HARDWARE/SOFTWARE CODESIGN FOR PERFORMANCE AND LIFETIME ENHANCEMENT IN NAND-FLASH-BASED EMBEDDED STORAGE SYSTEMS

RENHAI CHEN

Ph.D

The Hong Kong Polytechnic University

2016

THE HONG KONG POLYTECHNIC UNIVERSITY

DEPARTMENT OF COMPUTING

Hardware/Software Codesign for Performance and Lifetime Enhancement in NAND-Flash-based Embedded Storage Systems

By

RENHAI CHEN

A Thesis Submitted in Partial Fulfillment of

the Requirements for the Degree of

Doctor of Philosophy

May 2016

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

_____(Signature)

CHEIV RENHAI (Name of Student)

ABSTRACT

Embedded systems (e.g. smartphones) have become an integral part in peoples' daily life. In the last several years, most of efforts in improving embedded systems have been focusing on enhancing the network performance and CPU speed. On the other hand, the performance of NAND-flash-based storage in embedded systems is stagnant, and has become one of the major performance bottlenecks in embedded systems. Even worse, the performance and lifetimes of NAND flash memory are substantially degraded with the advent of multi-level cell and triple-level cell flash memory for increasing the capacity. In this thesis, we have addressed these issues from several aspects including the integration of the emerging hardware and the cross-layer software management for the optimization of the performance and lifetime.

First, we focus on employing the self-healing NAND flash memory to improve the lifetime and performance. Researchers have recently discovered that heating can cause worn-out NAND flash cells to become reusable and greatly extend the lifetime of flash memory cells. However, the heating process consumes a substantial amount of power, and some fundamental changes are required for existing NAND flash management techniques. In particular, all existing wear-leveling techniques are based on the principle of evenly distributing writes and erases. For self-healing NAND flash, this may cause NAND flash cells to be worn out in a short period of time. Moreover, frequently healing these cells may drain the energy quickly in battery-driven mobile devices, which is defined as the concentrated heating problem. We propose a novel wear-leveling scheme called DHeating (Dispersed Heating) to address the problem. In DHeating, rather than evenly distributing writes and erases over a time period, write and erase operations are scheduled on a small number of flash memory cells at a time, so that these cells can be worn out and healed much earlier than other cells. In this

way, we can avoid quick energy depletion caused by concentrated heating. In addition, the heating process takes several seconds and has become the new performance bottleneck. In order to address this issue, we propose a lazy heating repair scheme. The lazy heating repair scheme can ease the long time heating effect by delaying the heating operation and using the system idle time to repair. Furthermore, the flash memory's reliability becomes worse with the flash memory cells reaching the expected worn-out time. We propose an early heating strategy to solve the reliability problem. With the extended lifetime provided by self-healing, we can trade some lifetimes for reliability. The idea is to start the healing process earlier than the expected worn-out time. We evaluate our scheme based on an embedded platform. The experimental results show that the proposed scheme can effectively prolong the consecutive heating time interval, alleviate the long time heating effect, and enhance the reliability for self-healing flash memory.

Second, we jointly optimize the NAND flash memory's lifetime and performance with the integration of NVMs. Novel NVMs (non-volatile memories), such as PCM (Phase Change Memory) and STT-RAM (Spin-Transfer Torque Random Access Memory), can provide fast read/write operations. In this thesis, we propose a unified NVM/flash architecture to improve the I/O performance. A transparent scheme, vFlash (Virtualized Flash), is also proposed to manage the unified architecture. Within vFlash, inter-app and intra-app techniques are proposed to optimize the application performance by exploiting the historical locality and I/O access patterns of applications. Since vFlash is on the bottom of the I/O stack, the application features will be lost. Therefore, we also propose a cross-layer technique to transfer the application information from the application layer to the vFlash layer. The proposed scheme is evaluated based on an Android platform, and the experimental results show that the proposed scheme can effectively improve the I/O performance of mobile devices.

Third, we study the problem of performance and lifetime enhancement in the mobile virtualization environment. Mobile virtualization introduces extra layers in software stacks, which leads to performance degradation. Especially, each I/O operation has to pass through

several software layers to reach the NAND-flash-based storage systems. This thesis targets at optimizing I/O for mobile virtualization, since I/O becomes one of major performance bottlenecks that seriously affects the performance of mobile devices. Among all the I/O operations, a large percentage is updating metadata. Frequent updating metadata not only degrades overall I/O performance but also severely reduces flash memory lifetime. In this thesis, we propose a novel I/O optimization technique to identify the metadata of a guest file system which is stored in a VM (Virtual Machine) image file and frequently updated. Then, these metadata are stored in a small additional NVM (Non-Volatile Memory) which is faster and more endurable to greatly improve flash memory's performance and lifetime. To the best of our knowledge, this is the first work to identify the file system metadata from regular data in a guest OS VM image file under mobile virtualization. The proposed scheme is evaluated on a real hardware embedded platform. The experimental results show that the proposed techniques can improve write performance to 45.21% in mobile devices with virtualization.

Keywords: NAND flash memory, self-healing, wear leveling, non-volatile memory, mobile virtualization, power consumption.

PUBLICATIONS

Journal Papers

1. **Renhai Chen**, Yi Wang, Duo Liu, Zili Shao, and Song Jiang, "Heating Dispersal for Self-Healing NAND Flash Memory", Accepted in *IEEE Transactions on Computers (TC)*, 2016.

2. **Renhai Chen**, Yi Wang, Jingtong Hu, Duo Liu, Zili Shao, and Yong Guan, "Image-Content-Aware I/O Optimization for Mobile Virtualization", Accepted in *ACM Transactions on Embedded Computing Systems (TECS)*, 2016.

3. Yi Wang, Zhiwei Qin, **Renhai Chen**, Zili Shao, and Laurence T. Yang, "An Adaptive Demand-Based Caching Mechanism for NAND Flash Memory Storage Systems", Accepted in *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2016.

4. Yi Wang, Zhiwei Qin, **Renhai Chen**, Zili Shao, Qixin Wang, Shuai Li, and Laurence T. Yang, "A Real-Time Flash Translation Layer for NAND Flash Memory Storage Systems", Accepted in *IEEE Transactions on Multi-Scale Computing Systems (TMSCS)*, 2016.

5. **Renhai Chen**, Zhaoyan Shen, Chenlin Ma, Zili Shao, Yong Guan, "NVMRA: Utilizing NVM to Improve the Random Write Operations for NAND-Flash-Based Mobile Devices", *Software: Practice and Experience (SPE)*, 46: 1263-1284, 2016.

6. **Renhai Chen**, Zhiwei Qin, Yi Wang, Duo Liu, Zili Shao, Yong Guan, "On-Demand Block-Level Address Mapping in Large-Scale NAND Flash Storage Systems", *IEEE Transactions on Computers (TC)*, vol.64, no.6, pp. 1729-1741, 2015.

7. Hongxing Wei, Zhenzhou Shao, Zhen Huang, **Renhai Chen**, Yong Guan, Jindong Tan, Zili Shao, RT-ROS: A Real-time ROS Architecture on Multi-Core Processors", Accepted in *Future Generation Computer Systems (FGCS)*, 2015.

8. Yi Wang, Zhiping Jia, **Renhai Chen**, Meng Wang, Duo Liu, Zili Shao, "Loop Scheduling with Memory Access Reduction Subject to Register Constraints for DSP Applications", *Software: Practice and Experience (SPE)*, vol. 44, issue 8, pp. 999-1026, 2014.

Conference Papers

1. **Renhai Chen**, Zili Shao, and Tao Li, "Bridging the I/O Performance Gap for Big Data Workloads: A New NVDIMM-based Approach", *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2016)*, Taipei, Taiwan, October 15-19, 2016.

2. **Renhai Chen**, Zili Shao, Chia-Lin Yang, Tao Li, "MCSSim: A Memory Channel Storage Simulator", *21th ACM/IEEE Asia and South Pacific Design Automation Conference (ASP-DAC)*, Macau, China, January 25-28, 2016.

3. **Renhai Chen**, Yi Wang, Jingtong Hu, Duo Liu, Zili Shao, Yong Guan, "Virtual Machine Image Content Aware I/O Optimization for Mobile Virtualization", *12th IEEE International Conference on Embedded Software and Systems (ICESS)*, New York, USA, August 24-26, 2015.

