. 0 means hard-decision LDPC which has no extra soft memory sensing. It is shown that soft-decision LDPC with extra soft memory sensing levels is required after 4000 P/E cycles. Under 6000 P/E cycle count and 1-month storage time, six extra soft memory sensing levels are necessary

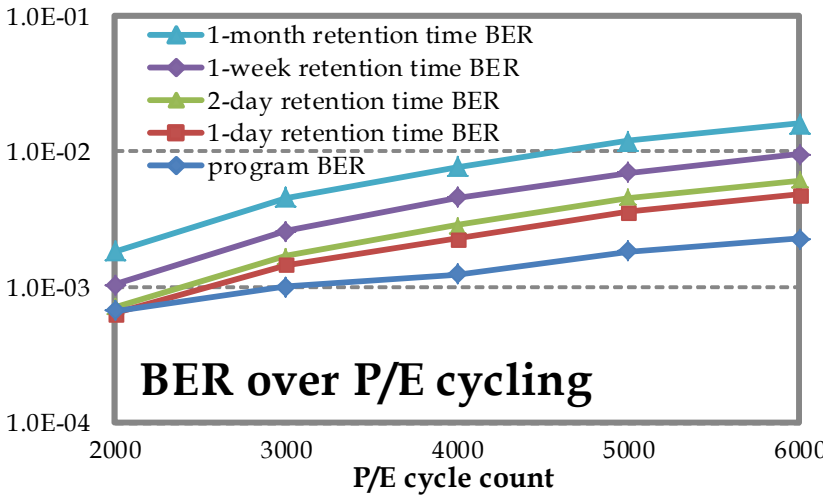


Figure 15: NAND flash memory BER over P/E cycling

to guarantee system reliability. Soft-decision LDPC overhead mainly results from retention time error and increases with storage time and P/E cycles. Intuitively, if we can minimize retention time BER, the incurred read latency can be reduced. In this work, we propose FlexLevel technique to minimize the BER in NAND flash storage system and consequently, reduce the LDPC latency and improve the read performance.

Table 6. Required extra LDPC soft sensing levels

P/E cycles	0 day	1 day	2 day	1 week	1 month
2000	0	0	0	0	0
3000	0	0	0	0	1
4000	0	0	0	1	4
5000	0	0	1	2	4
6000	0	1	2	4	6

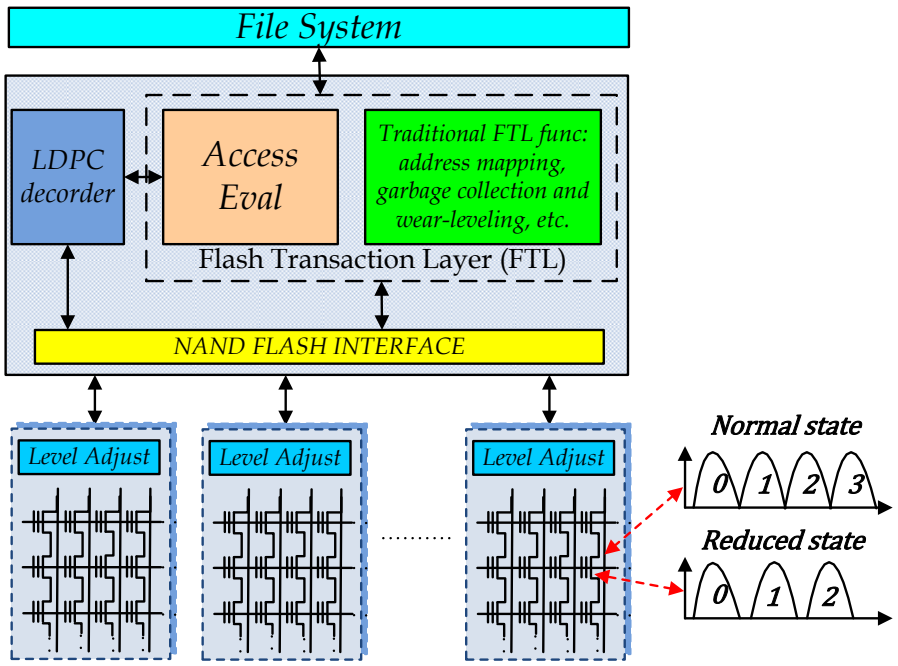


Figure 16: FlexLevel NAND flash storage system overview

3.3 FLEXLEVEL NAND FLASH STORAGE SYSTEM OVERVIEW

Fig. 16 shows the overview of FlexLevel design, including two major components: *LevelAdjust* and *AccessEval*. LevelAdjust technique is proposed to reduce BER at device level, i.e., adjusting cell noise margin by changing the number of V_{th} levels of floating gate transistors. This technique allows one MLC NAND flash cell to have two states: normal state and reduced state. In normal state, the cell has four V_{th} levels, working as regular MLC NAND flash cell. In reduced state, the cell has only three V_{th} levels. However, the BER of the cell in reduced state is reduced by allocating a larger noise margin to each V_{th} level. At early P/E cycling stage, BER is low and all NAND flash cells are in normal state. Following the increase of P/E cycle and storage time, cells may switch to reduced state to control the BER below a threshold. Note that gray code will not be applicable to reduced state as the half of the capacity will be lost. Hence, we developed new coding scheme and bitline structure to maximize the information storage density of the cells in reduced state. What is more,

based on the observation that different V_{th} levels may be associated with different BER, we proposed the non-uniform noise margin adjustment (NUNMA) technique to further reduce the maximum BER that needs to be handled in the design. The reduction of BER results in fewer soft sensing levels needed by LDPC and consequently, improves the system read performance. The LevelAdjust technique details will be discussed in Section 3.4. However, even applying our newly developed coding scheme, V_{th} level reduction introduced by LevelAdjust still cause up to $\frac{1}{4}$ storage capacity loss. To maximize performance improvement with minimized storage capacity loss, we propose AccessEval technique to selectively apply LevelAdjust to the NAND flash cells based on need. AccessEval module is implemented in FTL (Flash Translation Layer), which is a software layer emulating NAND flash as a block device [6]. AccessEval evaluates LDPC overhead for stored data based on their access patterns. For data with access patterns leading to high LDPC overhead, AccessEval manages to store the data in reduced state cells. On the contrary, data with access patterns that lead to low LDPC overhead will be stored in normal state cells. Thereby, soft-decision LDPC induced read latency can be effectively reduced by paying minimum storage loss.

3.4 LEVELADJUST: V_{TH} LEVEL ADJUSTMENT

3.4.1 Basic LevelAdjust Technique

LevelAdjust minimizes BER of a NAND flash cell through V_{th} levels reduction. Two states are introduced to the operations of the cell. In normal state, the cell has four V_{th} levels. It adopts the same even/odd bitline structure and program/erase operation as in regular MLC NAND flash device. Standard gray code is still deployed to map two bits to the four V_{th} levels. In reduced state, the cell has only three V_{th} levels: V_{th} level 0, 1 and 2. Compared with normal state, the cell in reduced state has enlarged noise margin at each V_{th} level and therefore can bear higher noise magnitude.

However, if gray code is still used to map the bits in reduced state cells, each cell can only store one bit. Hence, ReduceCode technique is proposed to maximize the information

storage density of each reduced state cell. We observed that each reduced state cell has three V_{th} levels and two cells indeed can represent nine V_{th} combinations. Therefore, ReduceCode uses eight out of nine V_{th} combinations to represent 3 bits. In this way, two cells can represent 3 bits instead of just 2 bits with gray code.

Table 7. Bit value mapping under ReduceCode

3-bit value	V_{th} I	V_{th} II	3-bit value	V_{th} I	V_{th} II
000	0	0	100	2	2
001	0	1	101	0	2
010	1	0	110	2	0
011	1	1	111	2	1

A mapping scheme between 3-bit value and V_{th} level combinations in a reduced state cell is shown in Table 7. V_{th} I and V_{th} II represent the V_{th} levels of the 1st cell and 2nd cell, respectively. Similar to gray code, ReduceCode aims to minimize BER when V_{th} distortion occurs. Take 3-bit value 101 as an example. It is mapped to V_{th} level 0 in the 1st cell and V_{th} level 2 in the 2nd cell. In case of V_{th} distortion, e.g., the V_{th} level of the 2nd cell changes from level 2 to level 1, the 3-bit value 101 will change to 001, causing only one-bit error. In summary, one level distortion in any of the two cells will cause only one bit error. Thus, bit error is effectively minimized.

A dedicated ReduceCode bitline structure is also designed, as shown in Fig. 17(a). Two neighboring even or odd cells are combined to represent 3 bits and a pair of even cells or odd cells have one MSB and two LSBs totally. Two LSBs from all even cells on one wordline form a "lower page" while two LSBs from all odd cells on the same wordline form a "middle page". Also, the MSBs from all cells on the same word line form "upper page". In NAND flash, program operation is performed in unit of page. Thus, in ReduceCode bitline structure, we propose a new two-step program algorithm to program each page as in Fig. 17(b): in the 1st step, two LSBs, i.e., the lower or middle page is programmed; in the 2nd step, the MSBs, i.e., the upper page is programmed.

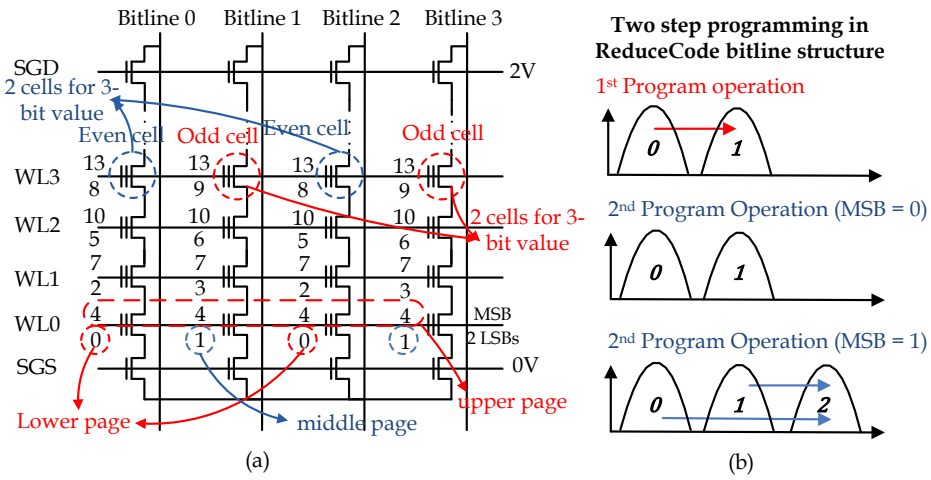


Figure 17: ReduceCode bitline structure.

The V_{th} transitions under two program steps are summarized in Table 8. Here, ΔV_{th} I and ΔV_{th} II denote the V_{th} level transition of the 1st and 2nd cells, respectively. Targeted V_{th} I and targeted V_{th} II denote the V_{th} levels that the cells are programmed to. Before programming, erase operation resets the reduced state cell to V_{th} level 0. During the 1st program step, depending on whether the lower or the middle page needs to be programmed, the even or the odd bitlines will be selected accordingly. The V_{th} level either increases to V_{th} level 1 or remains in V_{th} level 0 based on the stored bit value. During the 2nd program step, all bitlines will be selected. Since the MSBs of all cells form the upper page, the MSBs of all pairs of cells will be programmed. Note that the V_{th} level transition during 2nd program step depends on the least two significant bit values mapped in the 1st program step and MSB. If MSB is 0, V_{th} level transition stops and V_{th} levels remain the same as that after the 1st program step. If MSB is 1, V_{th} level transition follows Table 8. In read operations of ReduceCode bitline structure, V_{th} will be compared with the new read reference voltages.

3.4.2 NUNMA Technique: Non-uniform Noise Margin Adjustment

Besides by reducing V_{th} level number, our LevelAdjust also adopts non-uniform noise margin adjustment (NUNMA) technique to maximize BER reduction efficiency. When NAND flash cells enter post-cycling stage, retention time error starts to dominate the overall BER [2].

Table 8. V_{th} transaction under 2-step programming operation

MSB	two LSBs	targeted V_{th} I	targeted V_{th} II	ΔV_{th} I	ΔV_{th} II	program seq.
-	00	0	0	–	–	1st program
-	01	0	1	–	0→1	1st program
-	10	1	0	0→1	–	1st program
-	11	1	1	0→1	0→1	1st program
1	00	2	2	0→2	0→2	2nd program
1	01	0	2	–	1→2	2nd program
1	10	2	0	1→2	–	2nd program
1	11	2	1	1→2	–	2nd program

Simple V_{th} level reduction, however, is not adequate to inhibit retention time error. To further reduce cell BER, we first analyze the error patterns of MLC NAND flash cells. The error pattern, i.e., bit error occurrence probability, under 1-week/1-month storage time and different P/E cycle counts is shown in Fig. 18. Here, simulation method and NAND flash parameters keep the same as that in Section 3.2. X-axis shows combinations of P/E cycle count and storage time and Y-axis displays bit error occurrence probability breakdown at each V_{th} level. The results clearly show that higher V_{th} levels have larger retention time error occurrence probability: 51% and 30% bit errors occur at V_{th} level 3 and 2 on average. This implies that V_{th} in high levels decreases faster than that in low levels. Therefore, allocating V_{th} noise margins uniformly among all V_{th} levels may not an optimal solution as system reliability is only limited by the maximum BER.

Based on this observation, we propose NUNMA technique to maximize BER reduction efficiency. The main idea of NUNMA is to optimize the noise margins of different V_{th} levels globally. A V_{th} level region is confined by its lower and upper read reference voltages. Originally, program verify voltage is set to close to lower read reference voltage and the

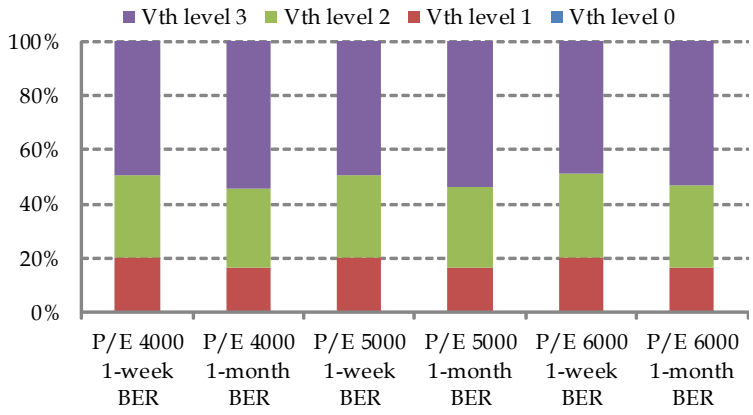


Figure 18: Bit error occurrence probability at four V_{th} levels.

V_{th} distribution is placed in the center of its V_{th} level region, as shown in Fig. 19(a). The decrease in V_{th} with storage time increase results in retention time errors. In order to improve retention time noise margin, the programmed V_{th} distribution should be shifted to right by increasing the verify voltage while maintaining the read reference voltages unchanged. As a result, the programmed V_{th} will be much higher than lower reference voltage, allowing the enhanced noise margin and better tolerance to charge loss, as shown in Fig. 19(b). However, increasing verify voltage may cause the level 1 V_{th} to exceed its upper read reference voltage, introducing cell-to-cell interference errors. As shown in Fig. 18, the retention time BER at low V_{th} levels is lower than that at high V_{th} levels. Hence, it is safe to allocate relatively small retention time noise margin to low level V_{th}'s and large retention time noise margin to high level V_{th}'s. A low verify voltage in V_{th} level 1 together with a high verify voltage in V_{th} level 2 can be employed, as shown in Fig. 19(c). Both cell-to-cell interference and retention time BER are reduced. NUNMA technique can be easily implemented in NAND flash as programming verifying and read reference voltages are all adjustable [46].

3.4.3 LevelAdjust Overhead Evaluation

This section, the hardware and storage overhead of LevelAdjust is evaluated. The hardware overhead introduced by LevelAdjust is the logic gates that are needed to implement Reduce-

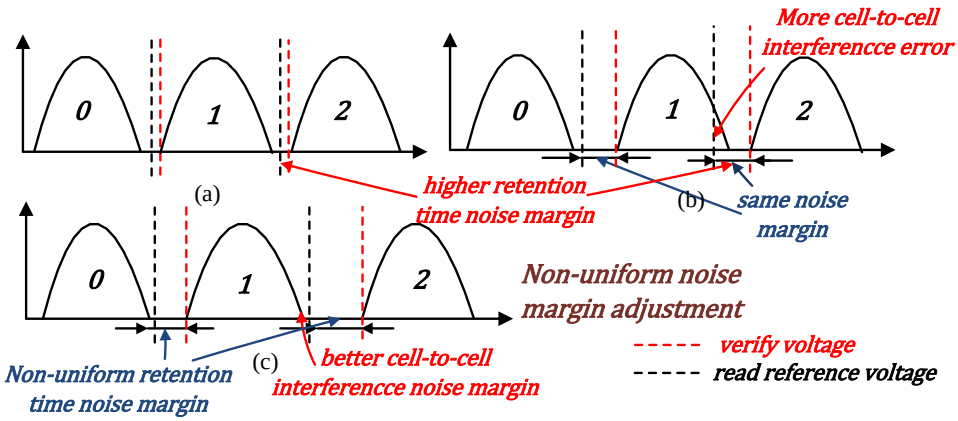


Figure 19: NUNMA technique.

Code circuit. We assume that $V_{11}V_{10}$ and $V_{21}V_{20}$ represent V_{th} levels of two neighboring cells. $b_2b_1b_0$ denote 3-bit value. The logic expression of encoding circuit is listed in Eq. 3.2∼3.5.

$$V_{11} = b_2b_1 + b_2\bar{b_0}. \tag{3.2}$$

$$V_{10} = \bar{b_2}b_1. \tag{3.3}$$

$$V_{21} = b_2\bar{b_1}. \tag{3.4}$$

$$V_{20} = \bar{b_2}b_0 + b_1b_0. \tag{3.5}$$

The logic expression of decoding circuit is listed in Eq. 3.6∼3.8.

$$b_2 = V_{11}\bar{V_{10}}\bar{V_{21}} + \bar{V_{10}}V_{21}\bar{V_{20}}. \tag{3.6}$$

$$b_1 = (V_{11} \bigoplus V_{10})\bar{V_{21}}. \tag{3.7}$$

$$b_0 = \bar{V_{11}}\bar{V_{21}}V_{20} + \bar{V_{10}}\bar{V_{21}}V_{20} + \bar{V_{11}}\bar{V_{10}}V_{21}\bar{V_{20}}. \tag{3.8}$$

The circuit only employs less than 100 gates. ReduceCode encoding and decoding overhead is only one clock cycle, e.g., 5ns for a 200MHz clock frequency. The induced overheads on data transfer and sensing latency (normally tens of microseconds) can be also ignored. Another hardware overhead is interface command decoding circuit. Since one cell can have two states, some logic is needed to configure cells into normal state or reduced state. However, the incurred hardware overhead is also very marginal.

The major overhead of LevelAdjust is the capacity loss incurred by V_{th} level reduction. In reduced state, two MLC NAND flash cells are combined to represent 3 bits, leading to 25% storage density reduction compared to normal state cell. This capacity loss has to be compensated since storage system capacity must be consistent to file system. Although over-provision space [47] may be used to compensate the capacity loss, it may cause severe write performance degradation. In order to minimize such capacity loss, AccessEval technique is introduced at system level, as we present in next section.

3.5 ACCESSEVAL: ACCESS PATTERN EVALUATION

3.5.1 AccessEval Overview

LevelAdjust introduces inevitable storage capacity loss of NAND flash cells. To reduce the storage overhead, AccessEval restricts the application of LevelAdjust to a minimum number of NAND flash cells that really need. In reality, not every data contributes equally to the overall LDPC overhead. Therefore, if we can identify the data which contribute to the majority of the LDPC overhead and apply LevelAdjust to only these data, i.e., storing them in reduced state pages, the impacts of LevelAdjust will be limited to a small scale. The LDPC overhead can be still reduced while the incurred system storage capacity loss is minimized. For this purpose, we propose AccessEval technique that can selectively apply LevelAdjust to the data based on their needs.

The architecture of AccessEval design is shown in Fig. 20. An AccessEval module consists of three components: IWFR identifier, ReducedCell pool and AccessEval controller.