4. **Renhai Chen**, Yi Wang, Jingtong Hu, Duo Liu, Zili Shao, Yong Guan, "Unified Non-Volatile Memory and NAND Flash Memory Architecture in Smartphones", *20th ACM/IEEE Asia and South Pacific Design Automation Conference (ASP-DAC)*, Chiba/ Tokyo, Japan, January 19-22, 2015.

5. **Renhai Chen**, Yi Wang, Jingtong Hu, Duo Liu, Zili Shao, Yong Guan, "Virtual-Machine Metadata Optimization for I/O Traffic Reduction in Mobile Virtualization", *IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, Chongqing, China, August 20-21, 2014.

6. **Renhai Chen**, Yi Wang, Zili Shao, "DHeating: Dispersed Heating Repair for Self-Healing NAND Flash Memory", *ACM/IEEE International Conference on Hardware-Software Codesign and System Synthesis (CODES+ISSS)*, Montreal, Canada, September 29- October 4, 2013.

7. Chi Zhang, Yi Wang, Tianzheng Wang, **Renhai Chen**, Duo Liu, Zili Shao, "Deterministic Crash Recovery for NAND Flash based Storage Systems", *51st ACM/IEEE Design Automation Conference (DAC)*, San Francisco, CA, USA, June 1-5, 2014.

8. Yi Wang, **Renhai Chen**, Zili Shao, Tao Li, "SolarTune: Real-Time Scheduling with Load Tuning for Solar Energy Powered Multicore Systems", *19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Taipei, Taiwan, August 19-21, 2013.

9. Yong Guan, Guohui Wang, Yi Wang, **Renhai Chen**, Zili Shao, "BLog: Block-level Log-block Management for NAND Flash Memory Storage Systems", *ACM SIGPLAN/ SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*, Seattle, Washington, June 20-21, 2013.

ACKNOWLEDGEMENTS

First and foremost, I want to express my gratitude to my supervisor, Prof. Zili Shao, whose expertise, understanding, and patience, added considerably to my Ph.D. study. I appreciate his vast knowledge and skill in many areas and his professional supervision. It is my great pleasure to work with Prof. Shao, and I want to thank him for supporting me over the years, and for giving me so much freedom to explore and discover new areas of research. Without his help and support, this body of work would not have been possible.

I also want to thank the other members of Prof. Shao's research group - Yi Wang, Zhiwei Qin, Duo Liu, Qi Zhang, Tianzheng Wang, Junhui Wang, Min Huang, Zhaoyan Shen, Chenlin Ma, and Yimei Kang - for the assistance they provided during my Ph.D. study. I also would like to thank all my teachers from whom I learned so much in my long journey of formal education.

I want to thank Prof. Chun Bun Henry CHAN from the Hong Kong Polytechnic University for kindly being the Chairman of the Board of Examiners (BoE), and the external examiners for taking their precious time to review my thesis and provide the valuable comments.

I recognize that this thesis would not have been possible without the financial assistance from the Hong Kong Polytechnic University. I appreciate Prof. Shao and the Department of Computing for offering me the travel grants to attend several international conferences.

Finally, I want to thank my family. They educated and guided me and have watched over me every step of way. I want to thank them for their care and love, support, and encouragement through my entire PhD study, for letting me pursue my dream for so long and so far

away from home, and for giving me the motivation to finish this thesis. Special thanks to my wife, who witnesses the joys and sorrows of my PhD study miles away. I am grateful for her endless love, patience, understanding and support.

TABLE OF CONTENTS

CERTIFICATE OF ORIGINALITY ii

ABSTRACT iii

PUBLICATIONS vi

ACKNOWLEDGEMENTS ix

LIST OF FIGURES xiv

LIST OF TABLES xvi

CHAPTER 1. INTRODUCTION 1
1.1 Related Work 3
1.1.1 Wear Leveling 3
1.1.2 Non-Volatile Memory Integration 4
1.1.3 Software Managed Flash 5
1.2 The Unified Research Framework 6
1.3 Contributions 9
1.4 Organization 10

CHAPTER 2. HEATING DISPERSAL FOR SELF-HEALING NAND FLASH MEMORY 11
2.1 Overview 11
2.2 Background and Motivation 13
2.2.1 NAND Flash Memory and Worn Out 13
2.2.2 Self-Healing NAND Flash Memory and Heating Repair 15
2.2.3 The Architecture of Self-Healing NAND Flash Memory 16
2.2.4 Motivating Example 18
2.3 The DHeating Scheme 19
2.3.1 Overview 19
2.3.2 Dispersed Heating 20
2.3.3 Lazy Heating Repair 24

2.3.4 Early Heating 26
2.3.5 DHeating Working with NFTL 27
2.3.6 Performance and Overhead Analysis 29
2.4 Evaluation 30
2.4.1 Experimental Setup 30
2.4.2 Results and Discussion 32
2.5 Summary 41

CHAPTER 3. VIRTUALIZED FLASH FOR OPTIMIZING THE I/O PERFORMANCE IN MOBILE DEVICES 43
3.1 Introduction 43
3.2 Background 45
3.2.1 The Android I/O System Architecture 45
3.2.2 I/O Bottleneck and Non-Volatile Memory 46
3.2.3 Transparent Integration 48
3.3 vFlash Design 49
3.3.1 Overview 49
3.3.2 Cross Layer Management 51
3.3.3 Inter-app and Intra-app Management 54
3.3.4 Adaptive Space Preparation 56
3.4 Performance and Overhead Analysis 59
3.5 Evaluation 61
3.5.1 Experimental Setup 62
3.5.2 Results and Discussion 64
3.6 Summary 71

CHAPTER 4. IMAGE-CONTENT-AWARE I/O OPTIMIZATION FOR MOBILE VIRTUALIZATION 73
4.1 Introduction 73
4.2 Background 75
4.2.1 NAND Flash Memory and PCM 76
4.2.2 Storage Architecture 77
4.2.3 Metadata Analysis 77
4.3 Motivation 83
4.3.1 Metadata Effect 83
4.3.2 Motivational Example 85

4.4 VM Image File Content Aware Design 87
4.4.1 Overview 87
4.4.2 VM Image File Content Aware Scheme 89
4.4.3 Metadata Identification 91
4.5 Performance and Overhead Analysis 93
4.6 Evaluation 96
4.6.1 Experimental Setup 96
4.6.2 PCM and NAND Flash Memory Models 98
4.6.3 Results and Discussion 98
4.7 Summary 104

CHAPTER 5. CONCLUSION AND FUTURE WORK 105
5.1 Conclusion 105
5.2 Future Work 106

REFERENCES 108

LIST OF FIGURES

1.1 The Unified Research Framework. 7

2.1 The structure of a traditional NAND flash memory cell. 13
2.2 Illustration on wearing of a flash memory cell caused by (a) the programming operation, and (b) the erase operation. 14
2.3 The structure of a self-healing NAND flash memory storage cell. (a) 3D vision. (b) 2D vision. 15
2.4 A comparison of worn out flash memory and healed flash memory. (a) The worn out flash memory storage cell. (b) The healed flash memory cell. 16
2.5 (a) FTL-based self-healing NAND flash memory storage systems. (b) Flash-file-system-based self-healing NAND flash memory storage systems. 17
2.6 Motivating example. 18
2.7 Dispersed heating scheme. 19
2.8 Illustration on the procedure of lazy heating repair. 25
2.9 The raw bit error rate with different P/E cycles [42]. 26
2.10 The early heating strategy for reliability. 26
2.11 Illustration of the dispersed heating scheme working with NFTL [19]. 28
2.12 Experimental platform. (a) The top layer of our experimental platform. (b) The core development board. 31
2.13 The evaluation framework of DHeating. 32
2.14 The consecutive heating time interval of SWL and DHeating over six applications (part1). 35
2.15 The consecutive heating time interval of SWL and DHeating over six applications (part2). 36
2.16 The lazy heating repair effect with running the $File\ Copy$ application. (a) The baseline scheme without lazy heating repair. (b) The lazy heating repair technique with $60s$ heating period. (c) The lazy heating repair technique with $600s$ heating period. 38
2.17 The consecutive heating time intervals of DHeating over the $File\ Copy$ application with different lifetimes. 40

3.1 The I/O system architecture of Android mobile devices. 46
3.2 Two usage methods of non-volatile memory. 48
3.3 The storage architecture with NVM integration. 49
3.4 The disk cache effect. 50

3.5 Disk cache in the file system. 51
3.6 Data organization in the block device driver. 53
3.7 The data organization with the inter-app and intra-app technologies. 54
3.8 Experimental Android platform. 62
3.9 The I/O characterization with running (a) the WeChat application, and (b) the Facebook application. 65
3.10 The I/O performance by comparing the baseline schemes with the vFlash scheme (the first user). 66
3.11 The I/O performance by comparing the baseline schemes with the vFlash scheme (the second user). 67
3.12 The I/O performance by comparing the baseline schemes with the vFlash scheme (the third user). 67
3.13 Illustration of the I/O energy consumption (the first user). 69
3.14 Illustration of the I/O energy consumption (the second user). 69
3.15 Illustration of the I/O energy consumption (the third user). 70
3.16 The performance degradation caused by the space preparation. 71

4.1 The storage architecture of mobile virtualization. 78
4.2 The qcow2 VM image file format with the ext4 file system. 79
4.3 A file creation procedure in a VM image file. 80
4.4 A file deletion procedure in a VM image file. 80
4.5 A file content increase procedure in a VM image file. 81
4.6 A file content shrink procedure in a VM image file. 82
4.7 The number of the logical block updates when running the facebook application. 83
4.8 A motivational example of I/O optimization in mobile virtualization. 84
4.9 Three possible ways of integrating NVM into virtualization environment. 87
4.10 VM image file content aware scheme. 89
4.11 Two level address mapping in qcow2 image file. 90
4.12 Padding flag strategy to classify metadata from regular data. 93
4.13 The I/O performance with different number of VMs. 101
4.14 Comparison of all write requests and data updates in PCM with different number of VMs. 102

LIST OF TABLES

2.1 The characteristics of the applications. 33
2.2 Average heating time interval. 33
2.3 Extra valid page copies. 37
2.4 Extra block erasures. 37
2.5 A reliability comparison of early heating over different applications with different reduced lifetimes. 41

3.1 The characteristics of NAND flash memory and non-volatile memory [31, 49, 87, 104]. 47
3.2 The description of symbols used in analysis. 59
3.3 The Android applications and their usage. 63

4.1 The characteristics of NAND flash memory and PCM [18, 31, 76, 104]. 76
4.2 The Android applications and their usage. 97
4.3 The I/O requests with different numbers of VMs (part 1). 99
4.4 The I/O requests with different numbers of VMs (part 2). 100
4.5 The I/O energy consumption with different numbers of VMs. 103

CHAPTER 1
INTRODUCTION

NAND-flash-based storage device has been widely used in mobile devices due to its attractive features, such as low cost and high density. However, with the advent of multi-level cell (MLC)/triple level cell (TLC) NAND flash technology, several constraints are exposed, particularly, limited lifetime and lengthy response time. For example, TLC flash memory can only withstand 2,500 program/erase (P/E) cycles, and the program latency is up to $5ms$ [9, 29, 39, 65, 66]. In mobile devices, the lifetime of other components is usually much longer than that of the flash device, such as unlimited read/write cycles of DRAM-based main memory [76]. In addition, the network performance (e.g. 802.11ad with 7Gbps peak throughput) and CPU speed in embedded systems have been greatly improved in the last several years, while the performance of NAND-flash-based storage in embedded systems is stagnant. As a result, flash-based storage has become one of the major lifetime and performance bottlenecks in mobile devices.

In this thesis, we address the short lifetime and lengthy response time issues from several aspects including the integration of the emerging hardware and the cross-layer software management for the optimization. Specifically, we employ two emerging hardware, self-healing NAND flash memory and non-volatile memory, to improve the flash lifetime and performance. These hardware can provide much faster access speed and longer read/write cycles compared with the state-of-the-art flash memory, while also introduces new challenges to well utilize these new hardware. For example, to extend the flash lifetime, researchers at Macronix invent a self-healing flash memory [64], which can cause worn-out NAND flash cells to become reusable and greatly prolong the lifetime of flash memory cells. However, the heating process consumes a substantial amount of power. This means that some fundamental changes are required if existing NAND flash management techniques are to be applied

in self-healing NAND flash memory. In particular, all existing wear-leveling techniques are based on the principle of evenly distributing writes and erases. This causes NAND flash cells tend to wear out in a short time period. Moreover, healing these cells in a concentrated manner may cause power outages in mobile devices.

We first employ self-healing flash memory to improve the flash lifetime and performance. With the self-healing flash memory integration, a new wear-leveling scheme called DHeating (Dispersed Heating) is proposed to solve the concentrated heating problem in self-healing flash memory. In DHeating, rather than evenly distributing writes and erases during a certain time period, write and erase operations are concentrated on a small portion of flash memory cells, so that these cells can be worn-out and healed by heating first. In this way, we can disperse healing to avoid the problem of concentrated power usage caused by heating and eliminate the performance degradation caused by the traditional wear leveling strategies. Furthermore, with the very long lifetime that results from self-healing, we can sacrifice lifetime for reliability. Therefore, we propose an early heating strategy to solve the reliability problem caused by concentrated heating. The idea is to start the healing process earlier by heating NAND flash cells before their expected endurance.

Second, we exploit the novel NVMs (non-volatile memories), such as PCM (Phase Change Memory) and STT-RAM (Spin-Transfer Torque Random Access Memory), to enhance the flash lifetime and performance. In this thesis, we propose a unified NVM/flash architecture to improve the I/O performance. A transparent scheme, vFlash (Virtualized Flash), is also proposed to manage the unified architecture. Within vFlash, inter-app technique is proposed to optimize the application performance by exploiting the historic locality of applications. Since vFlash is on the bottom of the I/O stack, the application features will be lost. Therefore, we also propose a cross-layer technique to transfer the application information from the application layer to the vFlash layer.

Third, mobile virtualization introduces extra layers in software stacks, which leads to performance degradation. Especially, each I/O operation has to pass through several software layers to reach the NAND-flash-based storage systems. Among all the I/O operations, a large

percentage is updating metadata. Frequent updating metadata not only degrades overall I/O performance but also severely reduces flash memory lifetime. In this thesis, we propose a novel I/O optimization technique to identify the metadata of a guest file system which are stored in a VM image file and frequently updated. Then these metadata are stored in a small additional NVM (non-volatile memory) which is faster and more endurable to greatly improve flash memory's performance and lifetime. To the best of our knowledge, this is the first work to identify the file system metadata from regular data in a guest OS VM image file under mobile virtualization.

The rest of this chapter is organized as follows. Section 1.1 presents the related work. Section 1.2 discusses the unified research framework. Section 1.3 summarizes the contributions of this thesis. Finally, Section 1.4 gives the outlines of the thesis.

1.1 Related Work

In this section, we describe the state-of-the art work related to NAND-flash-based storage systems. We briefly introduce these schemes, and compare with representative techniques in detail in respective chapters.

1.1.1 Wear Leveling

The principle in existing wear-leveling techniques is to evenly distribute writes and erases. Basically, data are classified as hot or cold data according to data update times, while physical blocks are divided into old or young blocks. To evenly disperse erasures to all physical blocks, hot and cold data are allocated to young and old physical blocks, respectively. Existing techniques can be further divided into two categories: static and dynamic.

With the dynamic strategy, wear leveling is achieved by reclaiming young blocks or blocks with small P/E times, while with the static strategy cold data are swapped with hot data so cold data cannot stay in young blocks for a long time [14, 15, 19]. In dynamic wear leveling, if a young block is occupied by cold data but no update has occurred for a long time,

it will not be recycled and other blocks will be worn-out by updates of hot data. Therefore, static wear leveling is proposed to overcome the shortage of dynamic wear leveling.