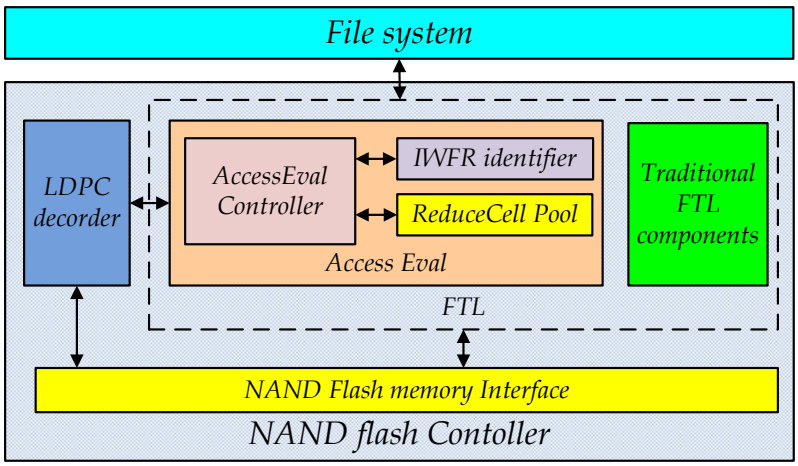


Figure 20: AccessEval architecture

ReducedCell pool is a data structure recording the data stored in reduced state pages. The size of ReducedCell pool limits the maximum number of reduced state pages. AccessEval controller manages the data allocation between reduced state pages and normal state pages. During read operations, once a data is identified as IWFR data, it will be stored in a reduced state page. If all reduced state pages are used, ReducedCell pool will first evict the least-recently-accessed data from the reduced state pages to normal state pages, and upcoming IWFR data is stored in the reduced state pages.

Note that in AccessEval, data migration between reduced state pages and normal stage pages incurs extra program and erase operations. Improving the identification accuracy of IWFR data becomes essential to enhance the AccessEval efficiency. More details on the design of IWFR identifier will be discussed in following subsections.

3.5.2 IWFR Data Identification

One of the main challenges in AccessEval is how to identify data that will contribute to the majority of overall LDPC overhead. The LDPC overhead contributed by a data depends on the LDPC overhead per read and the read frequency of this data. Here the LDPC overhead per read is determined by the number of extra sensing levels needed to decode this data correctly. Based on Table 6, the number of extra sensing levels is mainly decided by the

retention time BER of this data. When the write frequency of a data is infrequent, the storage time of the data becomes long, causing a high retention time BER. Hence, we can conclude that a data with low write frequency and high read frequency will contribute more to the overall LDPC overhead. Accordingly, such a type of the data is defined as IWFR (infrequently-write-and-frequently-read) data, which shall be identified by IWFR identifier and intuitively stored in reduced state pages.

Table 9. Workloads access pattern characterization

Workload	Application	Read rt.	IWFR rt. in read data
fin-2	OLTP	78.46%	0%
websearch-1	web server	99.9%	100%
websearch-2	web server	99.98%	100%
prj-1	research project	72.3%	69.23%
prj-2	research project	62.3%	59.73%
user-1	p2p client, office	43.7%	41.5%
user-2	p2p client, office	52.03%	44%

Many techniques have been invented to recognize the frequently read and write data in [48, 49, 50]. All these techniques adopt only one constraint (e.g., frequently-read or frequently-write) in data identification process. However, our initial analysis shows that IWFR data are not necessarily frequently-read data. To reveal the difference between IWFR and frequently read data, access patterns under eight workloads are investigated. The workload statistics data is collected up to 4 days. Statistics about fin-2 workload is collected within 1 day. Websearch-1 and websearch-2 workload data is collected within 4 days. Other

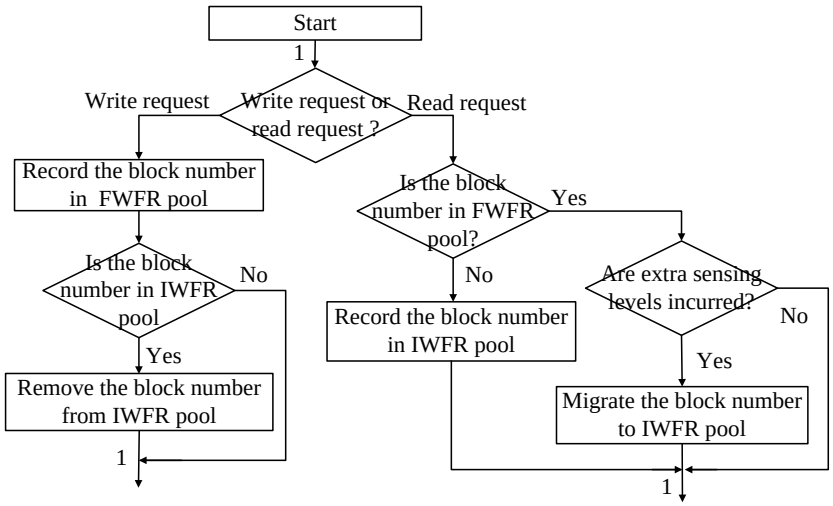


Figure 21: IWFR identification flow.

workload data is collected within 2 days. We define frequently read data as the read data with 20% highest count [48]. Here, the collected IWFR data includes write-once-multiple-read and read-only data. The read patterns are shown in Table 9. The third column shows the ratio of frequently read data among all data. The fourth column shows the ratio of IWFR data among all read frequently data. From the table, we can see that frequently read data is not necessarily IWFR. IWFR data ratio varies a lot with workloads. In fin-1 and fin-2 workload, there is less than 5% IWFR data in frequently read data. Except workload websearch-1 and websearch-2, less than 70% frequently read data is IWFR. Based on the analysis above, we can conclude that the previous techniques cannot be directly incorporated in AccessEval to identify IWFR data.

To achieve high identification accuracy, we design our own IWFR identifier. Two pools are used to facilitate the identification of IWFR data, including 1) IWFR pool, which records the block number of IWFR data; and 2) FWFR (frequently-write-frequently-read) pool, which records the block number of FWFR data. Here, FWFR pool is employed to improve the IWFR identification accuracy by preventing FWFR data from entering IWFR pool. IWFR identification flow is depicted in Fig. 21. When a write request is received, the block number of the data to be programmed is stored in FWFR pool. If this new block number already exists in FWFR pool, no action will be taken. Otherwise, this block number will

be inserted to FWFR pool. If there is no space in FWFR pool for the newly inserted block number, the least recently and least frequently accessed block number is evicted from the pool to reserve the space for this block number. IWFR pool is then searched for this block number. If this block number is also in IWFR pool, it will be removed from the IWFR pool. The hypothesis is that a newly written data is relatively "fresh" and less likely to have error. Thus, it does not need to be stored in a reduced state page. When a read request is received, we first check whether the block number of the data to be read has been recorded in FWFR pool. If it is in FWFR pool, then we will continue to check whether extra sensing levels are needed to correctly decode this data. If so, the block number will migrate from FWFR pool to IWFR pool. Otherwise, no action will be taken since the data still has low error rate. The hypothesis is that a data in FWFR that needs extra sensing levels is likely has been stored for long time and been read for many times. Therefore, it should be categorized as a IWFR data. If the data is not in FWFR pool, its block number needs to be recorded in IWFR pool if we have not done so. Again, if there is no free space in IWFR pool, the least recently and least frequently accessed block number need to be evicted to reserve the space for this new block number.

To further improve IWFR data identification accuracy, we must take into account sensing levels of the data. In IWFR pool, the data may have been stored for different time periods. The longer the data have been stored for, the higher extra sensing levels they may need. By considering both access patterns and sensing levels, a modified LDPC overhead estimation rule can be created as follows: in IWFR pool, the access frequency of a data is divided into N levels (L_f) while its soft sensing levels are divided into M buckets ($L_{sensing}$). LDPC overhead is measured by $L_f \times L_{sensing}$. If the LDPC overhead of a data exceeds a pre-defined threshold, this data will be stored in reduced state pages.

3.5.3 AccessEval Overhead Discussion

Although the capacity loss incurred by LevelAdjust at device level is 25%, applying AccessEval can successfully reduce the capacity loss down to 6% at system level, as we shall show in Section 3.6. The application of LevelAdjust is constrained to only 25% of total NAND flash

Table 10. Non-uniform LevelAdjust configuration

Scheme	V_{pp}	$V_{verify1}$	$V_{verify2}$	$V_{read-ref1}$	$V_{read-ref2}$
NUNMA 1	0.15	2.71	3.61	2.65	3.55
NUNMA 2	0.15	2.70	3.65	2.65	3.55
NUNMA 3	0.15	2.75	3.70	2.65	3.55

pages. The write overheads caused by the reduced over-provisioning space is very marginal. More details on the relevant experiments can be found in Section 3.6. In AccessEval, ReducedCell pool is stored in data buffer (DRAM) of NAND flash storage system [47] for fast access. Assume that each entry in ReducedCell pool is 4 bytes. The required storage size is equal to $\frac{4\times S_{rs}}{S_{page}}$. here S_{rs} and S_{page} represent the size of the data stored in reduced state pages and the size of a NAND flash page, respectively. Assume up to 32GB data needs to be stored in reduced state pages and the page size is 16KB, ReducedCell pool only occupies 8MB.

3.6 EXPERIMENTAL RESULTS

3.6.1 LevelAdjust Efficiency

Experiments were performed to evaluate the effectiveness of LevelAdjust in LDPC overhead reduction. The values of BER are obtained from Monte-Carlo simulations. Three NUNMA configurations are explored to find out the optimal device parameters for BER reduction. The program verify and read reference voltages of three NUNMA configurations are listed in Table 10. Regular MLC NAND flash cell (i.e., the normal state cell) is used as the baseline in our comparison. Parameters adopted in the experiments keep the same as that in Section 3.2. We first simulate program BERs and retention time BERs before and after applying LevelAdjust and then evaluate the corresponding LDPC overheads based on the simulated BERs.

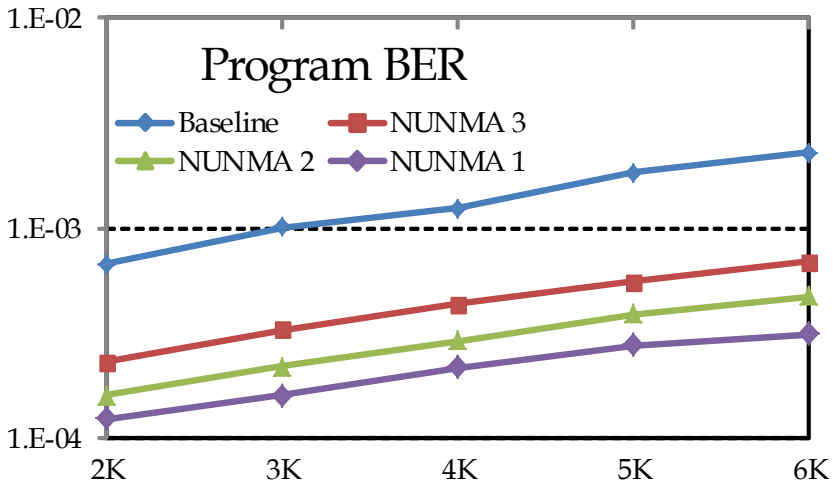


Figure 22: Program BER in reduced state cells.

Program BERs of reduced state cells under different configurations are shown in Fig. 22. We found that compared with the baseline, program BERs can be reduced by up to 6× in NUNMA 1 due to enhanced noise margin. The program BER of NUNMA 3 is 50% and 20% higher than NUNMA 1 and 2, respectively. This is because that the verify voltage in NUNMA 3 is higher than that in NUNMA 1 and 2, causing more prominent cell-to-cell interference. Based on the result in [12], the BER limit that triggers extra sensing levels is 4×10^{-3}. Nonetheless, the program BERs of three NUNMA configurations are all lower than the limit so that none of them incurs extra sensing levels during programming.

The simulated retention time BERs of reduced state cells under three NUNMA configurations are shown in Table 11. On average, BERs are reduced by 2×, 5× and 9× under the three NUNMA configurations, respectively. NUNMA 3 achieves the lowest retention time BER because high verify voltage provides more retention time noise margin. The highest retention time BER of NUNMA 3, i.e., 1.51×10^{-3}, occurs after 1 month and 6000 P/E cycles. Again, it is lower than the BER limit that incurs extra sensing levels. Among all NUNMA configurations, NUNMA 3 achieves the lowest combined program and retention time BERs, which correspond to the minimum LDPC overhead. No extra sensing levels will be required.

Table 11. BER comparison under three NUNMA configurations

P/E cycles	**scheme**	**1 day**	**2 days**	**1 week**	**1 month**
2000	Baseline	0.000638	0.000715	0.00103	0.00184
	NUNMA 1	0.000370	0.000453	0.000827	0.00149
	NUNMA 2	0.000167	0.000173	0.000243	0.000330
	NUNMA 3	0.000120	0.000133	0.000167	0.000181
3000	Baseline	0.00146	0.00169	0.00260	0.00459
	NUNMA 1	0.000677	0.000860	0.00143	0.00249
	NUNMA 2	0.000343	0.000367	0.000570	0.000807
	NUNMA 3	0.000237	0.000257	0.000293	0.000390
4000	Baseline	0.00229	0.00284	0.00456	0.00778
	NUNMA 1	0.00117	0.00149	0.00240	0.00402
	NUNMA 2	0.000443	0.000633	0.000820	0.00150
	NUNMA 3	0.000327	0.000343	0.000457	0.000633
5000	Baseline	0.00359	0.00457	0.00699	0.0120
	NUNMA 1	0.00177	0.00233	0.00349	0.00545
	NUNMA 2	0.000690	0.000853	0.00123	0.00227
	NUNMA 3	0.000460	0.000540	0.000713	0.00109
6000	Baseline	0.00484	0.00613	0.00961	0.0161
	NUNMA 1	0.00218	0.00288	0.00446	0.00672
	NUNMA 2	0.00100	0.00131	0.00192	0.00324
	NUNMA 3	0.000623	0.000627	0.000973	0.00151

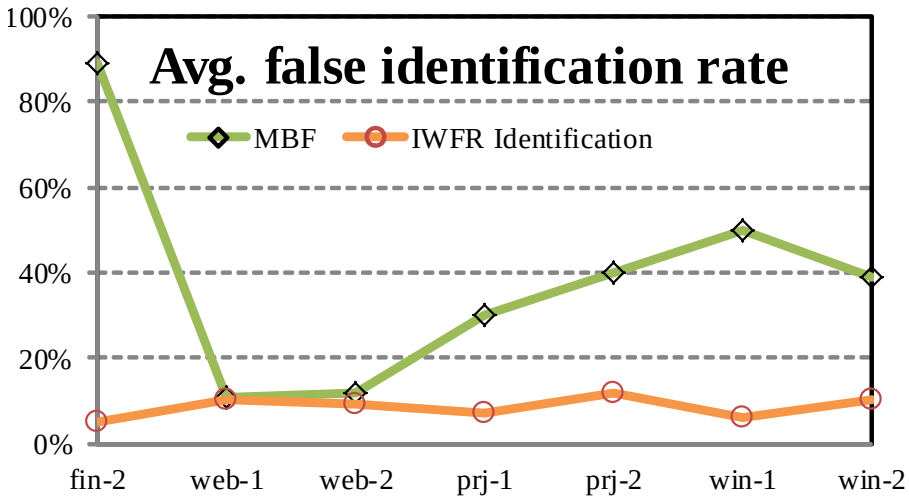


Figure 23: Average false identification rate of IWFR identification technique.

3.6.2 AccessEval Performance Evaluation

In this section, we will evaluate the effectiveness of the proposed AccessEval design. We will first evaluate the accuracy of IWFR identification technique since AccessEval efficiency heavily depends on IWFR identification. We will compare read and overall performance gain under the systems with and without AccessEval design. In the experiments, we will also investigate how P/E cycle affect AccessEval efficiency. Finally, we will evaluate AccessEval's impact on flash memory endurance. We assume that the target UBER is 10^{-15}. In the simulated system, a 8/9-rate LDPC (512B coding redundancy per 4KB user data) is employed to protect data integrity. The storage system requires that NAND flash be used under 6000 P/E cycle count.

First we show the IWFR identification technique efficiency by evaluating false identification rate. We adopt MBF [48] to identify IWFR data in read-only pool. We set two L_f and two $L_{sensing}$, respectively. In IWFR identification experiment, sensing levels of 1-2 and 4-6 belong to $L_{sensing}$ level 1 and level 2. We set two L_f levels 3 and 4. The pre-defined soft-decision cost threshold is 4. The simulation result is shown in Fig. 23. The false identification rate of the proposed IWFR identification technique is only 10% and decreases by 4.5× compared with MBF. The maximum improvement occurs in fin-2 workload since most

frequently read data is FRFW in this workload. Compared with MBF, the average false identification rate is reduced by 4.5×. Under read-only intensive workload web-1 and web-2, false identification rate also remains unchanged.

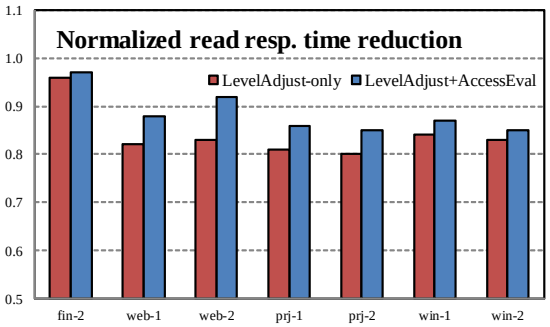
Table 12. MLC NAND flash specification

Capacity	Block Size	Block Number	Page Size
	1MB	4096	16KB
Timing	Program Latency	Read Latency	Erase Latency
	900 μs	44μs(normal state) 36μs(reduced state)	3.5ms

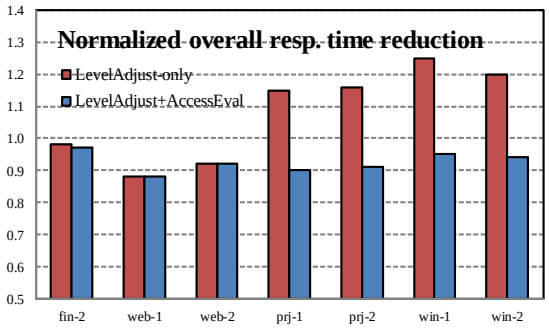
The achieved performance gain can be obtained by comparing the read and overall average response time of the system before and after LevelAdjust and AccessEval are applied. The simulations on AccessEval performance are performed on the simulator Flashsim [51]. The proposed AccessEval design is incorporated into the simulator and a 256GB NAND flash storage system is emulated. Parameters of regular MLC NAND flash and normal state cells are summarized in Table 12. Extra sensing levels can be obtained based on the simulation results in Table 6. The P/E cycle of NAND flash memory is set to 6000. The timing overhead of each sensing level is set to 8 μs and data transfer time is set to 20 μs [12]. In reduced state cells, NUNMA 3 configuration is adopted and no extra sensing overhead is introduced. The LDPC-SSD scheme [12] is employed as our LDPC design baseline and the UBER and LDPC configuration listed in Section 3.2 is used. Two storage system configurations are tested: 1) the one only has LevelAdjust design (LevelAdjust-only) and 2) the one incorporates both LevelAdjust and AccessEval (LevelAdjust+AccessEval). We assume the storage system has 30% over-provisioning portion. In AccessEval, the size of storage space that can be used for LevelAdjust is limited to 64GB. After LevelAdjust, the capacity of this portion is reduced to 48GB with 16GB capacity loss. The benchmarks in Table 9 are used in the experiments.

Fig. 24(a) shows the read response time reduction achieved in our simulated two configurations. Compared with baseline, LevelAdjust can reduce read latency by 14% on average

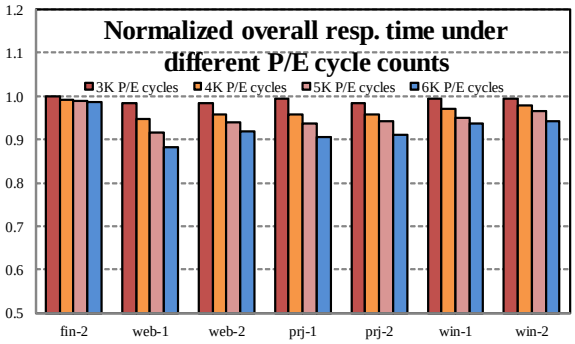
with 25% capacity loss. The maximum response time reduction occurs at two read-intensive workloads: web-1 and web-2. As a comparison, the average read response time reduction in AccessEval+LevelAdjust is 10%. Although it is slightly lower than that achieved in LevelAdjust, AccessEval+LevelAdjust successfully reduces the capacity loss down to 16GB, or 6.25% of the total NAND flash storage system capacity.