SWL is a typical static wear-leveling strategy proposed in [19]. In this strategy, a BET (Block Erasing Table) is used to record which block has been erased in a pre-determined time frame. In order to save memory space, a one-to-many mapping mode is adopted in SWL. That is, one BET flag can be shared by 2^k physical blocks. If one of the 2^k physical blocks is erased, the corresponding BET flag is set at 1. At the same time, two numbers, f_{cnt} and e_{cnt}, are used to record the number of 1 in the BET and the total number of block erases done since the BET was reset, respectively. When the ratio of f_{cnt} and e_{cnt} is equal to or larger than a threshold T, which means that some level of unevenness has occurred or many erases have been done on a small portion of blocks, SWL will find out those used blocks whose corresponding BET flag is 0, and these blocks will be swapped with old blocks. SWL can effectively achieve wear leveling with a small space overhead. Therefore, in this paper, we compare our method with SWL.

1.1.2 Non-Volatile Memory Integration

Non-volatile memory (NVM) has drawn a lot of attentions due to its attractive features, such as high density, fast access speed, and low leakage power, which makes NVM as a promising candidate to replace DRAM. However, compared with DRAM, PCM consumes more energy than that of DRAM and can only sustain limited program cycles. So, the state-of-the-art work mainly focuses on overcoming these issues for NVMs to be integrated into the main memory systems. Current researches that aim to overcome these two challenging issues are mainly classified into two categories. The first category focuses on reducing the total number of writes to NVM. The second category targets on evenly distributing writes to the whole memory. These two categories are complementary to each other. Dhiman et al. [30] and Park et al. [69] propose hybrid PCM and DRAM main memories. Ferreira et al. [34, 35] also work on hybrid PCM main memories and propose write-back minimization with new cache replacement policies, unnecessary write avoidance, and PCM-aware swap algorithm

for wear-leveling. Shi et al. [79] propose a caching policy to reduce writes on non-volatile main memory. Besides these techniques, Hu et al. [40] propose algorithms to reduce the total number of write through software optimization. These work aims at reducing the total number of writes. In addition, some researches focus on evenly distributing writes. Qureshi et al. [72] propose a Start-Gap technique to effectively conduct wear levelling for PCM. Seong et al. [75] propose dynamic addresses randomization to prevent malicious wear-out and increase durability for PCM. Wu et al. [95] also propose an address re-mapping method to prevent malicious wear-out. Liu et al. [62] propose an application-specific wear levelling method, which gradually changes the mapping of hot region in address space to different part of physical memory.

The state-of-the-art work also explores how to integrate NVM into the storage system to improve the flash performance and lifetime. NAND flash memory has several constraints, such as erase-before-rewrite, limited lifetime, and lengthy garbage collection operation. These constraints seriously affect the flash memory performance. Compared with flash memory, NVM can well compensate the above constraints, benefiting from the high I/O performance, low standby power, and in-place update features. These good features make NVM as a promising solution to effectively improve the performance and enhance the lifetime of the NAND-flash-based storage system. Hyojun Kim and Seongjun Ahn propose a BPLRU scheme to improve the random writes in flash storage [51], and a non-volatile memory based cache policy for solid state drives is proposed to improve the flash performance in [33, 80]. In [70], Park et al. explore the linux kernel to find out the write-intensive segments in virtual address space, and employ the NVM to cache these data. While these work can improve the NAND-flash-based storage performance, a more effective scheme is required to further consider the mobile environment.

1.1.3 Software Managed Flash

Several state-of-the-art work has studied the NAND-flash-based storage performance issue and used software-assisted method to improve the flash performance. In [50], Hyojun Kim

et al. comprehensively study and analyze the storage architecture and the I/O performance of smartphones. They discover that NAND-flash-based storage has become the new performance bottleneck in smartphones. However, their work focuses on exploring the storage problem without providing solid solutions. Wu et al. [96] present a file-system-aware FTL design to identify file system metadata from regular data and propose to use a fine-grained page-level mapping method to manage metadata. The results shown in this work demonstrate promising performance and lifetime improvement in flash system. However, this work can only identify the file system metadata in the host OS file system and ignore the metadata in a guest OS file system. Jian et al. [68] present a software-defined flash (SDF), a hardware/software co-designed storage system to maximally exploit the performance characteristics of flash memory in the context of server cache workloads. SDF exposes individual flash channels to the host software and eliminates space over-provisioning. The host software, given direct access to the raw flash channels of the flash memory, can effectively organize its data and schedule its data access to better realize the SSDs raw performance potential. However, this study focuses on the web server environment without considering the unique features of the mobile devices.

In addition, several state-of-the-art work exploits the file-system-assisted method to address the slow random-write issue, since flash memory's random-write is much less than its sequential write throughput. At the file system level, Min et al. [67] propose a log-structured file system (SFS) to transform random writes in the file system into sequential writes. Other log-based file systems, such as JFFS2 [93] and YAFFS2 [6] are designed by considering NAND flash characteristics and are used widely, especially in mobile and embedded domains.

1.2 The Unified Research Framework

In this section, we present the unified research framework for the proposed techniques. Figure 1.1 illustrates the sketch of the research framework.

In this thesis, we employ the emerging non-volatile techniques to improve the stor-

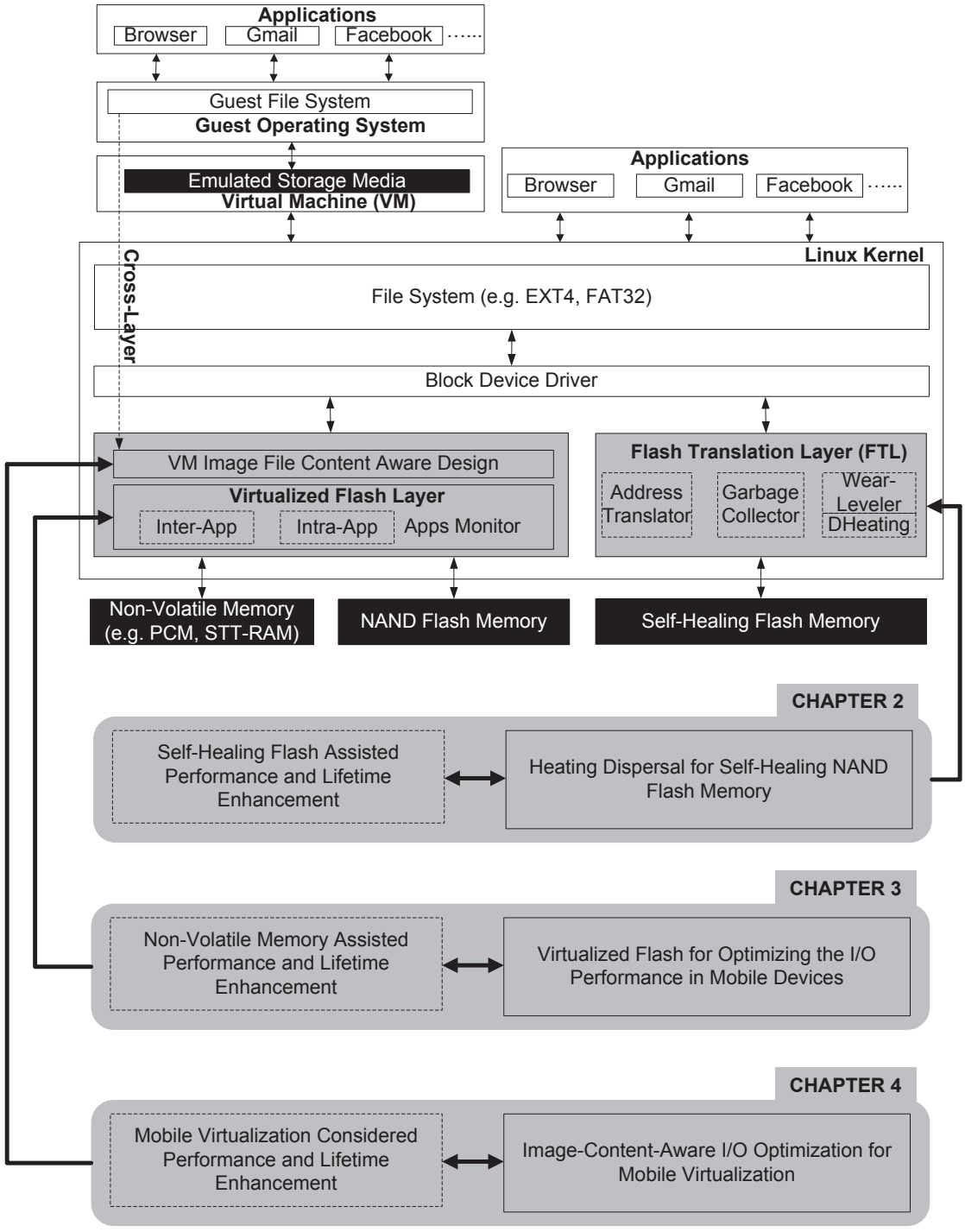


Figure 1.1. The Unified Research Framework.

age lifetime and performance in embedded systems. Three hardware and software codesign techniques are presented in this thesis. As shown in Figure 1.1, we propose to utilize the

self-healing flash memory and integrate the non-volatile memory in the embedded storage systems with the objective of improving the flash lifetime, reducing the wear leveling overhead, and the garbage collection overhead.

- For the first technique, in Chapter 2, we propose a novel wear-leveling scheme, called DHeating, to manage self-heating NAND flash memory. DHeating performs wear leveling by storing frequently updated data, called hot data, in a small number of flash memory cells. By applying this technology to self-healing flash memory, heating operations are dispersed so that only a small portion of the cells will be healed each time, and the time interval for the consecutive heating operations is prolonged. In addition, the heating procedure takes several seconds, which becomes the new performance bottleneck in self-healing flash memory. In order to ease the long time heating effect, we propose a lazy heating repair scheme. The basic idea of the lazy heating repair scheme is to delay the heating operation and to repair during the system idle time period. Furthermore, the flash memory's reliability degrades dramatically with self-healing flash memory reaching its worn out point [10, 36, 42]. In order to enhance the reliability of the self-healing flash memory, we propose an early heating scheme in which we start to heal flash memory cells earlier than their expected endurance. To the best of our knowledge, this is the first work to address the concentrated heating problem through the use of a dispersed heating strategy, to ease the long time heating effect with a lazy heating repair technique, and to enhance the reliability of the self-healing flash memory by adopting an early heating scheme.

- For the second technique, in Chapter 3, we propose vFlash (Virtualized Flash), a transparent cross-layer scheme, to manage heterogeneous NVM and flash memory storage system. vFlash is a software layer that works on the bottom of the software I/O stack and makes use of application I/O features to transparently manage NVM. In vFlash, two techniques, intra-app and inter-app, are integrated to utilize application behaviors to manage NVM. These two techniques are introduced in vFlash based on different considerations. Inter-app technique utilizes the historical locality information of ap-

plications, while intra-app considers the I/O access patterns for each application. At different time periods of each day, a user tends to regularly use some fixed applications, while let the other applications sleep. Based on this historical locality information, inter-app technique can determine which application should be served by NVM at a given time period of each day. At the same time, if too many applications are used at the same period, NVM spaces might be used up. Thus, intra-app can be adopted to decide which application data should be stored in the NVM with the first priority.

- For the third technique, in Chapter 4, we present an image-content-aware scheme to effectively identify both the host and guest metadata in a mobile virtualization platform. In order to achieve this, the proposed image-content-aware scheme identifies guest OS file system metadata according to the I/O request address in the guest OS file system, and uses a padding flag to the virtual machine to trace the metadata. After the metadata are successfully identified, both the guest metadata and host metadata are stored in a small faster and endurable NVM, such as phase change memory. Smart data management techniques are proposed to manage flash memory and the auxiliary NVM. With proposed management techniques, we can greatly eliminate the frequent update effects to the NAND-flash-based storage caused by the file system metadata in a VM image file and in the host file system.

1.3 Contributions

The contributions of this thesis are summarized as follows.

- This is the first work to solve the concentrated heating problem for self-healing NAND flash memory from the wear-leveling perspective. In addition, we propose an early heating strategy to enhance the reliability of the self-healing flash memory in embedded systems, and a lazy heating scheme to eliminate the delay caused by the lengthy heating procedure.

- We take advantage of user behaviors to improve I/O performance in mobile devices. Inter-app and intra-app techniques are crafted to optimize application performance by exploiting historic locality and I/O access patterns of applications. A cross-layer technique is proposed to transfer the application information from the application layer to the vFlash layer to enable optimization.

- We identify the file system metadata from regular data in a guest OS VM image file under mobile virtualization. We propose to utilize a small faster and more endurable NVM to store file system metadata to reduce the flash memory I/O traffic with negligible extra cost. Smart data management techniques are proposed to manage flash memory and the additional NVM.

1.4 Organization

The rest of this thesis is organized as follows.

- In Chapter 2, we address the concentrated heating problem for self-healing NAND flash memory. In the first phase, we present DHeating to solve the concentrated heating problem in self-healing flash memory. In the second phase, we propose early healing and lazy healing to handle some special cases.

- In Chapter 3, we propose a transparent scheme, vFlash (Virtualized Flash), to manage the unified NVM and NAND flash memory architecture.

- In Chapter 4, we present a novel I/O optimization technique to identify the metadata of a guest file system which is stored in a VM image file and frequently updated in the mobile virtualization environment.

- In Chapter 5, we present conclusions and possible future work of research from this thesis.

CHAPTER 2

HEATING DISPERSAL FOR SELF-HEALING NAND FLASH MEMORY

2.1 Overview

NAND flash memory has many advantages such as non-volatility, low power consumption, and good shock resistance. It has been widely used as storage devices in embedded systems. However, NAND flash memory has some constraints, particularly, limited lifetime. For example, multi-level cell (MLC) flash memory, which is the mainstream NAND flash product on the market, can withstand 10,000 program/erase (P/E) cycles; triple-level cell (TLC) flash memory, which is the emerging flash memory product, can withstand only 2,500 P/E cycles [9, 29, 32, 39, 65, 68]. In order to overcome this constraint, Researchers at Macronix recently invented self-healing flash memory, in which worn-out flash memory cells can be rejuvenated by thermal annealing [64].

However, heating a self-healing flash memory cell does consume a substantial amount of power [26]. If a large number of flash memory cells are heated in a short time period, the energy will be exhausted in a battery-driven embedded system such as smartphones. For traditional NAND flash memory, wear-leveling strategies are employed in an attempt to evenly distribute write and erase operations [14, 15, 19, 20, 54, 85]. If such strategies are used to manage self-healing flash memory, it will cause the concentrated heating problem, that is, when all flash memory cells wear out together, healing these cells in a concentrated manner might drain the energy. This chapter focuses on solving this problem.

Several challenging issues should be considered when self-healing flash memory is used in embedded systems. First, the lifetime of self-healing flash memory is prolonged by the heating of worn-out flash memory cells. However, heating does not come out without a

cost, as heating shortens the battery life and a heating operation consumes a large amount of power [26]. As a result, heating should not occur frequently, and only a small portion of flash memory cells should be healed at a time. Second, heated cells can be treated as newborn cells rather than as permanently retired cells. Since the reliability of newborn cells is better than that of nearly worn-out flash memory cells, data should be moved from nearly worn-out cells to newborn cells. As discussed above, previous wear-leveling strategies are not suitable for use on this emerging self-healing NAND flash memory. Thus, it is very important to design a new management strategy for self-healing flash memory.

In this chapter, we first propose a novel wear-leveling scheme, called DHeating, to manage self-heating NAND flash memory. DHeating performs wear leveling by storing frequently updated data, called hot data, in a small number of flash memory cells. By applying this technology to self-healing flash memory, heating operations are dispersed so that only a small portion of the cells will be healed each time, and the time interval for the consecutive heating operations is prolonged. In addition, the heating procedure takes several seconds, which becomes the new performance bottleneck in self-healing flash memory. In order to ease the long time heating effect, we propose a lazy heating repair scheme. The basic idea of the lazy heating repair scheme is to delay the heating operation and to repair during the system idle time period. Furthermore, the flash memory's reliability degrades dramatically with self-healing flash memory reaching its worn out point [10, 36, 42]. In order to enhance the reliability of the self-healing flash memory, we propose an early heating scheme in which we start to heal flash memory cells earlier than their expected endurance. To the best of our knowledge, this is the first work to address the concentrated heating problem through the use of a dispersed heating strategy, to ease the long time heating effect with a lazy heating repair technique, and to enhance the reliability of the self-healing flash memory by adopting an early heating scheme.