(a) Normalized read average response time under the Flex-level design.



(b) Normalized overall average response time under the Flex-level design.



(c) Normalized average response time under different P/E cycle counts.

Figure 24: The performance improvement of the Flex-level design.

The reduction of the overall response time in different configurations, including both read and write latencies, is summarized in Fig. 24(b). In order to compensate the capacity loss, part of the over-provisioning space is used as the normal storage capacity in both configurations. In LevelAdjust, if the 25% capacity loss is fully compensated, the remaining over-provisioning space is only 5% of the normal system storage capacity. The signif-

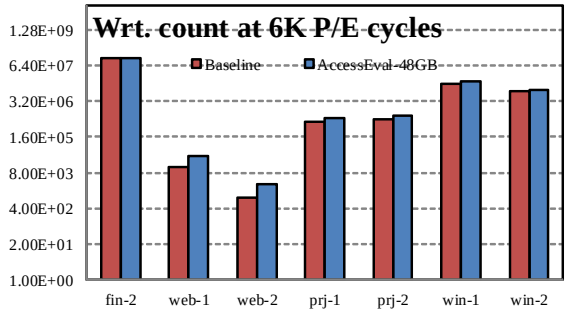
icantly reduced over-provisioning space dramatically increases garbage collection frequency and therefore prolongs write latency. In the last 4 benchmarks, for example, the overall system response time of LevelAdjust is even longer than the baseline. It indicates that the incurred write latency increase even exceeds the read latency reduction achieved by LevelAdjust. In AccessEval+LevelAdjust, however, the capacity of over-provisioning space only slightly decreases (i.e., 30% $\rightarrow$ 23.75%). The impact on garbage collection and write latency is small. The overall system response time indeed reduces by 8% on average. Experiment results also show that the performance gain of AccessEval+LevelAdjust increases with P/E cycle count compared to the baseline system. As shown in Fig. 24(c), the average response time reduction achieved by AccessEval+LevelAdjust w.r.t. baseline system increases from 2% to 11% on average when the P/E cycle increases from 3000 to 6000.

Finally, we evaluate the impact of our techniques on system endurance. Simulation is carried out at a P/E cycle of 6000. Fig. 25(a) shows the write count increases with LevelAdjust+AccessEval, which is only 6% on average. The write count increase comes from the data migration between normal state cells and reduced state cells. The maximum relative write increase happens in web-1 and web-2 workloads simply because their original write numbers are low. Fig. 25(b) shows the erase count increases with LevelAdjust+AccessEval. On average, the erase count increases by 19% across all the simulated workloads. Although web-1 and web-2 has high relative increase in write count, their erase counts almost unchange. This is because the actual write count number is too small to invoke considerably large volume of garbage collections. Since LevelAdjust+AccessEval only applies when the system BER is high enough to incur extra sensing levels, its impact on system lifetime is quite marginal: Table 6 shows that LevelAdjust+AccessEval is needed only when the P/E cycle exceeds 4000. Hence, the average lifetime reduction across all the workloads is only 7%, as depicted in Fig. 25(c).

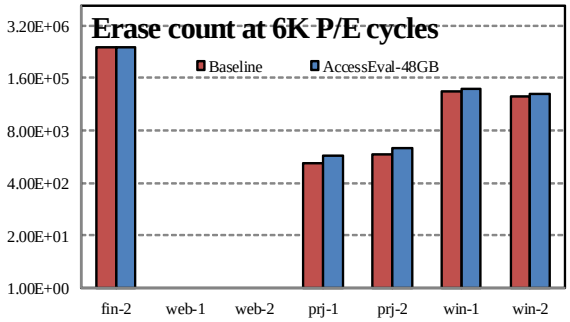
3.7 CHAPTER 3 SUMMARY

In this work, we propose FlexLevel technique to reduce LDPC-induced read latency. We propose device-level LevelAdjust technique to reduce BER via V_{th} level reduction. By minimiz-

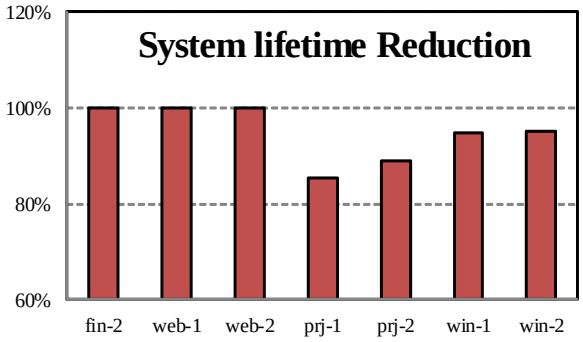
ing BER, extra sensing levels can be effectively reduced and read performance is improved. To balance performance improvement and density loss, we propose AccessEval technique. Simulation results show that compared with the best prior works, the proposed design can achieve read speedup by up to 11% with negligible capacity loss.



(a) The write count increase under LevelAdjust+AccessEval technique.



(b) The erase count increase under LevelAdjust+AccessEval technique.



(c) The lifetime reduction under LevelAdjust+AccessEval technique.

Figure 25: The lifetime cost of LevelAdjust+AccessEval technique.

4.0 PERFORMANCE OF OBJECT BASED NAND FLASH STORAGE SYSTEM

In Chapter 4, we will present our works on optimization of object-based NAND flash storage system. This chapter is organized as follows: Section 4.1 presents the basics of NAND flash memory and object-based NAND flash device; Section 4.2 and Section 4.3 introduce the motivation for our research and related works, respectively; Section 4.4 describes the proposed MLGC, virtual B+ tree and diff cache in detail; Section 4.5 introduces our implemented simulation platform; Section 4.6 presents the experimental results; Section 4.7 summarizes this chapter.

4.1 BACKGROUND

4.1.1 Basics of NAND Flash Memory

The NAND flash memory adopts a block-page structure. A NAND flash memory is divided into a number of blocks. Each block is divided into a number of pages, whose size ranges from 512B to 16KB [52]. The erase operation is conducted by unit of block while the program operation is performed by the unit of page. A NAND flash memory page (i.e., *a physical page*) consists of a data area and out-of-band (OOB) area. The data area stores the page data; the OOB area stores page metadata such as page data information and ECC parity [10][53]. The page metadata and the page data are written together to a NAND flash page. Due to the block-page structure and out-of-place update, write amplification is introduced. In the NAND flash memory based storage system, garbage collection is employed to reclaim dirty

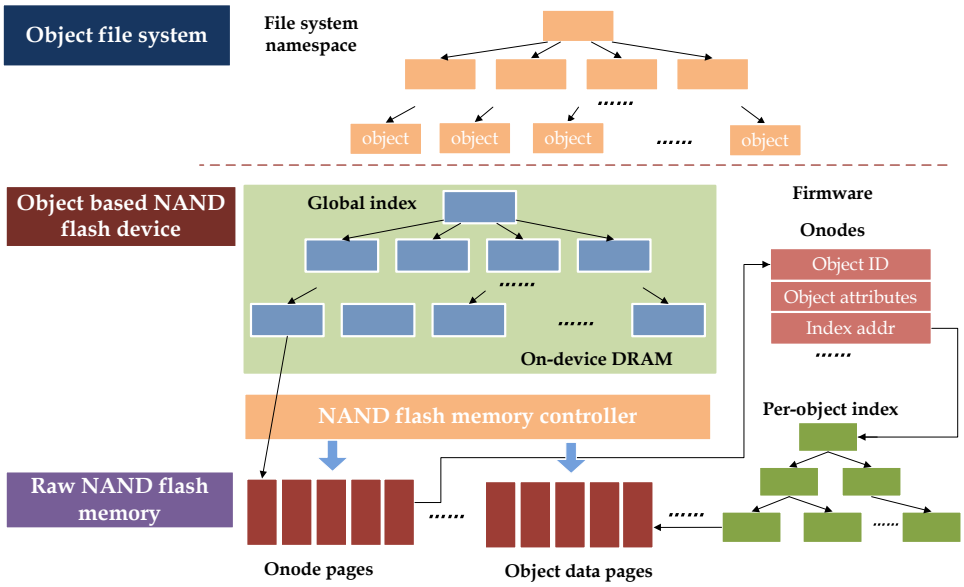


Figure 26: The architecture of object-based storage system.

blocks for upcoming write requests. Usually, a dirty block has both valid and invalid pages. The valid pages have to migrate to a clean block before erase operation. Hence, more data is written than the requested, which is called write amplification. Write amplification reduces system endurance and degrades write performance.

4.1.2 Basics of Object-based NAND Flash Device

To improve garbage collection efficiency and reduce write amplification, an object-based NAND flash storage model is proposed in the previous works [19]. The architecture of the object-based NAND flash storage model is shown in Fig. 26. In this model, the *object* is the basic storage unit. An *object* denotes a variable-sized data container, e.g., a file, which can be uniquely identified by an 64-bit object ID. Each object includes *data* and *attributes* [54]. The object data contains file data with variable-length. The object attributes, stored in *inode* describe file metadata such as ownership and access control.

The object-based model contains an object file system and an object-based NAND flash device (ONFD). The object file system only maintains the name space. The ONFD manages object storage. The ONFD consists of raw NAND flash memories, a NAND flash mem-

ory controller, an on-device DRAM and a firmware. The NAND flash memory controller provides datapath and applies data protection, e.g., ECC. DRAM is deployed to buffer the frequently accessed data. The firmware handles object management and NAND flash memory management.

In the ONFD, the object data is accessed via object metadata – per-object indices and onodes. Like the object data, per-object indices and onodes are also stored in the NAND flash memories. The per-object maintains the physical addresses of the object data. To maximize the space efficiency, the per-object index is implemented with the extent based B+ tree [21]. The onode contains the inode and the address of the per-object index root node page. The physical addresses of onodes are maintained in a global index. The global index, with a B+ tree data structure, is stored in DRAM for fast access.

Besides object management, ONFD also manages NAND flash memory via physical page allocation, garbage collection and wear-leveling. The ONFD adopts the log-structured page allocation method [55]. The physical storage space is divided into a number of chunks. A chunk contains one or more NAND flash blocks. Each chunk has a chunk metadata. The chunk metadata includes chunk states, valid page count and page bitmap table [10]. The chunk is the basic garbage collection unit. When the number of clean chunks is under a threshold, garbage collection is invoked to reclaim the dirty chunks. Usually, a greedy algorithm is adopted [56]: the chunk with the fewest valid pages is selected for reclamation. During chunk reclamation, the pages which are marked valid in the page bitmap table migrate to a clean chunk before the dirty chunk is erased. To reduce data migration of garbage collection, object data and object metadata are stored in different chunks due to different data access patterns [19].

4.2 MOTIVATION

In this section, we introduce the design challenges in the ONFD architecture mentioned in Section 4.1. In the ONFD, two write amplification causes, *onode partial update* and *cascading update*, are identified by leveraging object semantics. The onode size, about one hundred

bytes, is smaller than the physical page size. To reduce internal fragmentation, more than one onode is stored in a physical page [19]. Update to an onode causes partial page update: The old onode in the physical page is invalidated while the un-updated onodes in the same page remains valid. The existing ONFD only allows page-unit invalidation since valid status is maintained at the page and block granularity [10]. Hence, to invalidate the entire page, the remaining valid bytes have to migrate, directly increasing write overhead [47]. The performance degradation is aggravated with increase of physical page size.

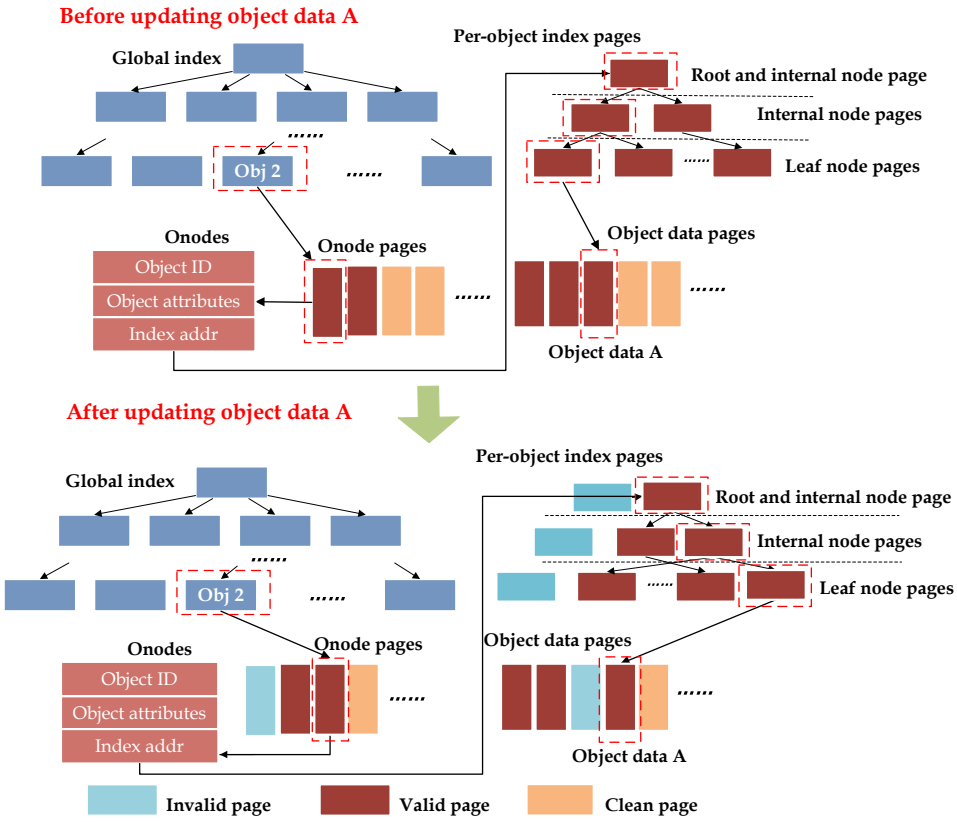


Figure 27: An example of cascading update.

Another write amplification cause is cascading update. When an object data is updated, the new address is updated in the corresponding per-object index. The cascading update occurs within the per-object index due to the wandering tree issue [22]. The per-object index is implemented with the extent-based B+ tree [21]. The leaf node page records the physical addresses of object data. The internal node page maintains the physical addresses of the

child node pages. When a child node page is updated, the new address will be updated in the parent node page in a cascading manner until the root node page is reached. In addition, updating the root node page also incurs onode update. Although update to the object metadata is only several bytes, two or more pages have to migrate entirely, significantly degrading write performance. An example of the cascading update is shown in Fig. 27. Updating object data page A of object 2 incurs update to three per-object index pages. The new address of the per-object index root node page is updated in the corresponding onode. Therefore, total four object metadata page writes are induced.

The cascading update also increases the overhead of garbage collection. During garbage collection, moving a object data page induces write operations to the corresponding object metadata pages; migration of a per-object index node page involves update to the associated parent node pages and the corresponding onode.

4.3 RELATED WORKS

To reduce partial page update and cascading update induced data migration, several works were proposed. [47] proposed to merge partial page updates by byte-addressable emerging non-volatile memory such as ReRAM, STT-MRAM and phase change memory. However, the emerging non-volatile memories are either expensive or not under mass production. [10] identifies object data induced partial page update under the byte-unit access interface. To mitigate the partial page update, [10] proposed to compact partial object data updates in diff-pages before merging them with the un-updated object data. However, this scheme is not applicable to onode partial update.

[22] proposed μ tree to mitigate cascading update of the B+ tree. The μ tree realizes write operation reduction by compacting the most recently updated nodes in one page. However, with the reduced node size, μ tree increases internal fragmentation. The search overhead of μ tree also increases due to increase of the tree height. [10][57] proposed to delay and merge updates to B+ tree with a cache in main memory. However, this lazy update scheme significantly increases main memory consumption. To mitigate the data migration

of onode partial update and cascading update, we propose the Data Migration Minimization (DMM) device design. Compared with the previous works, our DMM device design can more effectively mitigate the write amplification with only marginal memory consumption.

To evaluate the efficiency of the proposed DMM device, a simulation infrastructure needs to be setup. There are several simulators for NAND flash storage systems and object storage device. However, none of them can be immediately applicable to the experiments of our proposed scheme. Flashsim [51] is one of the most popular NAND flash based solid disk drive simulator used in academia research. However, Falshsim is developed based on the block-level interface and therefore cannot be directly adopted to simulate the object-based device. Moreover, Flashsim only stores the accessed LBA without the requested data. Such an approach simplifies system design and speeds up simulation. However, it prevents adoption of Flashsim from evaluation of algorithms based data access patterns.

Several object-based device softwares are commercially available. A typical example is the open-source software OSC-OSD developed by Ohio Supercomputer Center [58]. It is implemented as the middleware on top of local file system, *e.g.*, ext4. Object storage in the underlying storage medium is handled by local file system, which is hidden from the user. Hence, the OSC-OSD software cannot be directly applied to simulation of the ONFD, either. Some works such as [19] uses NANDsim, a built-in linux kernel module for simulation. However, NANDsim does not support multiple-chip parallelism. In addition, modifying and debugging kernel modules are time-consuming. To provide a simulation infrastructure to evaluate and analyze the ONFD performance, we develop a simulator ObjNandSim which is introduced in Section 4.5.

4.4 OPTIMIZATION OF OBJECT-BASED NAND FLASH DEVICE

In this section, the data migration minimizing (DMM) device design is illustrated. First, an overview of our DMM device architecture is described in Section 4.4.1. Multi-level garbage collection (MLGC) which handles onode partial update are depicted in Section 4.4.2. After

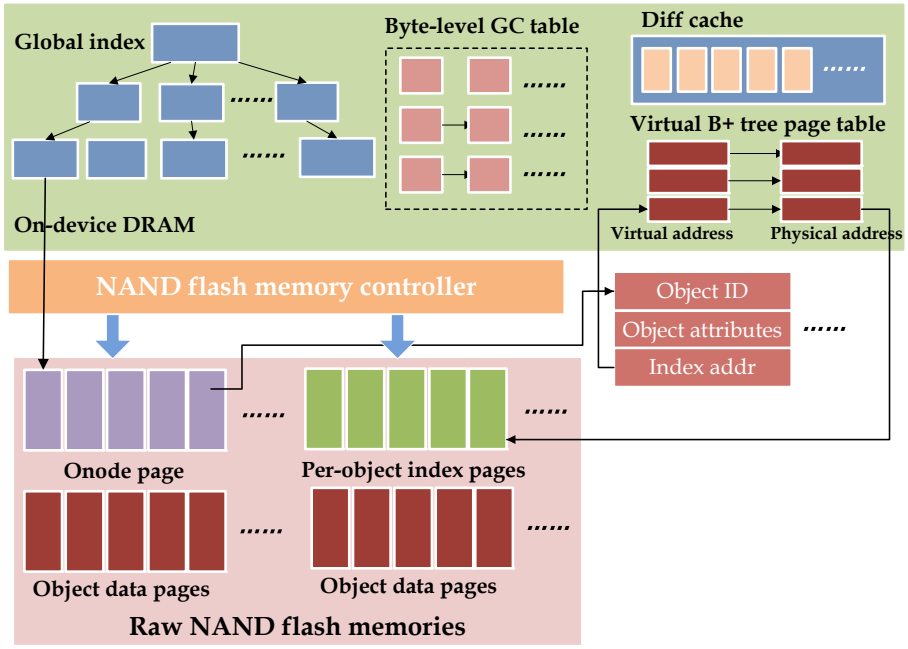


Figure 28: The overall architecture of the DMM device.

that, virtual B+ tree and diff cache are presented to handle cascading update in Section 4.4.3 and Section 4.4.4, respectively. Finally, a power failure handling approach is illustrated in Section 4.4.5.

4.4.1 An Overview of Data Migration Minimizing (DMM) Device

The overall architecture of the proposed DMM device is shown in Fig. 28. The DMM device is based on the ONFD architecture introduced in Section 4.1. Like [19], multiple onodes are compacted in one physical page to reduce internal fragmentation. The per-object index and object data are stored by unit of physical page. The object metadata and object data are stored in separate chunks to improve garbage collection efficiency. The DMM device optimizes the system architecture with following techniques:

Multi-level garbage collection. To the handle onode partial update, the multi-level garbage collection (MLGC) technique is proposed. Besides page-level garbage collection, MLGC adopts byte-level garbage collection. Instead of moving the valid bytes of the same page immediately, MLGC records the information of the invalidated bytes in a byte-level

GC table. By grouping invalid bytes incurred by more than one onode partial updates in a physical page and moving the remaining valid bytes at one time, the data migration is reduced. The detail of MLGC is depicted in Section 4.4.2.