We have conducted experiments with various applications based on an embedded system with an ARM11 processor and an 8Gb NAND flash memory chip. The experimental results show that DHeating not only addresses the concentrated heating problem, but also improves the system response time in self-healing flash memory compared with the baseline

scheme. The improvement of the system response time is mainly caused by reducing the swapping of hot and cold data that occurs frequently in previous wear-leveling strategies.

The main contributions of this work are:

- This work addresses the concentrated heating problem for self-healing NAND flash memory from the wear-leveling perspective.
- A lazy heating repair scheme is proposed to alleviate the long time heating effect.
- An early heating strategy is proposed to enhance the reliability of the self-healing flash memory.

The rest of this chapter is organized as follows: Section 2.2 presents the background of this work and our motivation on conducting this work. Section 2.3 introduces our DHeating strategy with performance and overhead analysis. The experimental results are presented and discussed in Section 2.4. In Section 2.5, we summarize this chapter.

2.2 Background and Motivation

2.2.1 NAND Flash Memory and Worn Out

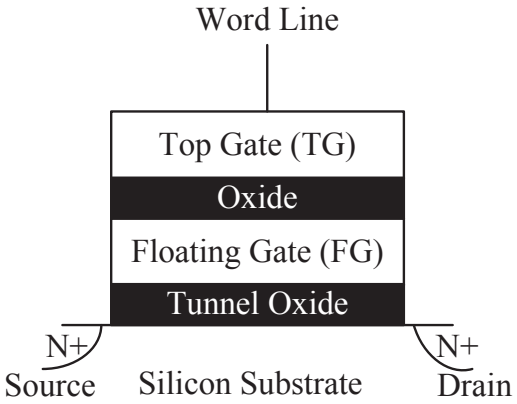


Figure 2.1. The structure of a traditional NAND flash memory cell.

In a traditional NAND flash memory chip, a floating-gate transistor is used as the storage cell as shown in Figure 2.1. A floating-gate transistor is made of one top gate (or control gate), one floating gate, two oxide layers, and the silicon substrate. The top gate, floating gate, and silicon substrate are surrounded by the oxide, which functions as insulation. The oxide layer between a floating gate and the silicon substrate is called *tunnel oxide*. When a high voltage is applied to a word line, which is connected to the top gate of a cell, the electrons from the silicon substrate will traverse the tunnel oxide and be trapped in a floating gate. The number of electrons trapped in a floating gate is used to denote the stored data [81, 88, 100].

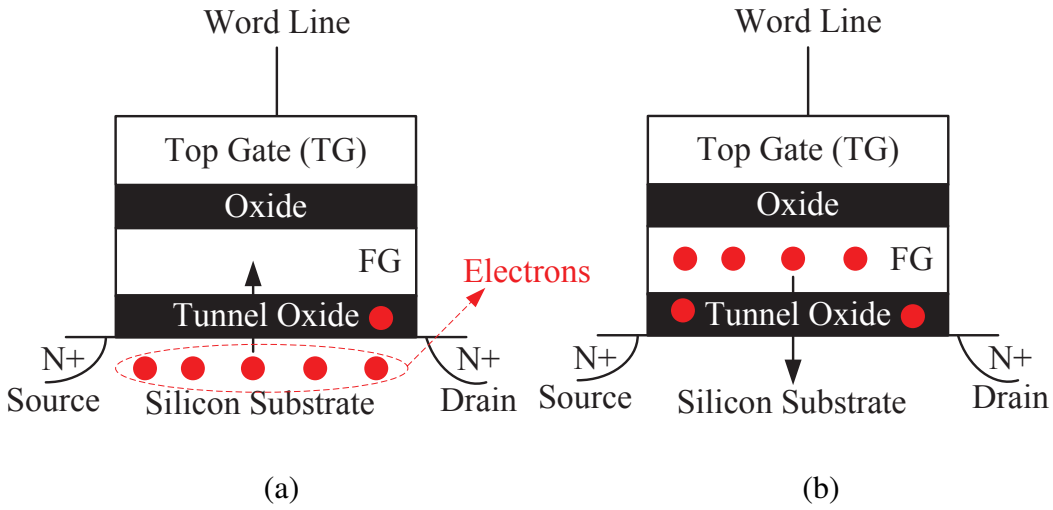
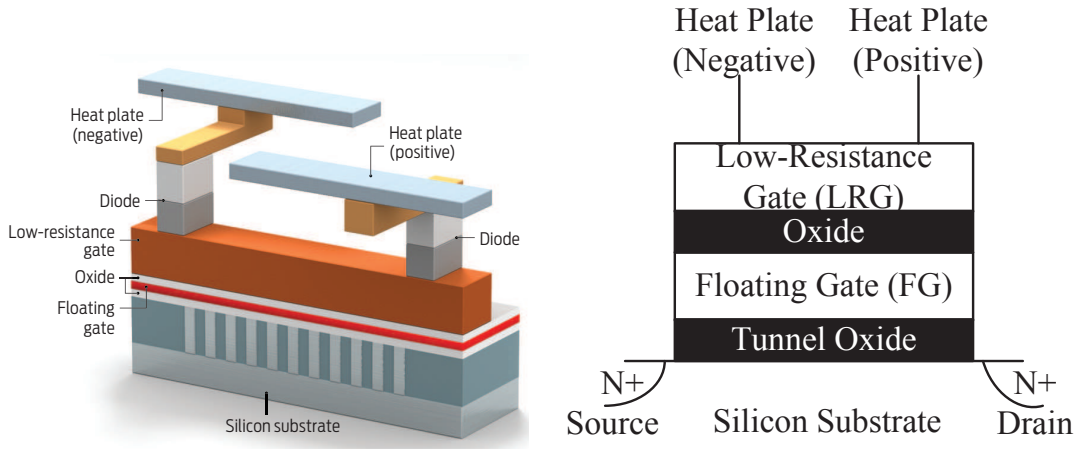


Figure 2.2. Illustration on wearing of a flash memory cell caused by (a) the programming operation, and (b) the erase operation.

The lifetime of a flash memory cell decreases with constant program and erase operations, since repeated P/E cycles can damage the tunnel oxide layer as shown in Figure 2.2. Program operations can cause electrons to traverse through the tunnel oxide layer to the floating gate as shown in Figure 2.2 (a), while erase operations make electrons to traverse through the tunnel oxide layer to the silicon substrate as shown in Figure 2.2 (b). The repeated traverse procedures may cause electrons to be trapped in the tunnel oxide layer. Eventually, the tunnel oxide layer is damaged because of too many electrons being trapped in the tunnel oxide layer.

2.2.2 Self-Healing NAND Flash Memory and Heating Repair



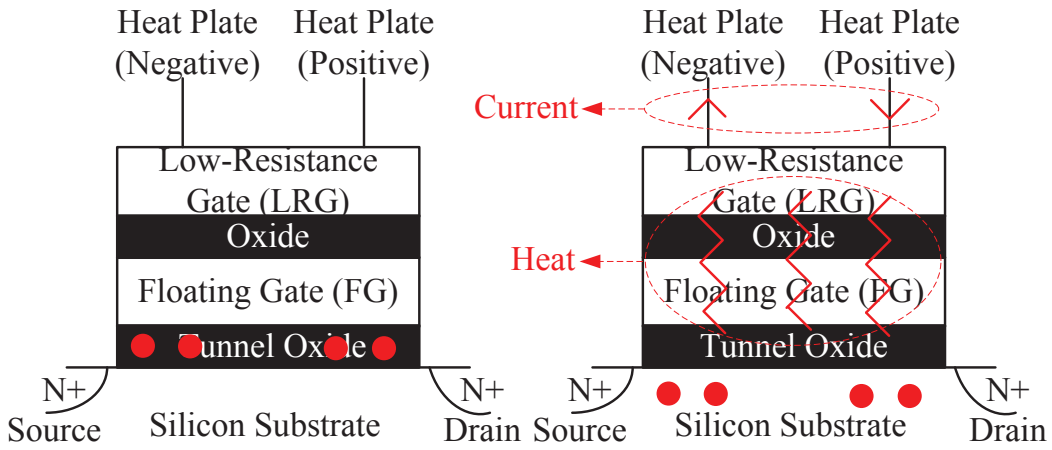
(a) (b)

Figure 2.3. The structure of a self-healing NAND flash memory storage cell. (a) 3D vision. (b) 2D vision.

Several state-of-the-art works [18, 26, 71] discover that high temperature can make electrons to be detrapped from the oxide layer. Therefore, researchers at Macronix modify the traditional NAND flash memory structure and invent self-healing NAND flash memory. The structure of a self-healing flash memory cell is shown in Figure 2.3 (a) and Figure 2.3 (b). In self-healing NAND flash, a word line is modified to become a double-ended structure with one positive heat plate and one negative heat plate; a low-resistance gate is used as the top gate, which enables the current to pass through the gate [26]. The two heat plates are connected to a double-ended word line-one plate on each end.

If a flash memory cell reaches its lifetime as shown in Figure 2.4 (a), a certain level of voltage will be applied to the corresponding heat plates and a high temperature ($>800°C$) will be generated immediately after the current passes through the gate as Figure 2.4 (b) shown. Heating can repair the damaged tunnel oxide of the cell, thus extending the lifetime of the cell.

Since self-healing NAND flash memory extends its lifetime by heating damaged tunnel oxide layers rather than adding new tunnel oxide layers, a flash memory cell will permanently retire after several heating repairs. We assume that a self-healing NAND flash



(a) (b)

Figure 2.4. A comparison of worn out flash memory and healed flash memory. (a) The worn out flash memory storage cell. (b) The healed flash memory cell.

memory cell can be healed at most θ times, and after $i's$ $(1 \leq i \leq \theta)$ healing, it can be used L_i times. The lifetime of a self-healing flash cell can be represented as follows:

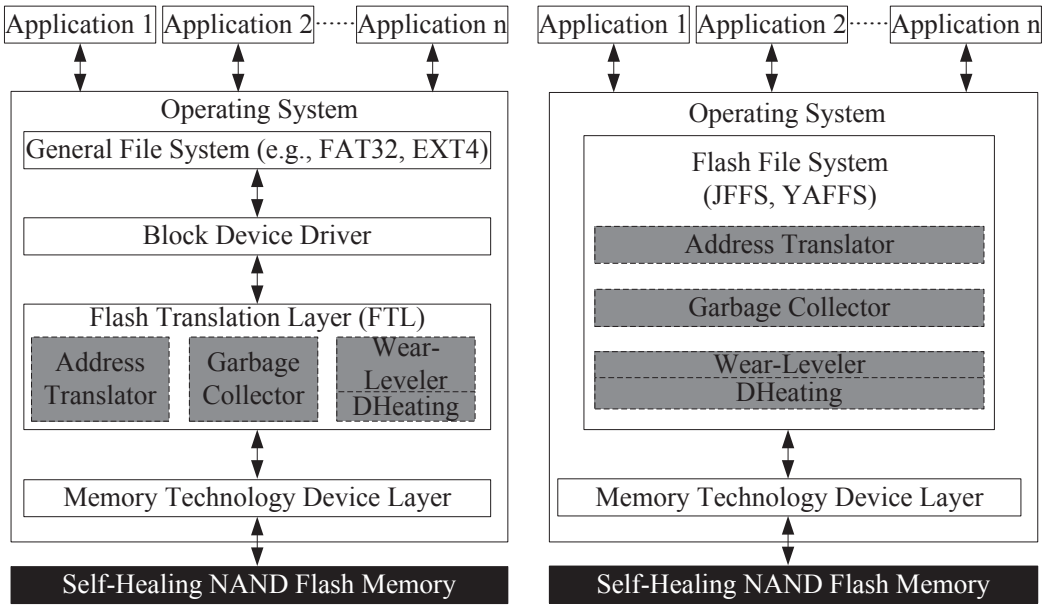
$$Lifetime = \sum_{i=0}^{\theta} L_i \tag{2.1}$$

In Equation (2.1), L_0 denotes the factory lifetime or the lifetime prior to healing, and $L_i(i\,\epsilon[1,\theta])$ denotes the lifetime in each stage.

Suppose that we combine this self-healing technology with TLC NAND flash memory. Typically the factory lifetime of a TLC NAND flash memory chip is 2,500 or L_0 =2,500 [5]. Assume that after each healing, its lifetime is decreased by 10, that is, $L_i - L_{i-1} = 10$ $(1 \leq i \leq 250)$. Based on Equation (2.1), we find that the total lifetime of TLC NAND flash memory is extended to $\sum_{i=0}^{250} L_i$ = (2500+2490+...+10) = 313,750 that is about 125 times longer than that of traditional TLC NAND flash memory.

2.2.3 The Architecture of Self-Healing NAND Flash Memory

In general, there are two methods to make self-healing flash memory usable in embedded systems: *FTL-based flash memory storage systems* and *flash-file-system-based memory stor-*



(a) (b)

Figure 2.5. (a) FTL-based self-healing NAND flash memory storage systems. (b) Flash-file-system-based self-healing NAND flash memory storage systems.

age systems. As shown in Figure 2.5 (a) and Figure 2.5 (b), in these methods the MTD (Memory Technology Device) layer directly operates on a self-healing flash memory chip by providing primitive functions such as read, write, and erase operations; an FTL or a flash-file-system is used to manage NAND flash by handling such issues as out-of-place update, erase before rewrite, and limited lifetime. Both an FTL and a flash-file-system consist of three components: an address translator, a garbage collector and a wear-leveler. The address translator translates addresses between the logical page number (LPN) and the physical page number (PPN). The garbage collector reclaims space by erasing obsolete blocks in which invalidated data exist; the wear-leveler is an optional component that distributes write or erase operations evenly across all blocks, so that the lifetime of a flash memory system can be improved. In this chapter, we mainly focus on solving the wear-leveling problem for self-healing NAND flash memory chips.

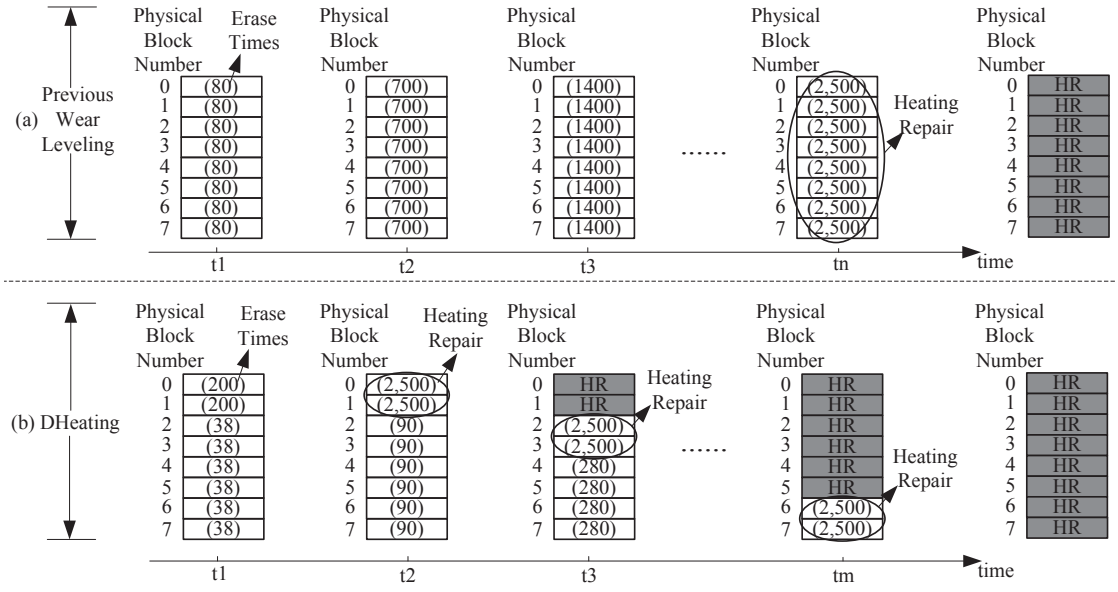


Figure 2.6. Motivating example.

2.2.4 Motivating Example

A motivating example is shown in Figure 2.6. The motivational example described in this section is the typical example in reality and our daily life. Assume that there are eight blocks and one block will carry out a heating operation after 2,500 erasures in a self-healing NAND flash memory chip.