Virtual B+ tree. To minimize data migration incurred by cascading update within per-object index, the virtual B+ tree is proposed. In the extent-based B+ tree, a major cause to internal node page update is that the frequently updated physical address of the child node pages. To reduce the address update frequency, the virtual B+ tree replaces physical addresses with virtual addresses. Every node page in the virtual B+ tree has a unchanged virtual address. The internal node pages record child node pages' virtual addresses. A page table is used to map the virtual addresses to the physical address. When a child node page migrates, the new address is updated in the page table instead of the parent node pages. Thereby, update to the internal node pages can be reduced. The virtual B+ tree is discussed in detail in Section 4.4.3.

Diff cache. To mitigate the object data induced cascading update, we propose the diff cache. Due to limited size, diff cache selectively buffers the data depending data type. To maximize cache utilization, a diff cache replacement policy is adopted. The diff cache technique is described in Section 4.4.4.

Power failure handling. In the DMM device, the byte-level GC table, the virtual B+ tree page table and the diff cache reside in volatile DRAM. Power failure will cause loss of the byte-level GC table and virtual B+ tree page table. Besides, the loss of diff cache data will result in inconsistency between object data and object metadata. To restore system consistency and the lost tables, we propose a power failure handling technique as described in Section 4.4.5.

4.4.2 Multi-level Garbage Collection (MLGC)

MLGC is proposed to mitigate onode partial update. Basically, MLGC delays migration of the un-updated bytes. By grouping invalid bytes incurred by two or more onode partial updates in a physical page and moving the remaining valid bytes at one time, the data migration can be reduced. To realize delay migration, we adopt both page-level and byte-

level garbage collection. Two tables are adopted: A page bitmap table is employed to record page statuses; a byte-level GC table is employed to keep track of byte statuses within a physical page. When onode partial update occurs, MLGC records the information of the invalid bytes <physical page number, offset, length> in the byte-level GC table. The physical page is still marked valid in the page bitmap table. During garbage collection, both the page bitmap table and the byte-level GC table are consulted. If a physical page is shown valid in the page bitmap table, MLGC checks whether this page has invalid bytes by the byte-level GC table. If there are invalid bytes, only the valid bytes migrate to a clean page. Otherwise, the entire page is moved.

The invalid bytes information is maintained in the byte-level GC table. Each invalid bytes information entry is represented by 8 bytes. Thus, if we book-keeps the invalid bytes information of every physical page, the memory consumption is prohibitively high. To reduce memory consumption, we set a pre-defined size for the byte-level GC table. When byte-level GC table is full, some physical page is selected for eviction. To minimize data migration, the physical page with most invalid bytes is selected. The valid bytes of the evicted page migrates to a clean page. Then the evicted physical page is invalidated in the page bitmap table. All invalid bytes information entries of the evicted page is removed from the byte-level GC table.

An example of MLGC is shown in Fig. 29. Upon onode partial update, the old onode stored at offset 150 in physical page 4 are invalidated. The invalid bytes information <physical page number 4, offset 150, length 50> are recorded in the byte-level GC table. At this point, the byte-level GC table is full. The physical page with most invalid bytes are selected for eviction. In the example, physical page 5 with 200 invalid bytes is selected. After moving the remaining valid bytes to physical page 127, the physical page 5 is invalidated in the page bitmap table. All invalid bytes information of physical page 5 is removed from the byte-level GC table.

The major overhead of MLGC is the byte-level GC table. The byte-level GC table is implemented with a hash table to quickly locate a physical page with invalid bytes. The invalid bytes information of the same physical page are stored in a single link table as

shown in Fig. 29. For fast access, the byte-level GC table is stored in DRAM. Hence, the performance overhead of MLGC is negligible. The size of the byte-level GC table is limited. The experiment results in Section 4.6 show that memory resources consumed by the byte-level GC table is only marginal.

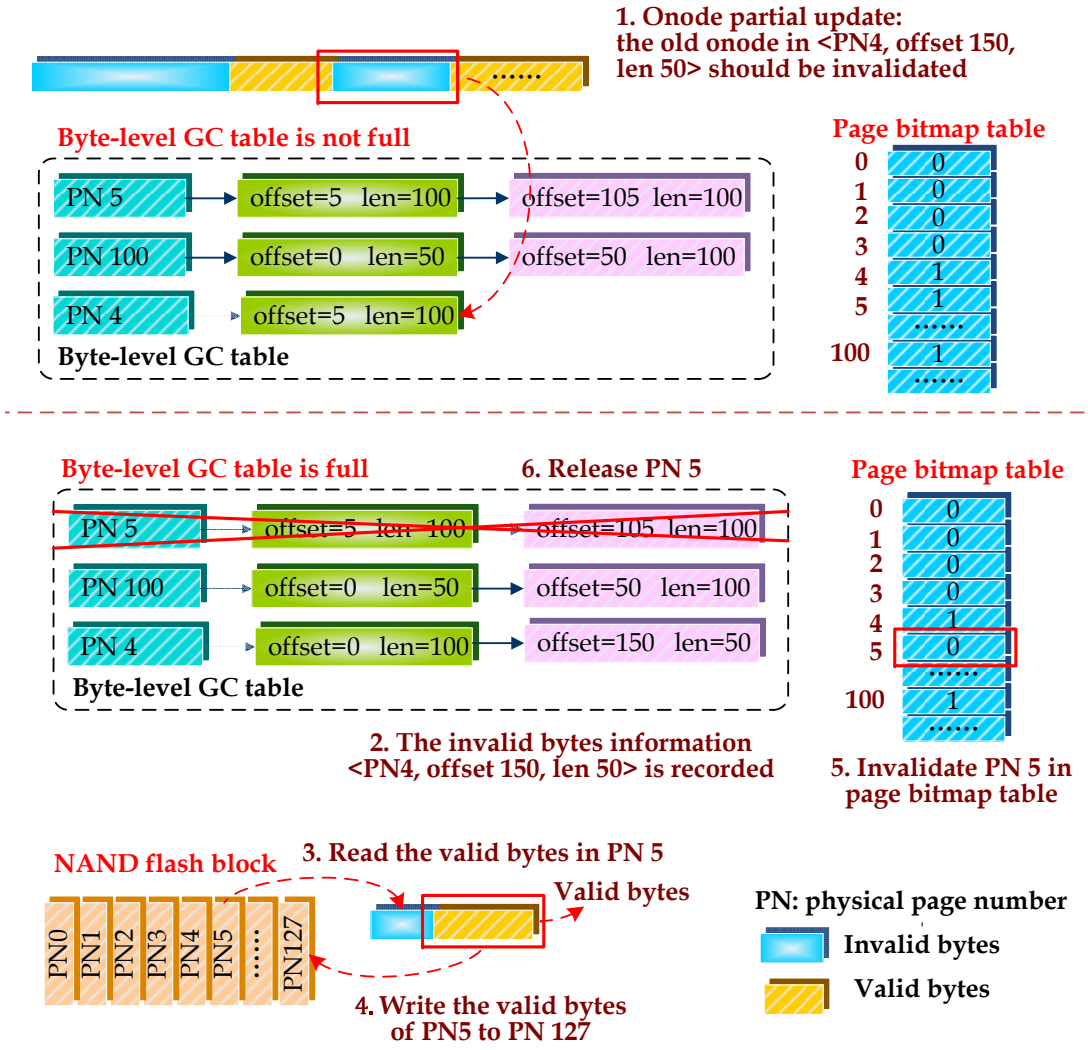


Figure 29: Multi-level garbage collection.

4.4.3 Virtual B+ Tree

Virtual B+ tree is proposed to mitigate the cascading update issue. Cascading update within per-object index is incurred by updates to the address of a child node page. If the addresses

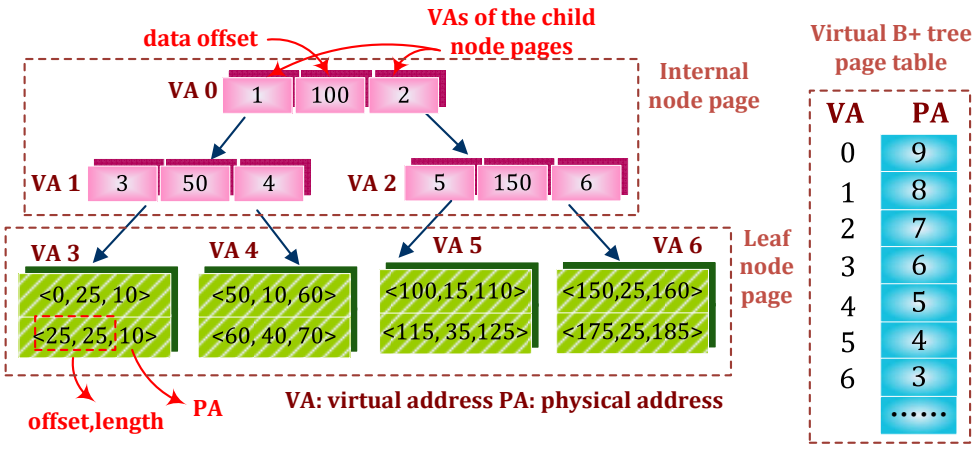


Figure 30: The virtual B+ tree.

of child node pages remain unchanged, update of parent node pages can be reduced. To reduce the update of child node page addresses, we propose the virtual B+ tree in this chapter.

4.4.3.1 Overview of Virtual B+ Tree The concept of the virtual B+ tree is shown in Fig. 30. The virtual B+ tree has the same structure as the extent-base B+ tree in [21]. The leaf node page records offset, length and physical address of the object data. The internal node page stores <data offset, child node page address> pairs, i.e., key-value pairs. Unlike the extent-based B+ tree, each node page of the virtual B+ tree has a virtual address (VA) besides the physical address (PA). The relationship between VAs and PAs is maintained in a page table. Instead of recording PAs, the internal node page stores the VAs of its child node pages. When a child node page migrates to a new page, the new PA is reflected in the page table instead of its parent node page. As such, updates to the internal node pages can be avoided. We also stores the VA of the per-object index root node page in the onode. Therefore, updates to onode can be also reduced.

The concept of VA is similar to the logic address in block-based NAND flash device. However, usage of the virtual address in the DMM device is different. In block-based NAND flash device, all stored data is indiscriminately assigned with logic addresses. In contrast, the DMM device only applies VAs to the per-object indices node pages. The object data

Table 13. Write Overhead of the Virtual B+ Tree

Operations	Extent-based B+ tree	Virtual B+ tree
Insertion without tree split	H	1
Insertion with tree split	2l+1+(H-l)	2l+1
Insertion with tree split(worst case)	2H+1	2H+1
Deletion without nodes merging	H	1
Deletion with nodes merging (worst case)	H-1	H-1

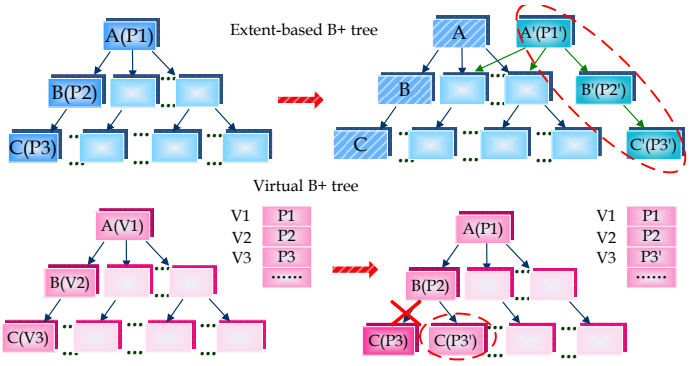
and onode have no virtual address. The per-object indices node pages only consume a small portion of the entire physical space. Hence, mapping between the VAs and PAs in the DMM device incurs low storage overhead.

4.4.3.2 Write overhead of virtual B+ tree The insertion and deletion write overheads of the virtual B+ tree are shown in Table 13. The overheads are evaluated with the number of page write. H and l denote the tree height and the number of splitting nodes, respectively. The overhead of virtual B+ tree insertion without tree splitting is always 1, 1/H of the extent-based B+ tree overhead. The example of Fig. 31(a) depicts the overhead difference. Node pages A, B, C are stored in PA P1, P2 and P3, respectively. P1, P2 and P3 are mapped to VA V1, V2 and V3. When the leaf node page C is updated, the virtual B+ tree only updates the new PA of C in the page table. Therefore, only one per-object index node page migrates. In contrast, up to three node pages, A, B and C are moved in the extent-based B+ tree.

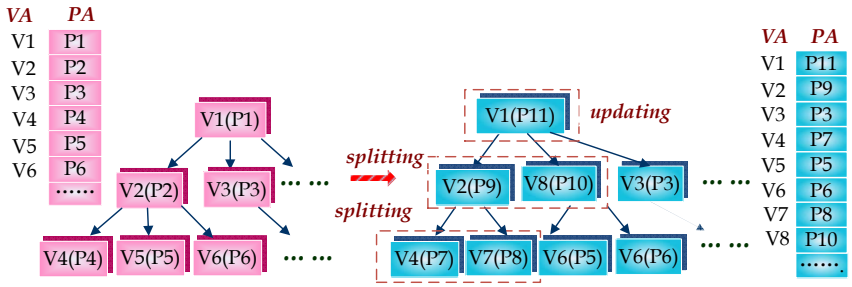
When insertion with tree splitting occurs, the splitting node, the newly inserted node and their parent node is updated. Compare with the extent-based B+ tree, the virtual B+ tree can reduce the page writes by $H - 1$. The worst case is that the tree splitting cascades to the root node page as shown in Fig. 31(b). In this example, tree splitting occurs to two node pages. Five page writes are incurred, which is equal to the overhead of the extent-based

B+ tree. Since the worst case occurs much less frequently than others, applying the virtual B+ tree can still significantly reduce the write overhead. The way to estimate the deletion overhead is similar to that of insertion.

Besides reducing the insertion and deletion overhead, the virtual B+ tree also reduces the overhead of garbage collection. In the extent-based B+ tree, migration of a child node page involves updating the parent node pages cascadingly. Comparatively, under the virtual B+ tree, migration of a child node page is only reflected in the page table. Therefore, update and migration of the parent node pages can be eliminated.



(a) Insertion operation without splitting in virtual B+ tree.



(b) Insertion operation with splitting in virtual B+ tree.

Figure 31: Insertion operation of virtual B+ tree.

4.4.3.3 Storage overhead of the virtual B+ tree Compared with the extent-based B+ tree, the virtual B+ tree consumes less storage space thanks to adoption of VA space. In the DMM device, only the per-object indices are assigned with VAs. The size of the VA

space is much smaller than that of the PA space. Hence, VA can be represented by fewer bytes than PA. For example, in 1TB ONFD with a 4KB page size, the key-value pair size of the extent-based B+ tree internal node consumes 7 bytes. In comparison, the key-value pair size of the virtual B+ tree only uses 6 bytes. The storage overhead is reduced by 14%. With smaller address size, an internal node page can store more key-value pairs. Therefore, the virtual B+ tree has a lower height and the PA of object data can be located faster. In addition, storing more more key-value pairs in one internal node page also lowers the re-balancing probability, further reducing the write overhead.

The virtual B+ tree introduces a page table to map the allocated VAs to PAs. The page table resides in DRAM for fast access. The size of the page table is variable due to dynamic VA allocation. A VA is dynamically allocated when a virtual B+ tree node page is inserted; the VA is de-allocated when the node page is deleted. In addition, the extent-based allocation may reduce the size of per-object indices and the allocated VA number. To achieve good space efficiency, we adopt a two-level page table to map allocated VAs to PAs. Although the exact size of the page table is not predicable, the maximum size of the page table is calculable: In 2TB ONFD with a 8KB page size, the maximum size of the page table is only 1MB. Compared with the size of DRAM (e.g., several gigabytes), the overhead of the page table is negligible.

4.4.4 Diff Cache

The virtual B+ tree can minimize update to the internal node pages. However, object data update still incurs write operations to leaf node pages. Besides, the tree rebalancing also incurs update to internal node pages and onodes. Caching per-object indices or/and object data is a way to reduce updates to per-object index node pages. However, due to limited on-device DRAM and object-based interface, the existing cache replacement policy [59][60] is sub-optimal. To maximize DRAM utilization, we propose a customized diff cache. The diff cache selectively buffers data depending on the data types. Object data is bypassed from the diff cache due to the following reason. In the existing object file system [61], a page cache is always used to buffer dirty object data. Committing dirty objects to the underlying ONFD is triggered by too much dirty object data or the object data being dirty for too

long. During commit, all the dirty data belonging to one object is flushed followed by the inode to keep system consistency. Hence, it is unnecessary to cache the object data in the limited-sized on-device DRAM.

The diff cache only buffers per-object indices and onodes. The per-object index leaf node page is updated by unit of a <offset, length, physical address> entry, i.e., leaf node entry. Each leaf node entry are only several bytes. To reduce memory consumption, instead of buffering the entire leaf node page, the diff cache only caches the updated leaf node entries. Only when the number of the updated entries in a leaf node page exceeds a threshold, the entire leaf node page is cached. For quick search, the related internal node pages are also cached in the diff cache. However, when tree rebalancing occurs, the internal node pages are not immediately updated. Keeping the internal node pages clean facilitates fast release of the diff cache space. Updating the internal node pages is delayed to the time of eviction. If the diff cache size exceeds a threshold, a diff cache replacement policy is adopted for eviction:

- Internal node page buffered in the diff cache are always clean. To accelerate release of cache space, the internal node pages of the the least recently used (LRU) objects are gradually released. The internal node pages at the higher level have lower revisiting probability. Hence, the internal node pages at highest levels are released first.
- If there is no internal node page, the updated object metadata of the LRU objects are evicted. The old per-object index leaf node pages are read out and merges with the updated entries. Once a leaf node page is updated, it is immediately committed to release DRAM space. To merge more updates, all the updated internal node pages reside in DRAM until the last updated leaf node pages are committed. The address of the per-object index root node page may be updated due to tree balancing. To reduce onode write incurred by per-object index root node page update, onode is committed after per-object index internal node pages.

An diff cache example is shown in Fig. 32. In the diff cache, the updated leaf node entries and clean internal node pages of object 2 and 3 are buffered. When updating per-object index of object 1, the diff cache is full and some cache space should be released. The clean internal node pages of objects 2 and 3 are first released. Then, the updated leaf node entries

of object 3 are evicted. During eviction, the updated leaf node pages of object 3 are first committed. After all updated leaf node pages are flushed, the updated internal node pages and onode are committed.

The diff cache consumes DRAM resources. During eviction, all the updated internal node pages are buffered in the memory at the same time, which may increase memory consumption. However, thanks to adoption of the virtual B+ tree, the frequency to update the internal node pages is significantly reduced. Hence, the probability to buffer many internal node pages simultaneously is low. The storage overhead of the diff cache is evaluated in Section 4.6.

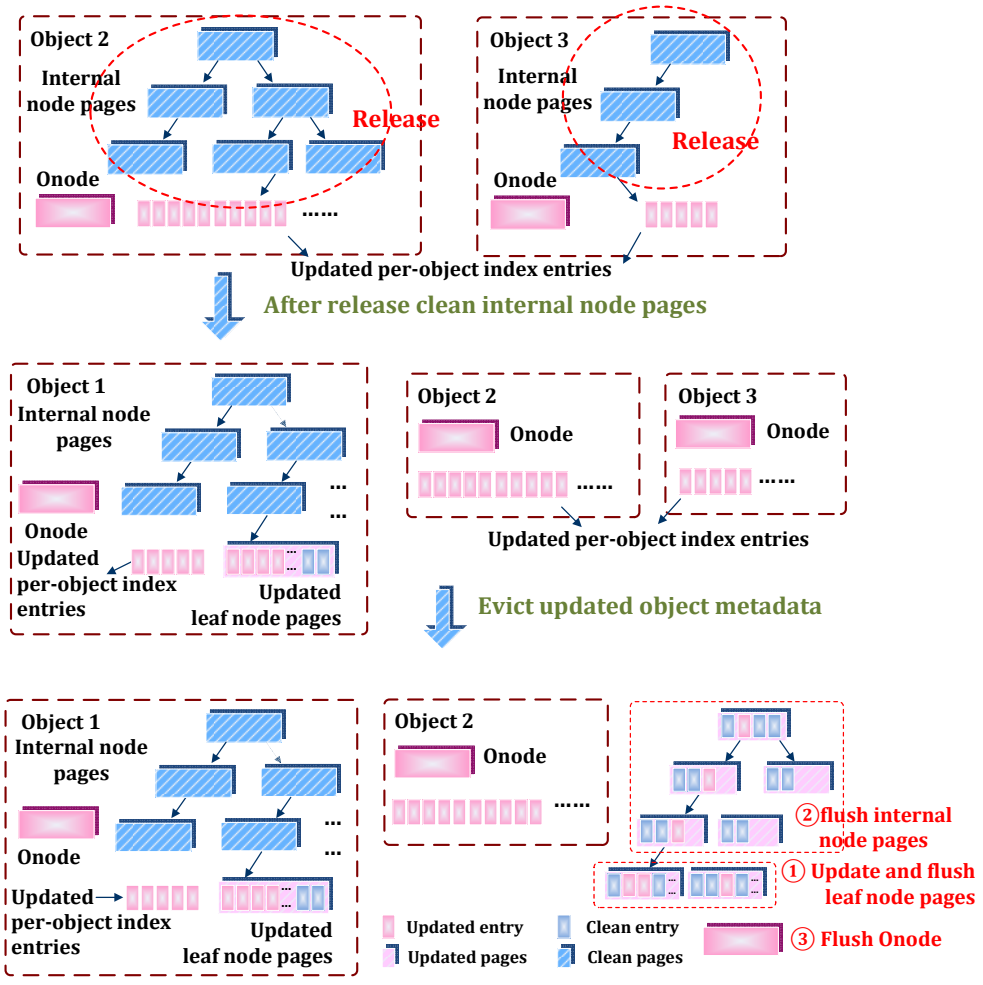


Figure 32: An example of the diff cache.

4.4.5 Power Failure Handling Approach

The byte-level GC table, the virtual B+ page table and the diff cache are buffered in on-device DRAM. Power failure will result in loss of byte-level GC table and virtual B+ tree page table. Besides, loss of the diff cache data upon power failure also causes inconsistency between object data and metadata. To restore system consistency and rebuild the lost byte-level GC table and virtual B+ tree page table, a data recovery technique is proposed.

4.4.5.1 Overview of DMM data recovery The data recovery technique leverages page metadata to restore system consistency. The page metadata includes an object ID, a sequence number (SN) and a VA. The SN represents the sequence of write operations. The SN is automatically incremented after a write operation. The byte-level GC table can be restored with SNs and object IDs. As described in 4.4.1, only onode partial update is recorded in the byte-level GC table. During data recovery, the metadata of the physical pages storing onodes are read out. If more than one physical page stores an onode with the same object ID, only the onode with the highest SN is valid. Others are invalidated in the byte-level GC table.

The system consistency can be restored with SNs and VAs. The system consistency is restored in two steps: 1) The per-object index of the inconsistent objects are rolled back to their own latest consistent state (LCS); 2) then the physical address of the object data written after the LCS is used to update the per-object index. In LCS, the corresponding per-object index and onode of the updated object data are completely committed. To roll back the per-object indices to the LCS, it is critical to reconstruct a consistent virtual B+ tree page table. To rebuild the virtual B+ tree page table, we handle the consistent and inconsistent objects in different ways. For the consistent object, the PAs of the per-object index node pages are inserted to the virtual B+ tree page table according to the VAs stored in these PAs. If more than one PA has the same VA, only the PA with the highest SN is inserted. For the inconsistent object, only the per-object index node pages before the LCS can be valid. Hence, the per-object index node pages with the SNs higher than the LCS are

all invalidated. The PA of a per-object index node page with SN lower than the LCS can be inserted to the virtual B+ tree page table. Again, if more than one PA has the same VA, only the PA with the highest SN is inserted.

The inconsistent objects are only the objects buffered in the diff cache. The LCS of the inconsistent object can be represented by the SN of its latest committed onode. To quickly identify the inconsistent objects, the object IDs and SNs of latest committed onodes of the objects in diff cache are flushed to NAND flash memory with back-up power upon power failure. The updated onode in the diff cache are also flushed to NAND flash memory to prevent data loss. Due to small size, flushing these data won't invoke high energy consumption. Thus, highly reliable back-up power tantalum capacitors can be to adopted to guarantee system reliability. After rolled back to the LCS, the object data of these inconsistent objects is read to update the corresponding per-object indices: If the inconsistent object have an object data whose SN is higher than the LCS, the PAs of these object data are used to update the per-object index. After per-object index update, the object metadata becomes consistent with the object data.

An example of consistency restoration is shown in Fig. 33. The power failure occurs when committing updated per-object index pages of object 1. Updated per-object index pages are not completely committed: VA1, VA2, VA5 and VA6 are committed while VA0 is still in diff cache. Hence, object 1 has inconsistent metadata and data. During restoration of the virtual B+ tree page table, the per-object index of object 1 is rolled back to the LCS which is SN5: VA1 and VA2 are mapped to PA2 and PA0 whose SNs are lower than SN5; The V5 and V6 have SNs higher than SN5 and therefore are invalidated. After restoration of the page table, the inconsistent object data with SN6 is found. The object data PA PA6 are used to update per-object index. Finally, the updated per-object index pages VA0, VA1, VA2, VA5 and VA6 are all flushed to PA15, PA13, PA11, PA12 and PA14.

Onode page, per-object index node page and data page need different page metadata for system restoration. Hence, we adopt different metadata formats for different types of pages as shown in Fig. 34 The data page metadata includes object data offset and object data length to update per-object index. The onode page metadata contains onode number and

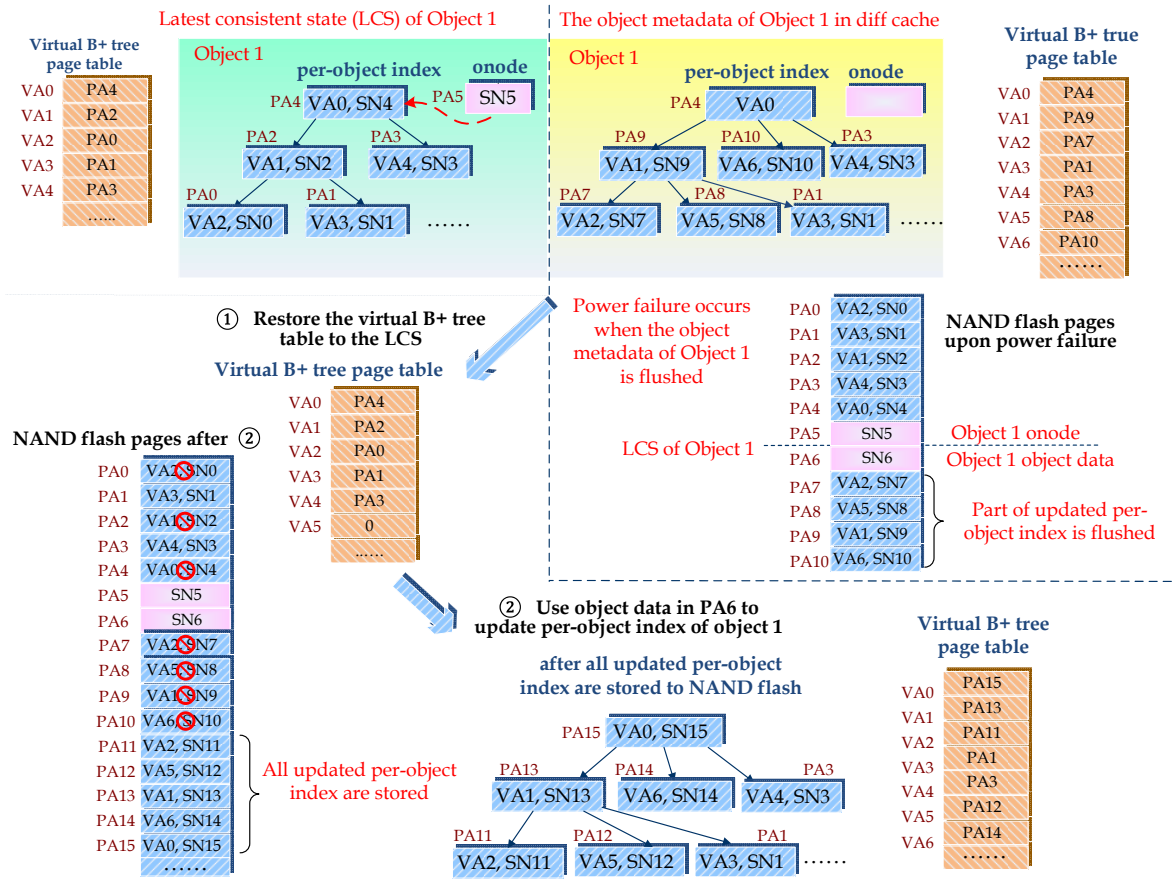


Figure 33: An example of per-object index recovery.

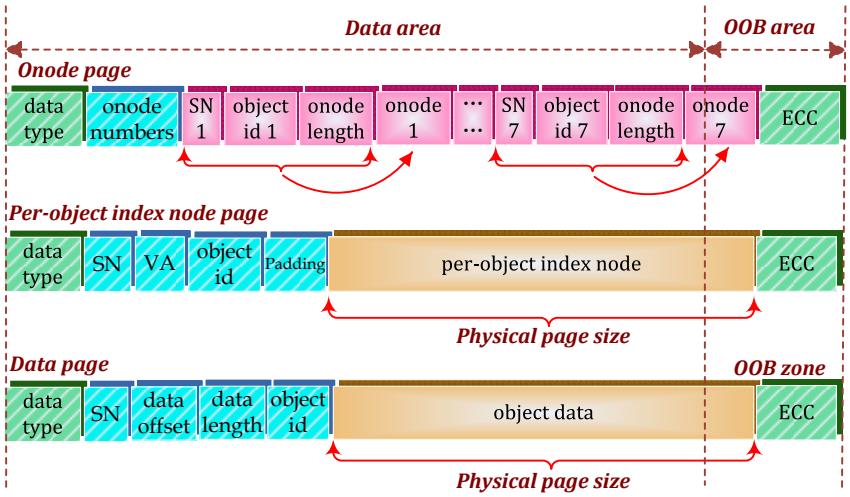


Figure 34: The format of the page metadata.

Table 14. The definition of the variables of the restoration procedure

Name	Description	Name	Description
GLB_IDX	global index	VB_TB	the virtual B+ tree page table
BL_GCT	byte-level GC table	IC_OBJ	list of inconsistent objects
CPN/CPA	current page number/address	TPN	maximum page number

onode length. The metadata of per-object index page contains VA. Unlike storage formats in [10], the page metadata and page data can both traverse the data area and OOB area. This is because the metadata of onode pages is too large to be all stored in OOB area.

4.4.5.2 Data recovery implementation The restoration procedure is depicted in Algorithm 1. We construct the global index simultaneously to facilitate the recovery of these two tables. The variables in the algorithm are listed in Table 14.

4.5 OBJNANDSIM: ONFD SIMULATOR

To evaluate the efficiency of the proposed DMM device, we set up a object-based simulation platform. A user-level object-based NAND flash device simulator ObjNandSim is implemented. In this section, we first introduce the overall architecture of the simulation platform in Section 4.5.1. Then, an overview of the ObjNandSim is introduced in Section 4.5.2. Finally, the hardware and software components of ObjNandSim are discussed in Section 4.5.3 and Section 4.5.4, respectively.

4.5.1 Simulation Platform

The overall architecture of the simulation platform is shown in Figure 35. The simulation platform incorporates an object file system, an object storage device (OSD) initiator and

Algorithm 1 The restoration procedure

```
    procedure RESTORE_SCAN(GLB_IDX,VB_TB,BL_GCT,IC_OBJ, TPN)
 2:     Initialize: CPN = 0                          ▷ CPN: current page number
        VB_TB.INIT(0);
 4:     while CPN ≤ TPN do
            if IS_FREE(CPN) then
 6:             goto INC_CPN
            end if
 8:         if PAGE_TYPE(CPN) == OBJ_INDEX && SN(CPN) >
    VB_TB.GET_SN(VA(CPN)) && (!IC_OBJ.HAVE(OBJ_ID(CPN) ||
10: IC_OBJ.GET_LAST_COMMIT_SN(OBJ_ID(CPN) > SN(CPN)) then
                VB_TB.UPDATE(VA(CPN),CPN, SN(CPN))
12:         end if
            if PAGE_TYPE(CPN) == OBJ_ONODE then
14:             if !GLB_IDX.HAVE(OBJ_ID(CPN)) then
                    GLB_IDX.UPDATE(OBJ_ID(CPN),CPA,SN(CPN))
16:             else if GLB_IDX.HAVE(OBJ_ID(CPN)) && SN(CPN) >
    GLB_IDX.GET_SN(OBJ_ID(CPN)) then
18:                 BL_GCT.INSERT(GLB_IDX.GET_PA(OBJ_ID(CPN)))
                    GLB_IDX.UPDATE(OBJ_ID(CPN),CPA,SN(CPN))
20:             end if
            end if
22:         INC_PLN:
            CPN=CPN+1
24:     end while
    end procedure
```

an OSD. The object file system only maintains a name space without object management functionality. EXOFS in Linux kernel is adopted as the object file system [61]. EXOFS does not support direct IO. It can be only mounted on ASYNC mode. Under this mode, object data and metadata are buffered in the page cache before flushed to the storage device. Upon receiving a data request, EXOFS will first find the data in the page cache. If the data does not reside in the page cache, a request will be sent to the OSD driver or OSD initiator. The OSD initiator is an Linux kernel library which generates and interprets OSD request packets according to the SCSI OSD command sets [54]. The application can also bypass the object file system and talk to the OSD initiator directly. Finally, OSD request packets are sent to the OSD via open-iSCSI, an iSCSI initiator. In this simulation platform, OSD is ObjNandSim, the object-based NAND flash device simulator.

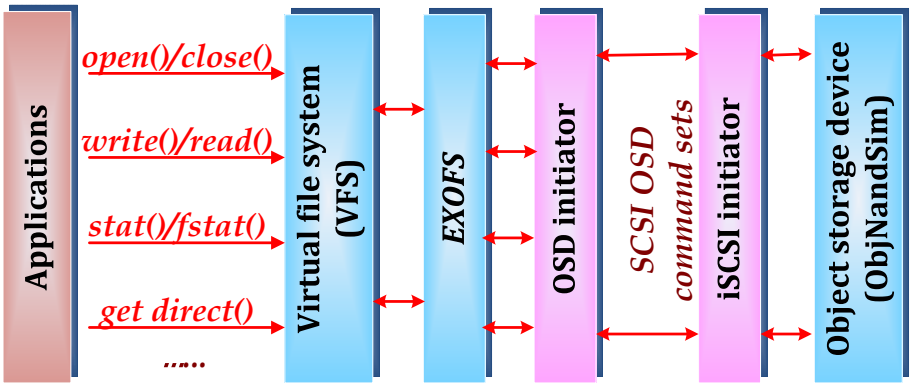


Figure 35: The architecture of simulation platform.

4.5.2 Overall Architecture of ObjNandSim

ObjNandSim is developed under the framework of open-source software OSC-OSD [58]. To facilitate further code modification and extension, ObjNandSim provides a simple API for object data and metadata (e.g., inode) access. The API functions are listed as follows:

- *nand_write_obj*(), *nand_read_obj*(): write and read object data;
- *nand_delete_obj*(), *nand_truncate_obj*(): delete or truncate object;
- *nand_write_inode*(), *nand_read_inode*(): write and read inode.

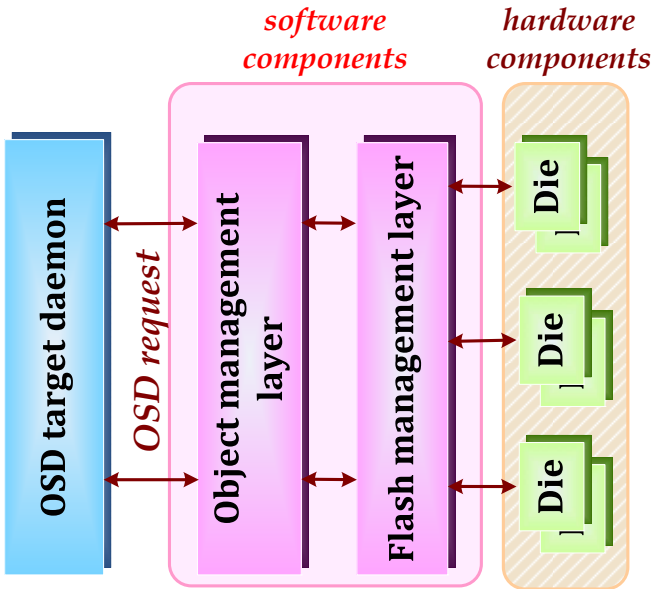


Figure 36: The architecture of ObjNandSim.

The ObjNandSim implements the ONFD architecture described in Section 4.1. The architecture of the ObjNandSim is illustrated in Figure 36. The ObjNandSim incorporates both hardware and software components. The hardware component mimics the hardware behavior of NAND flash memory array such as generating NAND flash memory latency and emulating multi-channel parallelism. The software component performs object management and NAND flash management as in [56].

- **Object management.** The software component controls object access through SCSI OSD commands such as READ, WRITE, DELETE, TRUNCATE and etc. It also manages the layout of object data and object metadata.

- **NAND flash management.** The software component implements physical page allocation and garbage collection as conventional FTL. Besides, the object-based interface allows variable-sized write operations. Hence, the size of write data may be smaller than the size of a physical page, which is partial page write. The software component also handles the partial page write.

4.5.3 Hardware Component

The hardware component of ObjNandSim emulates the NAND flash storage space, generates flash memory access latency and mimics multi-channel parallelism. The ObjNandSim allocates a part of the virtual memory space (main memory resource) as the storage space of NAND flash memories. The emulated storage space can be divided into the following parts:

- **Page**: A page is the smallest write unit. Each page contains of data area and OOB area.
- **Block**: A block is the smallest erase unit. Each block contains a number of pages. The typical page number in a block is 32, 64 or 128.
- **Die**: A die is a single NAND flash chip. Each die contains a number of blocks.
- **Channel**: The entire NAND flash memory storage space consists of a number of channels. Each channel contains one or more dies. The dies in different channels can be accessed simultaneously.

In the ObjNandSim, the sizes of channel, die, block and page are configurable. The program, erase and read latencies of NAND flash memory is emulated by using usleep(). Multi-channel parallelism is also implemented in the ObjNandSim. The multi-channel access parallelism is realized by book-keeping the access latency of each channel. The overall I/O latency is the maximum value of all channel latencies.

4.5.4 Software Component

Major functionality of the ObjNandSim is implemented in the software component. In this section, we will discuss the software component functionality and the related work flow.

4.5.4.1 Software component function The ObjNandSim software component includes two modules: object management layer and flash management layer. The object management layer controls object operations. Only the object operations used in EXOFS are implemented in the ObjNandSim. The implemented object operations are shown in Table 15. To realize the object management, three data types, i.e., onode, per-object index and object data are adopted as [19]. The layout of these three types of data in the ObjNandSim adopts data

Table 15. The object operations implemented NandOsdSim

SCSI OSD commands	Function call
CREATE	nand_write_inode()
REMOVE	nand_delete_obj()
WRITE	nand_write_obj()
READ	nand_read_obj()
GET_ATTRIBUTE	nand_read_inode()
SET_ATTRIBUTE	nand_write_inode(), nand_truncate_obj()

and metadata separation as [19]: To improve the efficiency of garbage collection, the object data and the object metadata (i.e., onodes and per-object indices) are stored in different chunks. The flash management layer implements physical page allocation as well as garbage collection and tackles partial page write. The log-structured physical page allocation [62] is employed in the ObjNandSim. The basic garbage collection unit is a chunk. A chunk consists of a number of physical blocks. The chunk size is configurable in the ObjNandSim. Garbage collection is performed in a greedy manner: The chunk with the fewest valid pages is selected for reclamation. The partial page write is also handled in the ObjNandSim. Two partial page write methods are adopted. 1) If the partial page write is performed to an existing data, read-before-write operation is performed as in [63]. The un-updated data is read out and stored in a clean page with the updated data. 2) If the partial page write is performed to new data, the new data is first stored in a page-size data buffer. When it is full, the data buffer is flushed to NAND flash memory.

4.5.4.2 I/O operation flow of the ObjNandSim

Based on the functionality of software component introduced in Section 4.5.4.1, we illustrate the I/O operation flow of the ObjNandSim. One I/O operation can be broken down into a number of object operations.

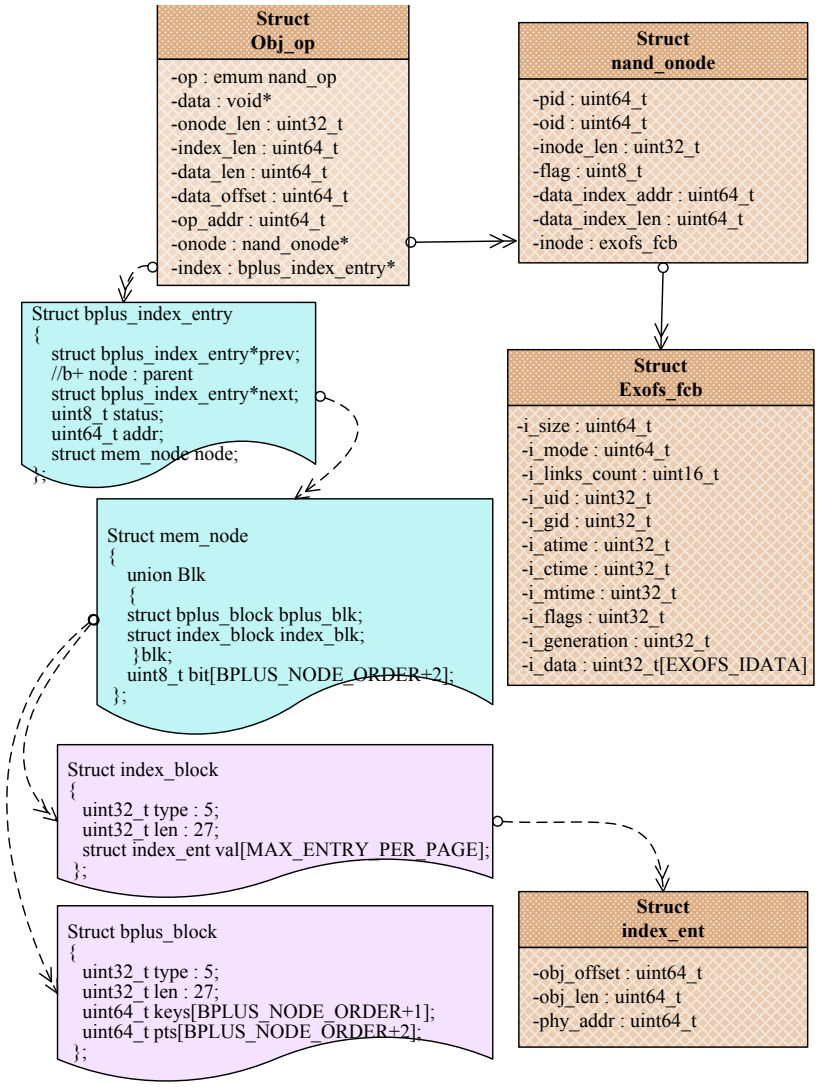


Figure 37: The data type and dependency in the ObjNandSim.

The data type and dependency related to the object operation are shown in Fig. 37. An object operation employs a data structure (*struct obj_op*) to maintain the information related to the operation and object. Each object operation involves accesses to onodes (*struct nand_onode*), per-object indices (*struct bplus_index_block*) and data. The onode records object inode (*struct exofs_fcb*) and the address of per-object index root node page. There are two types of per-object index node pages: the internal node page (*struct bplus_block*) and the leaf node page (*struct index_block*). The internal node page records <data offset, length, PA> pairs of the child node pages. The leaf node page records <offset, length, PA> of the

object data (*struct index_ent*). During a write I/O, garbage collection is usually invoked to free storage space for the upcoming write data. Hence, object operations to two or more objects are handled during one I/O. To maintain the object operations to object data, per-object indices and onodes, three queues, i.e., data queue, index queue and onode queue are employed. The data queue, index queue and onode queue are processed sequentially.

The write I/O operation flow is shown in Fig. 38. When a data write I/O request is received from the file system, the corresponding onode is first retrieved from the global index. According to the address stored in the onode, the per-object index root node page is read out. If the write I/O request updates an existing data, the old object data is invalidated first. The new object data is encapsulated into an object operation structure. The object operation is then inserted to the data queue and waits for process. After the object data is flushed to NAND flash memory, the per-object index entries are updated accordingly and inserted into the index queue. Similarly, when the per-object index is flushed, the new per-object index root node page is updated in the onode and the onode is inserted to the onode queue. When garbage collection occurs, the migrating data is inserted to the corresponding queue depending on the data type. The details of write I/O operation flow are shown in Algorithm 2.

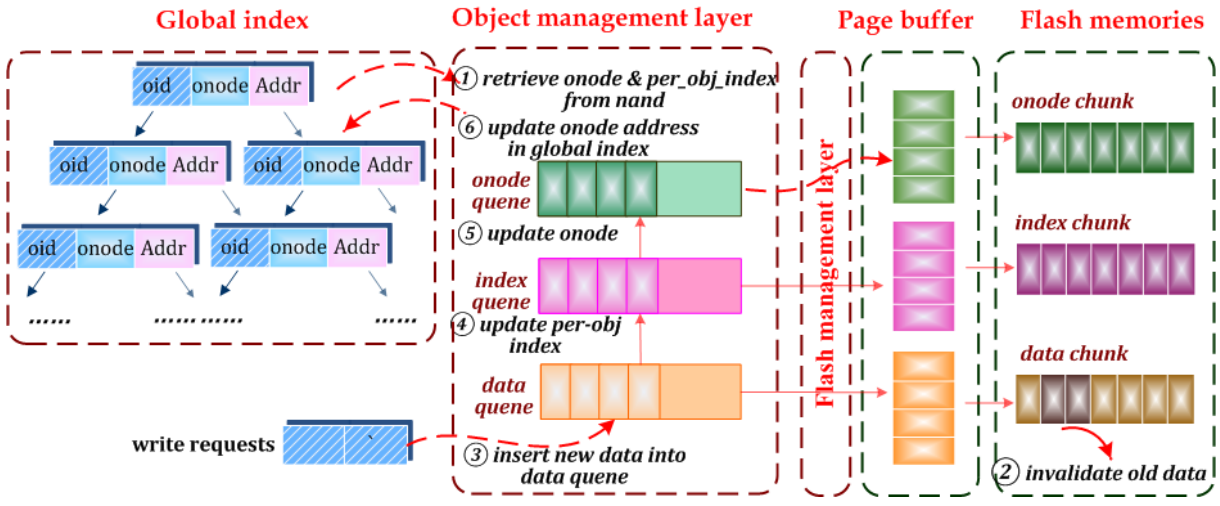


Figure 38: The ObjNandSim write I/O flows.

The object data read operation is relatively simple. Upon a read request, the onode is retrieved from the global index followed by the per-object index. Then object data can be located by looking up the per-object index.

Algorithm 2 ObjNandSim write I/O operation flow

```
1: procedure NAND_WRITE_OBJ(oid, offset, len, new_data)
2:     onode = global_index.read_onode(oid)
3:     index = onode.read_index()
4:     index.invalidate_data<offset,len>
5:     new_op(oid,offset,len,new_data)
6:     data_queue.insert(new_op)
7:     handle_data_queue:
8:     while !data_queue.empty() do
9:         cur_op = data_queue.dequeue()
10:        if available_chunks < T_chunk then
11:            recycle_dirty_block()
12:        end if
13:        addr = flush2nand(cur_op.get_data())
14:        new_op = index_queue.find(cur_op.get_oid())
15:        if !new_op then
16:            onode = global_index.read_onode(cur_op.get_oid())
17:            index = onode.read_index()
18:            new_op(cur_op.get_oid(),index)
19:            index_queue.insert(new_op)
20:            index.invalidate_index(cur_op.get_offset(), cur_op.get_len())
21:        end if
22:        new_op.update_index(cur_op.get_offset(), cur_op.get_len()),addr)
23:    end while
24:    while index_queue.empty() do
25:        cur_op = index_queue.dequeue()
```

```
26:         if available_chunks < T_chunk then
27:             recycle_dirty_block()
28:             goto handle_data_queue
29:         end if
30:         addr = flush2nand(cur_op.get_index())
31:         new_op = onode_queue.find(cur_op.get_oid())
32:         if !new_op then
33:             onode = global_index.read_onode(cur_op. get_oid())
34:             new_op(cur_op.get_oid(),onode)
35:             onode_queue.insert(cur_op.get_oid(),new_op)
36:             onode.invalidate()
37:         end if
38:         new_op.update_onode(addr)
39:     end while
40:     while onode_queue.empty() do
41:         cur_op = index_queue.dequeue()
42:         if available_chunks < T_chunk then
43:             recycle_dirty_block()
44:             goto handle_data_queue
45:         end if
46:         addr = flush2nand(cur_op.get_onode())
47:         global_index.update_onode_addr(cur_op.get_oid(), addr)
48:     end while
49: end procedure
```

4.6 EXPERIMENTAL RESULTS

In this section, we first measure I/O response time, page write counts and block erase counts to demonstrate validation of the ObjNandSim. The simulation results of the the ObjNandSim are shown in Section 4.6.3. Then the efficiency of the DMM device is evaluated by using the ObjNandSim. To fully demonstrate the efficiency of the DMM device, we separately show the simulation results of the proposed MLGC, virtual B+ tree and diff cache of the DMM device in Section 4.6.3.1∼ 4.6.3.3. Then we compare our works with previous works in Section 4.6.3.4.

4.6.1 Simulation Setup

In our experiments, we adopt the simulation platform shown in Fig. 35. The object file system, OSD initiator, and the ObjNandSim run the same host machine. The host machine is equipped with an Intel Quad-Core Xeon 5-2609 v2 (10MB 2GHz) processor and 128GB RAM. The operating system is RHEL 5 with Linux kernel 3.2.67. The OSD initiator and the ObjNandSim are connected via the Gigabit Ethernet. To exclude the network overhead for accurate evaluation of device overhead, we only show the I/O response time of the ObjNandSim instead of the entire simulation platform. The configuration of the ObjNandSim is adjustable. By default, The ObjNandSim is set with following configuration: the device capacity is set 64 GB with 16 NAND flash memory dies; the channel number and the chunk size are 16 and 2MB, respectively. The default NAND flash memory parameters are shown in Table 16. We set low clean chunk thresholds to quickly initiate garbage collection.

In the simulation, a customized benchmark, DMMbench are adopted to demonstrate the validation of the ObjNandSim. The DMMbench creates one large file and accesses the file data by using four types of I/Os: sequential write, sequential read, random write and random read. To accurately verify the functionality of the ObjNandSim, DMMbench communicates with the ObjNandSim only through OSD initiator without EXOFS to bypass the page cache. DMMbench emulates the behavior of synchronized I/Os from file system: Before the file data access, the file inode is read; each file data access is followed by a file inode write. The length

Table 16. The default parameters of NAND flash memory

Capacity	Block size	Block number	Page size
	512KB	8192	2KB
Timing	Program latency	Read latency	Erase latency
	900 μs	50 μs	1500 μs

of sequential I/O ranges from 4KB to 1MB. The length of random I/O is 4KB. The total accessed data amounts under different I/O types are the same. The device performance under DMMbench is predictable. The validity of the ObjNandSim can be demonstrated by comparing the measured performance and expected values.

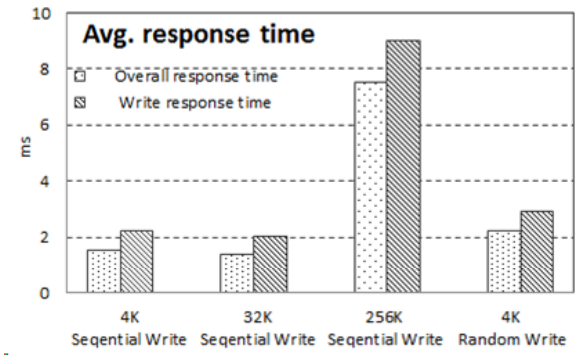
We evaluate the performance of the DMM device under four real-world workloads. The characteristics of these workload traces are depicted in Table 17. The TPCC workload trace is generated by Hammerora [64] and captured with strace utility [65]. We write a relay program to generate input to the simulation platform from these workload traces as shown in Fig. 26.

Table 17. Workload characteristics

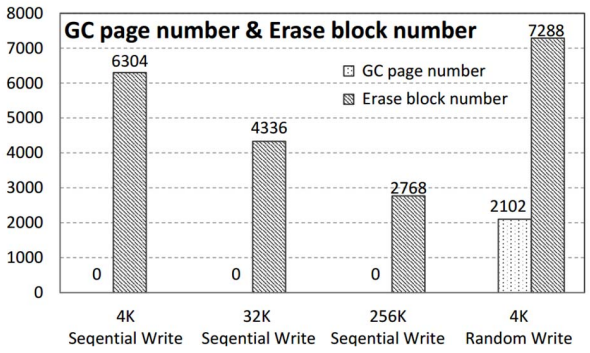
Workload	Application	File count	Avg. req. size (KB)	Write (%)	Write seq. (%)	Read seq. (%)
DVORAK (CODA) [66]	campus server environment	10262	1.23	4.1	86	39
iMOVIE (iBench) [67]	Mac application	85806	19.09	6.07	0	11.33
iPHOTO (iBench) [67]	Mac application	6902	6.06	83.10	50.83	37.54
TPCC	online transaction processing	677	4.59	22.64	19.71	14.54

4.6.2 ObjNandSim Evaluation

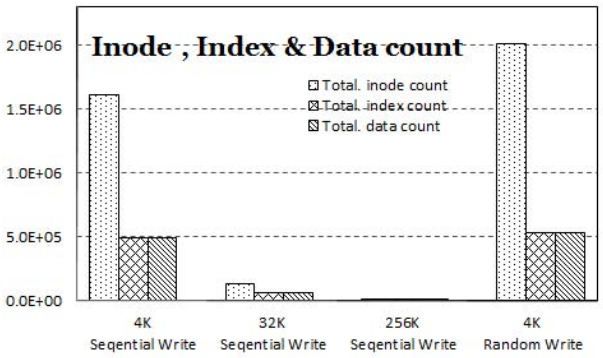
First, we demonstrate the memory latency emulation and the software component functionality of the ObjNandSim. The average response time of the ObjNandSim under sequential and random write I/Os is shown in Fig. 39(a). The average write response time under 4KB I/O is 2.24ms, which is between the latency of two page write operations and three page write operations. Write to one data page involves write to data, per-object index and onode. Since different types of data are distributed to different pages, total three write operations are performed. Although each data write I/O causes three write operations, the response



(a) The average and total response time under 4KB, 32KB, 256KB sequential and random write I/Os.



(b) The page write counts and the erase counts under sequential and random write I/Os.



(c) The write counts of onode, per-object-index and object data under sequential and random write I/Os.

Figure 39: The sequential and random average response time under DMMbench.

time is less than the latency of three page write operations. This is because several write operations to onodes or per-object indices incur one page write operation due to small object metadata size. The average response time is only 1.500ms, less than the average write response time. This is because there is inode read performed before the object data is accessed. The latency of inode read operation is much lower than the that of data write operation.

The average write response time under 32KB is 2.04ms, which is less than that of 4KB sequential write. The reduced response time can be explained by multi-channel parallelism and object metadata write reduction. Due to utilization of the multi-channel parallelism, write data is evenly distributed to all channels of the NAND flash array. Hence, the response time of object data write does not increase compared with that of 4KB write. However, I/O operation of larger size incurs fewer number of write operations to onode and per-object index as shown in Fig. 39(c). Each write request to object data is followed by the write to the corresponding inode. Hence, under the same total amount of object data write, large data I/O size incurs fewer inode write therefore fewer onode write operations. In addition, fewer object data write operation also incur fewer updates to the per-object indices. Reduction of the onode and per-object index write operations reduces the total page write amount. Hence, fewer erase operations are invoked to reclaim the dirty chunks, which can be demonstrated by Fig. 39(b). Reduction of the erase operations also reduces the response time.

The average response time under 256KB sequential I/Os is 7ms, which is much higher than that of 32KB sequential write. The response time increase can be explained by large data write per I/O operation and channel saturation. The write latency of object data write under 256MB sequential I/O is close to 8 times that of 32KB sequential write. 256KB I/O size can reduce write operations to onode and per-object index and incur fewer erase operations as shown in Fig. 39(b). However, the response time increase resulting from a large amount of data write still outnumbers the response time reduction. The average response time under 4KB random I/Os is 20% higher than that of 4KB sequential write. The performance degradation is induced by data migration during garbage collection.

The page write and erase counts of the ObjNandSim under sequential and random write I/O are shown in Fig. 39(b). As expected, there is no page write operation during garbage collection under sequential I/Os. The erase count under 256KB sequential I/Os is only

approximately 30% of 32KB sequential write and only 67% of 4KB sequential write. The page write and erase count reduction is due to fewer write operations to onode and per-object index. For random write, approximately two thousand page write operations are performed during garbage collection. Unlike sequential write which invalidates all pages in a chunk, random write induces partial update to a chunk. The valid pages in the dirty chunk have to migrate before the chunk is erased, which increases both the garbage collection overhead and the response time as shown in Fig. 39(a).

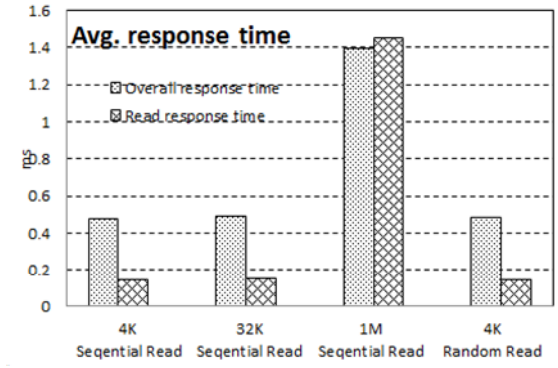
The statistics of writes to object data and metadata in Fig. 39(c) is employed to explain the page write and erase counts. The onode write count under 256KB sequential I/Os is only approximately 12% of 32KB sequential write and 1.5% of 4KB sequential write. The per-object index write count under 256KB sequential I/Os is also reduced compared with the 32KB and 4KB sequential writes. Similar to onode write, the per-object index write is reduced with the increase of I/O sizes. Under the 4KB random write, write operations to object data, per-object index and onode increase. Compared with the 4KB sequential write, write operations to data, per-object index and onode increase by 8%, 8% and 20%, respectively. The increase of data write count is within expectation: since the valid and invalid pages co-exist in the chunks storing object data, data migration occurs during reclaiming these chunks. Although there are all invalid pages in the dirty chunks storing per-object index and onode, there is still increase in onode and per-object index write counts. This is because increased object data write operations also incur more updates to per-object index and onode.

The average response time of sequential and random reads are shown in Fig.40(a). Under 4KB sequential write, the average read response time is approximately three times NAND flash read latency. This is because the object data can be accessed after its onode and per-object index are read out. Hence, reading one object data page requires three page read operations. Similar to sequential write, the average response time under 32KB sequential read is the same as that of 4KB sequential read by leveraging multi-channel parallelism. The average response time under 1MB sequential read I/O is 32 times that of 32KB and that of 4KB. The average response time is higher than the average read response time. This is because that each read request is followed by an inode write to update atime. Therefore, the

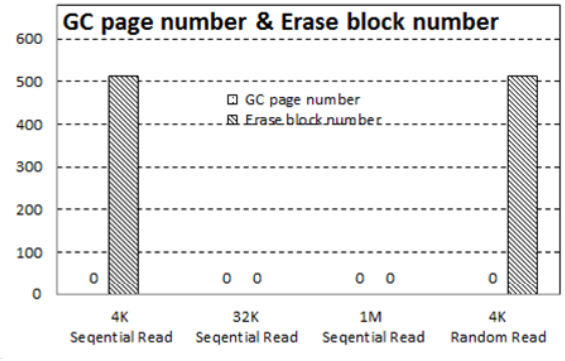
latency to handle inode write is also counted in the average response time calculation. Under 4KB and 32KB sequential read, the average read response time is lower than the average response time. When the I/O size increases to 1MB, the average read response time is higher than the average response time. This is because the read latency to read 1MB object data reaches up to 1500 μs, which is much higher than that of the average onode write.

Since the sequential and random read tests also incur write operations to inode, we also shows the write and erase statistics in Fig. 40(b) and Fig. 40(c). There are only erase operations during garbage collection. It can be explain by inode write. Write operations to inode only incurs write operations to onode as shown in Fig. 40(c). Repeated updating one onode eliminates valid pages in the dirty blocks. As expected, the onode write count decreases with increase of the I/O size and consequentially the erase count also decreases: onode write count of the 4KB sequential write is eight times that of 32KB sequential write. There is no onode write count difference between sequential and random 4KB write.

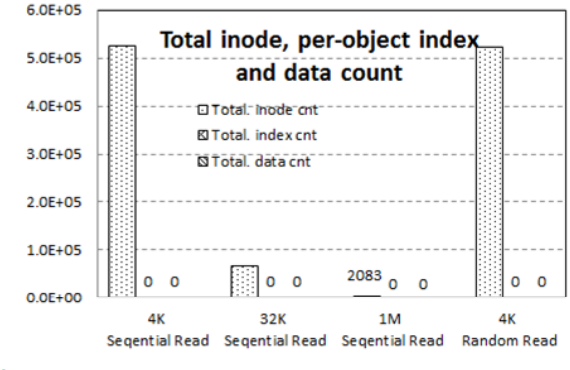
We also demonstrate the internal parallelism capability of ObjNandSim by comparing the response time under the configurations of different channels. We evaluate the performance under 4, 8 and 16 channels under 32KB sequential I/Os. Here, we assign one NAND flash die to one channel. To keep the block size unchanged, we set the page number of a physical block 1024,512,256 under 4, 8 and 16 channels, respectively. The average write response time of the sequential write and the average read response time of sequential read are shown in Fig. 40(c). As expected, the average write response time decreases when the channel number increases: when the channel number increases from 4 to 8 and from 8 to 16, the average response time increases approximately 0.7 ms, which is close to the write latency of one page write. The increase of the write response time is less than the latency of one page write. This is because the inode update induced write latency is also counted in the average response time calculation. Several inode updates incurs one physical page write. The sequential read I/O test demonstrates similar results: the average read response time under 16 channels is approximately 50 μs lower than that of 8 channels. As expected, the read response time reduction is close to the latency of one page read.



(a) The response time under 4KB, 32KB and 1MB sequential and random read I/O.



(b) The write and erase counts under 4KB, 32KB and 1MB sequential and random read I/O.



(c) The write counts of onode, per-object-index and object data under sequential and random read I/Os.

Figure 40: The sequential and random performance under DMMbench.

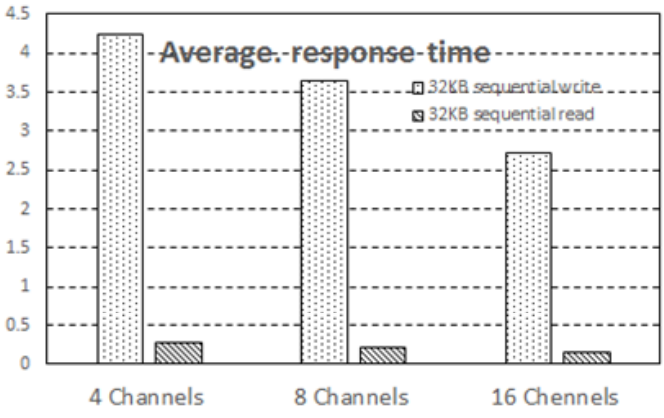
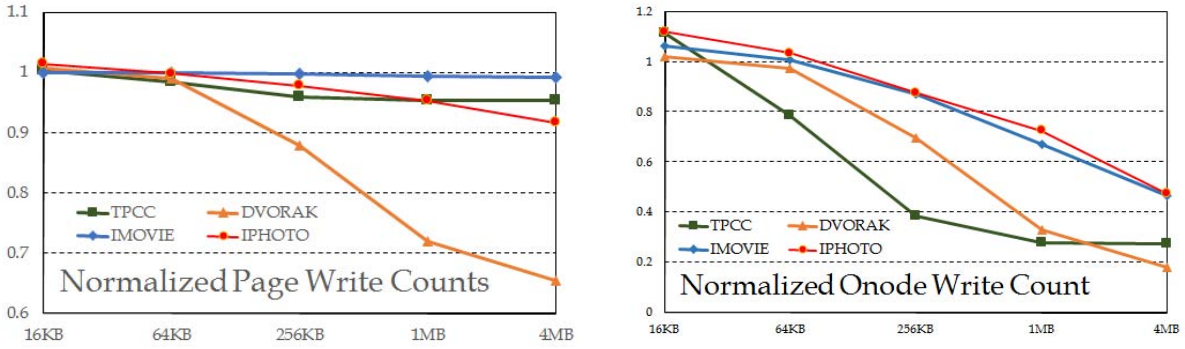


Figure 41: The sequential write and read response time under 4, 8 and 16 channels.



(a) The normalized page write counts under different byte-level GC table sizes.

(b) The normalized onode write counts under byte-level GC tables with different sizes.

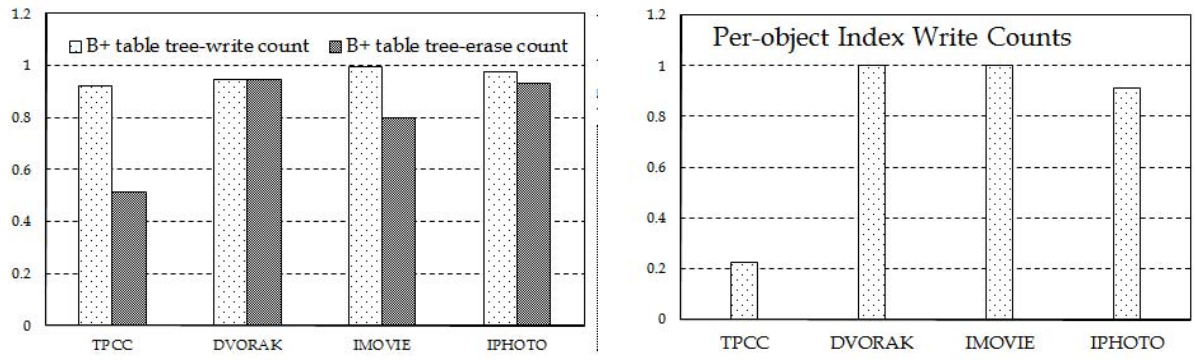
Figure 42: The performance of MLGC under different byte-level GC table sizes.

4.6.3 Evaluation of DMM Device Efficiency

In this section, we first evaluate the efficiency of the proposed MLGC, virtual B+ tree and diff cache, separately. Then, we will compare the overall performance of the DMM device with these of YAFFS2 and OBFS [19].

4.6.3.1 Evaluation of MLGC The write count of MLGC under different byte-level GC table sizes is shown in Fig. 42. As shown in Fig. 42(a), compared with the ONFD without MLGC, the systems with a 4MB byte-level GC table reduces page write count by 13% on average. Under 4MB byte-level GC table, the onode write count is reduced by 65% on average against the ONFD without MLGC. This result demonstrates that MLGC can effectively reduce onode write count with marginal memory resource.

The maximum page write reduction (65%) occurs in read-intensive DVORAK workload. Accordingly, the maximum onode write reduction (83%) also occurs under the DVORAK workload. The write reduction can be explained by the read incurred object metadata update. The read syscall invokes atime modification. Frequent read operations generates a large amount of inode update and invokes many onode writes. Hence, reduction of onode writes by the byte-level GC table greatly contributes to the page write reduction.



(a) The normalized page write and block erase counts under extent-based B+ tree and virtual B+ tree.

(b) The normalized per-object index and onode write counts under extent-based B+ tree and virtual B+ tree.

Figure 43: The efficiency of the virtual B+ tree.

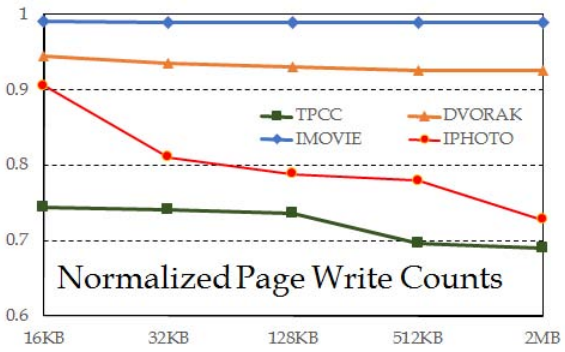
In all workloads, the page write counts have a slight increase under the 16KB byte-level GC table. As shown in Fig. 42(b), the onode write counts with a 16KB byte-level GC table are slightly more than that without MLGC. The page write increase can be explained by redistribution of reclaimed chunks. MLGC reduces updates to onode pages. Hence, the probability of reclaiming the chunks storing onode is reduced. More chunks storing object data (i.e., *object data chunk*) are reclaimed. Reclamation of object data chunks invokes updates to the corresponding per-object indices and onodes, directly leading to increase of data migration. Due to small table size, the MLGC cannot group enough amount of onode partial updates. Hence, the increased data migration outnumbers the onode write reduction by MLGC, leading to page write count increase. When the byte-level GC table size increases from 16KB to 4MB, the page write counts decrease. As show in Fig. 42(a) and Fig. 42(b), the page write count and onode write count decreases by 15% and 87% on average when the size of byte-level GC table increases from 16KB to 4MB. With large table size, the MLGC can group and merge a large mount of onode partial updates. Thus, MLGC can greatly reduce the data migration even through there is a slight increase of reclaimed object data chunks.

4.6.3.2 Evaluation of the virtual B+ tree The virtual B+ tree is evaluated under the default system configuration with a 1MB byte-level GC table. The efficiency of the virtual B+ tree is shown in Fig. 43. It it shown that the virtual B+ tree has different impacts under different workloads. The maximum data migration reduction of virtual B+ tree occurs under the TPCC workload. Fig. 43(a) shows that the page write count and block erase count are reduced by 55% and 58%, respectively. The per-object index write reduction reaches 20% as shown in Fig. 43(b). The TPCC workload is featured by frequent update to large-sized database tables which have high per-object index trees. Under the high extent-based B+ tree, update to a leaf node page invokes updates to several internal node pages. Comparatively, update to the internal node pages can be effectively reduced in the virtual B+ tree. The decrease of internal node page update also reduces the onode update since update to the per-object index root node page address is reduced.

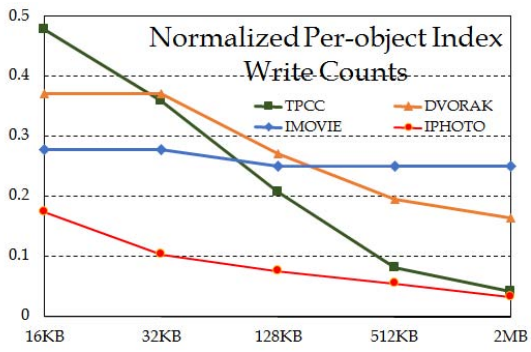
In comparison, other workloads only have 3% page write count reduction on average with the virtual B+ tree. Accordingly, there is only slight reduction on per-object index write count (4%). The slight write performance improvement can be explained by the fact that many of the frequent accessed objects have small size. These small-sized objects have data and onode stored together without per-object index or only have a leaf node page without internal node pages. Since the internal node pages only account for a small portion of per-object index node pages, reduction of the write operations to internal node pages by the virtual B+ tree only account for a slight reduction to the per-object index node pages write operation.

4.6.3.3 Evaluation of diff cache We evaluate the efficiency of the diff cache under the default system configuration with a 1MB byte-level GC table and the virtual B+ tree. The write count reduction under different diff cache sizes is shown in Fig. 44. As shown in Fig. 44(a) and Fig. 44(b), compared with the ONFD without the diff cache, the page write count and the per-object write count are reduced by 20% and 60% on average under a 2MB diff cache. The maximum page write reduction occurs to the TPCC workload: The page write count and per-object index count is reduced by 30% and 90%, respectively. The cache hit rate of the TPCC workload reaches 95% as shown in Fig. 44(c). In comparison,

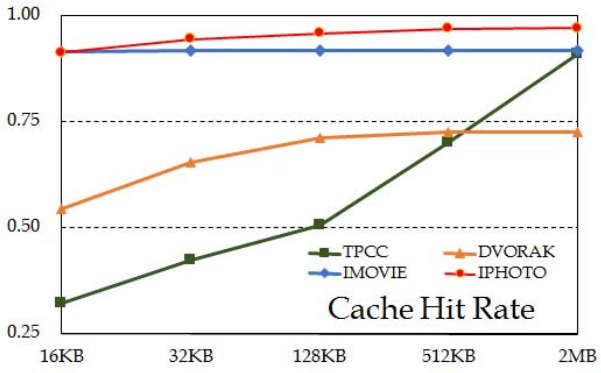
the diff cache has least page write reduction under the IMOVIE workload: With a 2MB diff cache, the page write count only decreases by 1%. However, there is 80% reduction in per-object index write. Per-object index write only accounts for a small part of total page write under the IMOVIE workload. This is because there are only a few write operations and most frequently written files are small files without per-object index. Hence, there is only a slight page write reduction. The page write reduction is 7% under the DVORAK workload. Similar to the IMOVIE workload, there are large per-object index write reduction (85%) but only a slightly page write reduction (7%).



(a) The normalized page write counts under different diff cache sizes.



(b) The normalized per-object index write counts under different diff cache sizes.

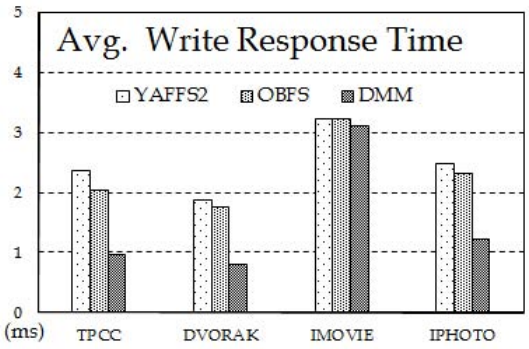


(c) The cache hit rates under different diff cache sizes.

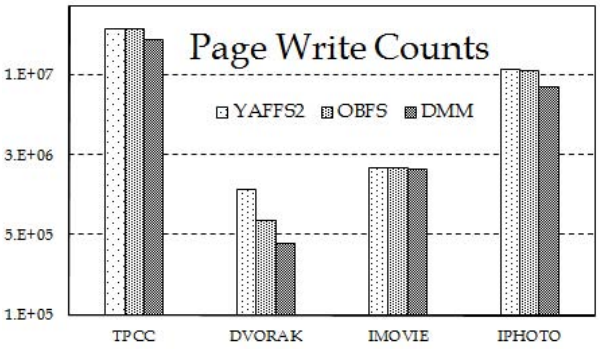
Figure 44: The efficiency of the diff cache.

The effect of diff cache size is also evaluated. As shown in Fig. 44(c), under the IMOVIE and IPHOTO workloads, the cache hit rate reaches up to 95% with only a 16KB diff cache.

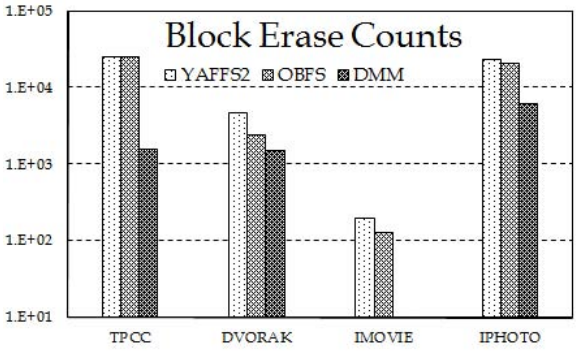
The high cache hit rate can be explained by small write working set. Comparatively, the cache hit rate of the 16KB diff cache is low under the TPCC workload. The low cache hit rate can be explained by random access patterns to the database tables. The cache hit rate increases to 95% when the diff cache size reaches 2MB. These results demonstrate that the diff cache can effectively reduce the per-object index write with only marginal memory consumption.



(a) The average write response time of the DMM device.



(b) The page write count of the DMM device.



(c) The block erase count of the DMM device.

Figure 45: The overall efficiency of the DMM device

4.6.3.4 The overall performance improvement Finally, we compare the overall performance of the DMM device with OBFS [19] and YAFFS2 [68]. The DMM device with the default system configuration has a 4MB byte-level GC table and a 2MB diff cache. We implement the functionality of OBFS and YAFFS2 in the ONFD simulator. The performance

improvement of the DMM device is shown in Fig. 45. Fig. 45(a) shows that compared with YAFFS2 and OBFS, the DMM device achieves the write response time reduction by 35% and 38% on average. Accordingly, the DMM device reduces the page write by 19% and 25%, respectively. Under the TPCC workload, the page write count is reduced up to 20% against OBFS. Due to the page write reduction, the DMM device improves the system endurance by 76% on average compared with OBFS.

4.7 SUMMARY

In the object-based NAND flash device (ONFD), onode partial update and cascading update are identified as two causes to write amplification. To minimize the data migration incurred by onode partial update and cascading update, the Data Migration Minimization (DMM) device design is proposed. To reduce the unnecessary write reduced by onode partial update, the multi-level garbage collection (MLGC) technique is proposed. MLGC groups invalid data induced by more than one onode partial updates to one physical page and moves the remaining valid data at one time. Thereby, migration of unnecessary data can be reduced. The virtual B+ tree is proposed to reduce cascading update within per-object index. With frequently updated physical addresses replaced by unchanged virtual addresses, update to parent node pages of per-object indices can effectively reduced. The diff cache is proposed to further reduce cascading update induced data migration. Due to limited size, the diff cache selectively buffers data and adopts a customized cache replacement policy to maximize per-object index update merging. Our experimental results show that, compared with the best previous work, the DMM device design can reduce page write by 20% with 76% system lifetime extension. To evaluate the efficiency of the proposed DMM device, we design a ONFD simulator, ObjNandSim. The ObjNandSim implements an ONFD architecture in the software component and emulates the hardware behavior of real NAND flash memory array. The simulation results demonstrate the validity of the ObjNandSim.

5.0 CONCLUSION AND FUTURE WORK

5.1 DISSERTATION CONCLUSION

Thanks to the fast access time, good scalability and low power consumption, NAND flash memory has been adopted in various storage systems ranging from low-power embedded systems to high-end enterprise servers. Despite efforts in exploring data protection schemes and architectural optimization, the system designers and researchers are still faced with challenges from three aspects: 1) Limited memory endurance due to high bit error rate (BER), 2) Long data protection overhead and 3) Write amplification due to out-of-update property.

Bit errors in the NAND flash memory results from the intrinsic device noises: random telegraph noise (RTN), cell-to-cell interference and retention time limit [2, 3, 4, 5]. To protect data integrity, hard-decision ECCs such as Bose-Chaudhuri-Hocquenghem (BCH) code can be employed to protect data integrity [69]. The noises rise up as program/erase (P/E) cycle increases, leading to high BER and limiting the memory endurance. To protect data integrity under high BER and extend system lifetime, high hardware cost is incurred. To handle the issues of limited system lifetime, we proposed a Data Pattern Aware (DPA) error prevention technique to extend the lifespan of NAND flash storage systems in Chapter 2. We note that the V_{th} level L0 is resistant to retention time error and cell-to-cell interference. Based on this observation, we proposed Pattern Probability Unbalance (DPA-PPU) technique to unbalance the number of 1's and 0's in the data. As such, more cells are placed on the V_{th} level L0. We also proposed Data Redundancy Management (DPA-DRM) to mitigate the performance overhead induced by DPA-PPU. Experimental results show that DPA can improve the NAND flash lifetime by 3× or 4× with no or $\sim$ 10% performance overhead, respectively. The incurred hardware cost is very marginal.

With technology node further scaling down, BER increases to 10^{-2}. Hard-decision ECCs and our proposed DPA either induces prohibitively high hardware cost or intolerable storage overhead. To provide more powerful error correction capability, low-density parity-check (LDPC) code is introduced in the NAND flash based storage system. However, the LDPC code incurs high decoding overhead, directly degrading read performance. To reduce LDPC-induced read latency, we proposed the FlexLevel technique in Chapter 3. The proposed device-level LevelAdjust technique can dynamically reduce BER via V_{th} level reduction. By minimizing BER, extra sensing levels can be effectively reduced and read performance is improved. To balance performance improvement and density loss, we proposed the AccessEval technique at the system level. Instead of employing LevelAdjust to all data stored in NAND flash, AccessEval only applies LevelAdjust to the data with high LDPC overhead. LDPC overhead is effectively reduced while the incurred capacity loss is kept at a minimum level. Simulation results show that compared with the best prior works, the proposed design can achieve read speedup by up to 11% with negligible capacity loss.

Besides the reliability issue, the performance degradation incurred by write amplification is another issues that the system designers have to tackle. In the object-based NAND flash device (ONFD), write amplification is induced by onode partial update and cascading update. Partial page update results from the inconsistent minimum write size between the object-based interface and NAND flash memory. Under the byte-unit access object-based interface, onode partial update always induces a write to an entire NAND flash page, leading to unnecessary write operation. Cascading update results from the out-of-update proberty: The object data update generates the per-object index and onode update in a cascading manner, resulting in large amount of unnecessary data migration. To reduce write amplification incurred by onode partial update and cascading update, we proposed the Data Migration Minimization (DMM) device in Chapter 4. We proposed a multi-level garbage collection (MLGC) technique to reduce onode partial update induced data migration. MLGC groups invalid data of more than one onode partial update and moves the remaining valid bytes of the same page at one time. As such, data migration can be reduced. The virtual B+ tree is proposed to reduce cascading update within per-object index. By using virtual address instead of physical address, updating the leaf nodes address does not incur the update to the

parent node pages in the virtual B+ tree. Thereby, the data migration of per-object indices can be effectively reduced. The diff cache is proposed to merge the per-object index updates by on-device DRAM. Due to limited size, the diff cache only buffers the updated per-object indices and onodes with the customized diff cache replacement policy. Our experimental results show that compared with the best previous work, the DMM device design can reduce response time by 35% with 20% data migration reduction.

5.2 FUTURE WORK

In Chapter 4, we focus on elimination of unnecessary write induced by onode partial update and cascading update. For future research work, we will concentrate on optimization of data layout to improve the wear-leveling efficiency. In this section, we will investigate the relationship between the data layout and the wear-leveling efficiency.

In the existing ONFD architecture (Fig. 26), the data storage layout is shown in Fig. 46. The physical storage space is divided into a number of chunks. One chunk contains one or more NAND flash blocks. The chunk without any data is a free chunk. The free chunk can be assigned to store any type of data. Once it has data, a chunk will become one of the three types of chunks depending on data stored in it: 1) Data chunk which stores object data, 2) index chunk storing per-object indices and 3) onode chunk storing object onodes. The chunk is the basic garbage collection unit : When the number of free chunks is under a threshold, the dirty chunks are selected and reclaimed for the upcoming write requests. Due to different access patterns, separately storing different types of data in different chunks offers better cold/hot data isolation. As such, the amount of migrating data can be reduced since the chunk with hot data is frequently reclaimed [56].

However, the data separation leads to reduction of wear-leveling efficiency. Compared with the object data, onodes and the per-object indices are more frequently updated. The onodes chunks and the per-object chunks are selected for reclamation at most times. The onode and per-object index chunks are repeatedly erased while the data chunks are less frequently moved. As shown in Fig. 47, per-object indices and onodes are stored in Chunk 3

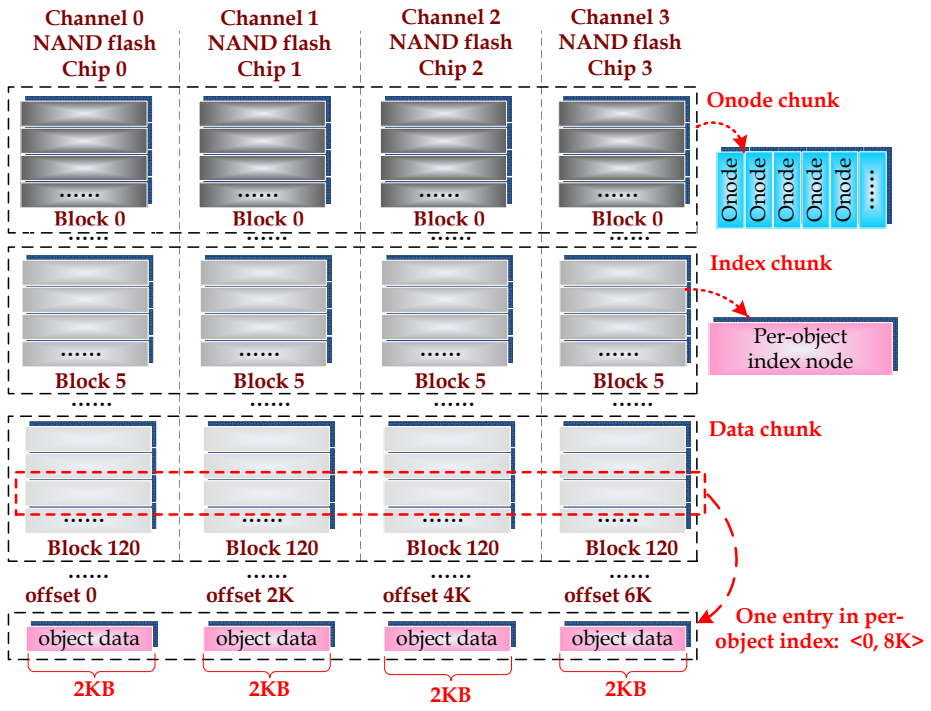


Figure 46: The data layout of existing ONFD.

and Chunk 0. Due to frequent update, per-object indices and onodes are stored to Chunk 0, 3, 4 and 5 repeatedly. Under the highly biased chunk reclamation, the data chunks wear out much slowly than other types of chunks. The data chunks stay young while other types of chunks grow old fast. To wear out all chunks evenly, the ONFD should manually swap the data stored in data chunks and other types of chunks during wear-leveling, which directly increases data migration. In our further work, we will propose solutions to reduce the data migration between old and young chunks by mitigating biased reclamation.

5.3 DISSERTATION SUMMARY

NAND flash memory has demonstrated wide application in the existing storage systems. However, limited memory endurance, long data protection overhead and write amplification continue to be the most critical design challenges. In this dissertation, we proposed the Data Pattern Aware (DPA), Flexlevel and Data Migration Minimization (DMM) techniques

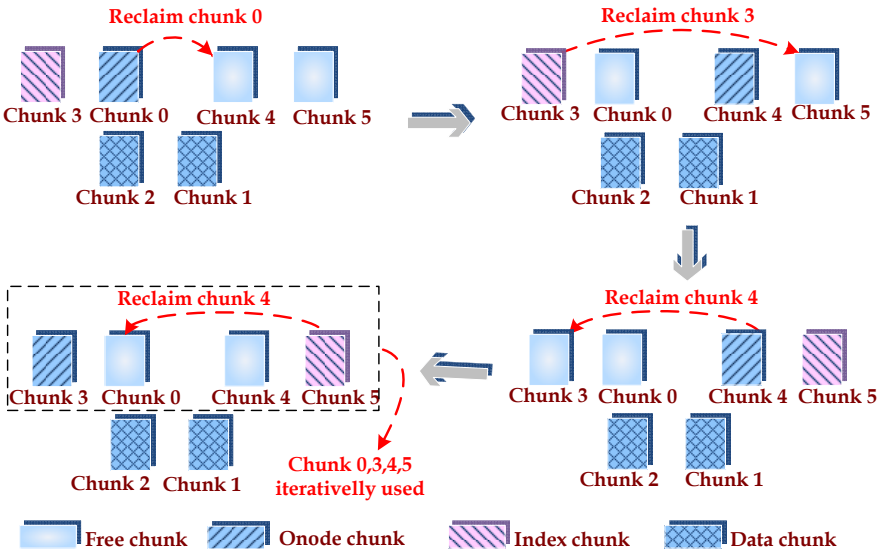


Figure 47: The biased chunk reclamation issue in ONFD.

to handle these three issues. In these works, we adopted Monte Carlo simulation for error pattern characterization. We employed an open-source simulator Flashsim to evaluate the efficiency of DPA and Flexlevel techniques. In addition, to evaluate the proposed DDM technique, we developed a ONFD simulator. Due to small hardware overhead, the FlexLevel technique is a possible solution that can be used in industry to reduce LDPC induced overhead. Our developed ONFD provides a easy-to-use platform for other researchers to evaluate their solution and algorithms of ONFD.

BIBLIOGRAPHY

[1] K. Vatto, "Samsung ssd 840: Testing the endurance of tlc nand," http://www.anandtech.com/show/6459/samsung-ssd-840-testing-the-endurance-of-tlc-nand.

[2] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L. R. Nevill, "Bit error rate in nand flash memories," in *International Reliability Physics Symposium (IRPS)*. IEEE, 2008, pp. 9–19.

[3] 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 *IEEE 18th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2012, pp. 1–10.

[4] Y. Cai, E. F. Haratsch, O. Mutlu, and K. Mai, "Error patterns in mlc nand flash memory: Measurement, characterization, and analysis," in *Conference on Design, Automation and Test in Europe (DATE)*. IEEE, 2012, pp. 521–526.

[5] H. Sun, P. Grayson, and B. Wood, "Quantifying reliability of solid-state storage from multiple aspects," in *IEEE International Workshop on Storage Network Architecture and Parallel I/O (SNAPI)*, vol. 11, 2011.

[6] Y. Cai, G. Yalcin, O. Mutlu, E. F. Haratsch, A. Cristal, O. S. Unsal, and K. Mai, "Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime," in *2012 IEEE International Conference on Computer Design (ICCD)*. IEEE, 2012, pp. 94–101.

[7] S. Tanakamaru, K. Takeuchi, and M. Doi, "Error-prediction analyses in 1x, 2x and 3xnm nand flash memories for system-level reliability improvement of solid-state drives (ssds)," in *2013 IEEE International Reliability Physics Symposium (IRPS)*. IEEE, 2013, pp. 3B–3.

[8] J. Wang, T. Courtade, H. Shankar, and R. D. Wesel, "Soft information for ldpc decoding in flash: Mutual-information optimized quantization," in *Global Telecommunications Conference (GLOBECOM)*. IEEE, 2011, pp. 1–6.

[9] G. Dong, N. Xie, and T. Zhang, "On the use of soft-decision error-correction codes in nand flash memory," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 58, no. 2, pp. 429–439, 2011.

[10] Y. Lu, J. Shu, and W. Zheng, "Extending the lifetime of flash-based storage through reducing write amplification from file systems." in *File and Storage Technologies(FAST)*, 2013, pp. 257–270.

[11] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *System and Storage Conference (SYSTOR)*, 2009, p. 10.

[12] K. Zhao, W. Zhao, H. Sun, X. Zhang, N. Zheng, and T. Zhang, "Ldpc-in-ssd: Making advanced error correction codes work effectively in solid state drives." in *File and storage technologies (FAST)*, 2013, pp. 243–256.

[13] J. Yoon and G. Tressler, "Advanced flash technology status, scaling trends and implications to enterprise ssd technology enablement," *Flash Memory Summit*, 2012.

[14] J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang, "Efficient identification of hot data for flash memory storage systems," *ACM Transactions on Storage (TOS)*, vol. 2, no. 1, pp. 22–40, 2006.

[15] S. Lee, D. Shin, Y.-J. Kim, and J. Kim, "Last: locality-aware sector translation for nand flash memory-based storage systems," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 6, pp. 36–42, 2008.

[16] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Consistency without ordering." in *File and Storage Technologies (FAST)*, 2012, p. 9.

[17] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "De-indirection for flash-based ssds with nameless writes," in *File and Storage Technologies (FAST)*, 2012, p. 1.

[18] A. Rajimwale, V. Prabhakaran, and J. D. Davis, "Block management in solid-state devices," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2009, pp. 279–284.

[19] Y. Kang, J. Yang, and E. L. Miller, "Object-based scm: An efficient interface for storage class memories," in *IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2011, pp. 1–12.

[20] W. Wang, Y. Lu, and J. Shu, "P-oftl: An object-based semantic-aware parallel flash translation layer," in *Proceedings of the conference on Design, Automation & Test in Europe (DATE)*. European Design and Automation Association, 2014, p. 157.

[21] Y.-G. Lee, D. Jung, D. Kang, and J.-S. Kim, "μ-ftl:: a memory-efficient flash translation layer supporting multiple mapping granularities," in *Proceedings of the ACM international conference on Embedded software (EMSOFT)*. ACM, 2008, pp. 21–30.

[22] D. Kang, D. Jung, J.-U. Kang, and J.-S. Kim, "μ-tree: an ordered index structure for nand flash memory," in *Proceedings of the ACM & IEEE international conference on Embedded software (EMSOFT)*, 2007, pp. 144–153.

[23] J. Moon, J. No, S. Lee, S. Kim, J. Yang, and S. H. Chang, "Noise and interference characterization for mlc flash memories," in *International Conference on Computing, Networking and Communications (ICNC)*, 2012, pp. 588–592.

[24] J. Guo, Z. Chen, D. Wang, Z. Shao, and Y. Chen, "Dpa: A data pattern aware error prevention technique for nand flash lifetime extension," in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2014, pp. 592–597.

[25] 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 *52nd Annual Design Automation Conference (DAC)*. ACM, 2015, p. 194.

[26] K. Takeuchi, T. Tanaka, and T. Tanzawa, "A multipage cell architecture for high-speed programming multilevel nand flash memories," *Journal of Solid-State Circuits (JSSC)*, vol. 33, no. 8, pp. 1228–1238, 1998.

[27] G. Dong, S. Li, and T. Zhang, "Using data postcompensation and predistortion to tolerate cell-to-cell interference in mlc nand flash memory," *IEEE Transactions on Circuits and Systems I: Regular Papers (TCAS)*, vol. 57, no. 10, pp. 2718–2728, 2010.

[28] L. Cola, M. De Tomasi, R. E. Vaion, A. Mervic, and P. Zabberoni, "Read disturb on flash memories: Study on temperature annealing effect," *Microelectronics Reliability*, vol. 52, no. 9, pp. 1803–1807, 2012.

[29] C. M. Compagnoni, R. Gusmeroli, A. S. Spinelli, and A. Visconti, "Analytical model for the electron-injection statistics during programming of nanoscale nand flash memories," *IEEE Transactions on Electron Devices*, vol. 55, no. 11, pp. 3192–3199, 2008.

[30] H. P. Belgal, N. Righos, I. Kalastirsky, J. J. Peterson, R. Shiner, and N. Mielke, "A new reliability model for post-cycling charge retention of flash memories," in *Annual Reliability Physics Symposium Proceedings*. IEEE, 2002, pp. 7–20.

[31] K. Fukuda, Y. Shimizu, K. Amemiya, M. Kamoshida, and C. Hu, "Random telegraph noise in flash memories-model and technology scaling," in *IEEE International Electron Devices Meeting (IEDM)*, 2007, pp. 169–172.

[32] M. R. Stan and W. P. Burleson, "Low-power encodings for global communication in cmos vlsi," *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, vol. 5, no. 4, pp. 444–455, 1997.

[33] S. Li and T. Zhang, "Improving multi-level nand flash memory storage reliability using concatenated bch-tcm coding," *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, vol. 18, no. 10, pp. 1412–1420, 2010.

[34] Y. Maeda and H. Kaneko, "Error control coding for multilevel cell flash memories using non-binary low-density parity-check codes," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*. IEEE, 2009, pp. 367–375.

[35] B. Shin, C. Seol, J.-S. Chung, and J. J. Kong, "Error control coding and signal processing for flash memories," in *2012 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2012, pp. 409–412.

[36] B. Chen, X. Zhang, and Z. Wang, "Error correction for multi-level nand flash memory using reed-solomon codes," in *IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE, 2008, pp. 94–99.

[37] Y. Lee, S. Jung, and Y. H. Song, "Fra: a flash-aware redundancy array of flash storage devices," in *IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*. ACM, 2009, pp. 163–172.

[38] "A simulator for various ftl scheme," http://csl.cse.psu.edu/?q=node/322.

[39] "Oltp application i/o and search engine i/o," http://traces.cs.umass.edu/index.php/storage/storage.

[40] S. Tanakamaru, Y. Yanagihara, and K. Takeuchi, "Highly reliable solid-state drives (ssds) with error-prediction ldpc (ep-ldpc) architecture and error-recovery scheme," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2013, pp. 83–84.

[41] G. Dong, N. Xie, and T. Zhang, "On the use of soft-decision error-correction codes in nand flash memory," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 58, no. 2, pp. 429–439, 2011.

[42] G. Dong, Y. Zou, and T. Zhang, "Reducing data transfer latency of nand flash memory with soft-decision sensing," in *International Conference on Communications (ICC)*. IEEE, 2012, pp. 7024–7028.

[43] N. H. Seong, S. Yeo, and H.-H. S. Lee, "Tri-level-cell phase change memory: toward an efficient and reliable memory system," in *International Symposium on Computer Architecture (ISCA)*, 2013, pp. 440–451.

[44] K. Prall, "Scaling non-volatile memory below 30nm," in *IEEE Non-Volatile Semiconductor Memory Workshop (NVMW)*. IEEE, 2007, pp. 5–10.

[45] R.-S. Liu, C.-L. Yang, and W. Wu, "Optimizing nand flash-based ssds via retention relaxation," in *File and storage technologies (FAST)*, 2012, pp. 243–256.

[46] Y. Cai, E. F. Haratsch, O. Mutlu, and K. Mai, "Threshold voltage distribution in mlc nand flash memory: Characterization, analysis, and modeling," in *Conference on Design, Automation and Test in Europe (DATE)*, 2013, pp. 1285–1290.

[47] C. Sun, K. Miyaji, K. Johguchi, and K. Takeuchi, "Scm capacity and nand over-provisioning requirements for scm/nand flash hybrid enterprise ssd," in *International Memory Workshop (IMW)*, 2013, pp. 64–67.

[48] D. Park and D. H. Du, "Hot and cold data identification for flash memory using multiple bloom filters," in *File and Storage Technologies (FAST)*, 2011.

[49] T. Luo, R. Lee, M. Mesnier, F. Chen, and X. Zhang, "hstorage-db: heterogeneity-aware data management to exploit the full capability of hybrid storage systems," *Very Large Data Base Endowment (VLDB Endowment)*, vol. 5, no. 10, pp. 1076–1087, 2012.

[50] J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang, "Efficient identification of hot data for flash memory storage systems," *ACM Transactions on Storage (TOS)*, vol. 2, no. 1, pp. 22–40, 2006.

[51] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar, "Flashsim: A simulator for nand flash-based solid-state drives," in *1st International Conference on Advances in System Simulation (SIMUL'09)*. IEEE, 2009, pp. 125–131.

[52] L. M. Grupp, J. D. Davis, and S. Swanson, "The bleak future of nand flash memory," in *File and Storage Technologies (FAST)*, 2012, pp. 2–2.

[53] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design tradeoffs for ssd performance," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2008, pp. 57–70.

[54] D. Nagle, M. Factor, S. Iren, D. Naor, E. Riedel, O. Rodeh, and J. Satran, "The ansi t10 object-based storage standard and current implementations," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 401–411, 2008.

[55] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 26–52, 1992.

[56] Y.-S. Lee, S.-H. Kim, J.-S. Kim, J. Lee, C. Park, and S. Maeng, "Ossd: A case for object-based solid state drives," in *IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2013, pp. 1–13.

[57] S. T. On, H. Hu, Y. Li, and J. Xu, "Lazy-update b+-tree for flash devices," in *IEEE International Conference on Mobile Data Management: Systems, Services and Middleware (MDM)*, 2009, pp. 323–328.

[58] O. S. Center, "Osc software osd implementation," http://jp.jplovetv.com/2015/09/special-drama-20150905.html.

[59] R. Van Riel, "Page replacement in linux 2.4 memory management." in *USENIX Annual Technical Conference (ATC) FREENIX Track*, 2001, pp. 165–172.

[60] H. Kim and S. Ahn, "Bplru: A buffer management scheme for improving random writes in flash storage." in *File and Storage Technologies (FAST)*, vol. 8, 2008, pp. 1–14.

[61] B. Harrosh and B. Halevy, "The linux exofs object-based pnfs metadata server," 2009.

[62] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin, "An implementation of a log-structured file system for unix," in *Proceedings of the USENIX Winter 1993 Conference*. USENIX Association, 1993, pp. 3–3.

[63] C. Sun, K. Miyaji, K. Johguchi, and K. Takeuchi, "Scm capacity and nand over-provisioning requirements for scm/nand flash hybrid enterprise ssd," in *2013 5th IEEE International Memory Workshop (IMW)*. IEEE, 2013, pp. 64–67.

[64] "Hammerora: the open source oracle load test tool," http://hammerora.sourceforge.net/faq.htm.

[65] "Strace - trace system calls and signals," http://man7.org/linux/man-pages/man1/strace.1.html.

[66] C. M. University, "Coda project traces and dfstrace," http://coda.cs.cmu.edu/DFSTrace/.

[67] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "A file is not a file: understanding the i/o behavior of apple desktop applications," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 3, p. 10, 2012.

[68] "Yaffs a flash file system for embedded use," http://www.yaffs.net/.

[69] S. Li and T. Zhang, "Improving multi-level nand flash memory storage reliability using concatenated bch-tcm coding," *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, vol. 18, no. 10, pp. 1412–1420, 2010.