As shown in Figure 2.6 (a), SWL [19] causes all of the blocks to be worn out evenly. As a result, at tn, all eight physical blocks reach their healing threshold of 2,500, and the concentrated heating problem occurs. This can exhaust the energy in the battery within a very short time period and reduce the lifetime of the battery.

Having made this observation, our idea in DHeating is to disperse the heating of the eight blocks over different times. As shown in Figure 2.6 (b), at $t2$, only Block 0 and Block 1 reach the heating bar. At t_m, after Block 6 and Block 7 complete heating operations, all blocks finish the heating operation.

Normally, $tm > tn$, since previous wear-leveling strategies have introduced some valid page copies and block erasure overheads to cause all blocks to wear out evenly. On

the other hand, our proposed DHeating scheme only introduces these overheads when the system carry out a heating operation. As a result, our proposed scheme can further extend the lifetime of self-healing flash memory and improve the performance of the system.

2.3 The DHeating Scheme

In this section, we introduce the DHeating scheme to effectively address the concentrated heating problem. We first give an overview in Section 2.3.1, and then present the dispersed heating strategy and the lazy heating repair scheme in Sections 2.3.2 and 2.3.3, respectively. In Section 2.3.4, we discuss the early heating technique. An example to show how DHeating works with NFTL is presented in Section 2.3.5. Finally, we analyze the performance and overhead in Section 2.3.6.

2.3.1 Overview

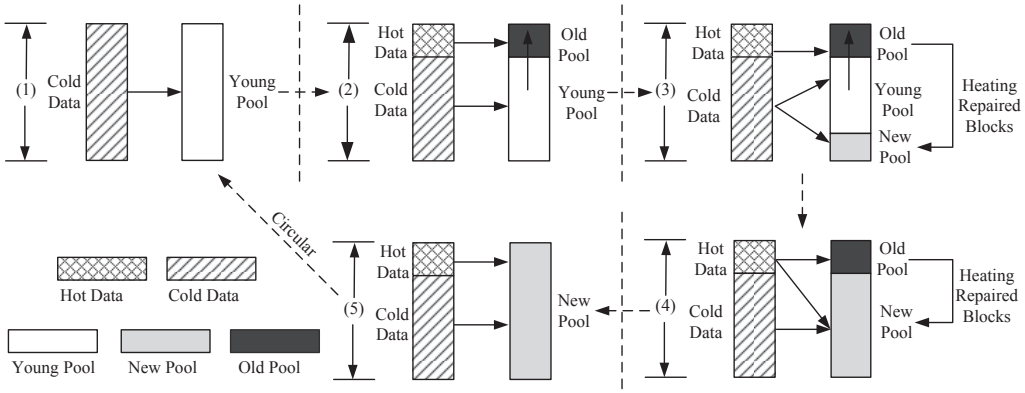


Figure 2.7. Dispersed heating scheme.

The DHeating strategy consists of three schemes: dispersed heating, lazy heating repair and early heating. The dispersed heating scheme is to disperse heating operations to address the concentrated heating problem. The basic idea of the dispersed heating scheme is to intensively wear a small number of flash memory cells at a time. These cells will be worn out quickly and be heated much earlier than that of the other cells. Then, we swap

out these heated flash memory cells, and repeat the above procedure. To this end, data are divided into two categories, hot data and cold data, based on data update frequency. According to block erasure times, physical blocks are classified into old, young, and new physical blocks. Old physical blocks are stored in the old pool, young physical blocks are kept in the young pool, and new physical blocks are stored in the new pool. When a write request is received, the dispersed heating scheme will check which pool the data request belongs to and allocate the corresponding physical blocks. In addition, the lazy heating repair scheme is proposed to address the long time heating issue. The heating procedure takes several seconds and seriously degrades the I/O performance. In order to address this issue, the lazy heating repair scheme delays the heating operation and employs the system idle time to repair. Furthermore, the flash memory's reliability becomes worse with the flash memory cells reaching the expected worn-out time. We propose an early heating strategy to solve the reliability problem. With the extended lifetime provided by self-healing, we can trade some lifetimes for reliability. The idea is to start the healing process earlier than the expected worn-out time. With DHeating, the energy consumed during each occurrence of heating is minimized, the long time heating effect can be eased, and the reliability of self-healing flash memory is enhanced.

2.3.2 Dispersed Heating

In the dispersed heating scheme, all blocks are categorized into three types, namely, young, old, and new blocks and are accordingly put into the young, old and new pools. Meanwhile, we divide the data into cold and hot and put them into blocks in different pools based on the different stages of our scheme.

The dispersed heating scheme has five stages, as shown in Figure 2.7. Stage 1 is the starting point, where cold data are stored in young blocks. In Stage 2, we divide the hot and cold data and the young and old blocks. In Stage 3, old blocks will be heated and moved to the new pool. In Stage 4, no valid blocks exist in the young pool, and all blocks are kept in the old pool and the new pool. In Stage 5, all old blocks are healed and put into the new

pool. We can then repeat the above five stages for a new round. Next, we will present the details of each stage and introduce a key function used in our scheme, the *hot data filter*.

Stage 1 is the initial stage. In Stage 1, all data are cold data and all physical blocks are stored in the young pool.

In Stage 2, hot/cold data and young/old blocks are classified. Based on block erasure times, young blocks will be selected as old blocks and put in the old pool, since we do not want old blocks to be allocated to cold data and young blocks to be allocated to hot data. Then, the blocks in the old pool will continue to be used by the hot data. These blocks will be healed first. A *hot data filter* will be used to identify hot/cold data and young/old blocks, which will be discussed later.

In Stage 3, we start to heal blocks in the old pool, and these healed blocks are moved to the new pool. Then cold data are allocated to blocks in the new and young pools, while hot data continue to be assigned to blocks in the old pool. We move healed blocks to the new pool, because if they are kept in the old pool, they will be used by hot data and may reach their permanent retiring time much earlier than the other blocks. This can lead to very early shrinkage of the capacity of NAND flash memory. When allocating cold data, we will first use blocks in the young pool so blocks in the young pool will become old and move to the old pool, and blocks in the new pool will only be utilized until there are no free blocks in the young pool. In this way, the size of the young pool will decrease while that of the new pool will increase.

We enter Stage 4 after all of the blocks in the young pool are used up. At this stage, there are only the new and old pools, containing new and old blocks, respectively. Different from the above stages, in Stage 4 hot data can be allocated to blocks in both the old and new pools. However, blocks in the new pool are only used when there are no free blocks in the old pool. In the new pool, a free block with the smallest number of erasures is used to hold hot data, and this block will be kept in the new pool after it has been reclaimed. In the old pool, blocks will be healed and moved to the new pool As a result, all blocks will become new at some point in time, and we will enter Stage 5.

Stage 5 is the last stage, and only the new pool exists. The main function of this stage is to clear up history records such as update times and hot or cold data. Then, we will go back to Stage 1 and repeat the above procedures until a flash memory enters the last heating phase, and permanently retires.

Hot data filter: Hot/cold data are identified based on the update frequency. We use T to represent the number of hot data threshold that is calculated as follows:

$$T = (\sum_{i=0}^{n} U_i)/n \tag{2.2}$$

In this equation, n denotes the number of blocks updated by hot and cold data during a time period and U_i $(1 \leq i \leq n)$ is the update times of i's logical block. $U_{max}(= Max_{1 \leq i \leq n} U_i)$ denotes the maximum update times of all cold data. If U_{max} is equal to or larger than the threshold T, a new piece of hot data is detected. Correspondingly, the oldest block or the block with the largest number of erasures in the young pool will be moved to the old pool. The threshold T is calculated based on the average update times during a time period because of the following considerations:

- If T is selected larger than the average update times, some hot data may not be identified. For example, if all update requests are issued from one piece of data, and U_{max} is equal to the average update times, then, this piece of hot data cannot be identified.
- If T is selected smaller than the average update times, too many pieces of cold data may be selected as hot data. This will lead to the selection of many blocks as hot data.

For each logical block, we use a number to store its page update times. Initially, all numbers are set as zero. When one of the numbers achieves its maximum value, the *hot data filter* will be triggered, and all numbers will be reset as zero. It is important to determine how many bits are used to represent the number. If too many bits are used, a large memory space will be consumed and hot data will be detected for a very long period of time; otherwise, the *hot data filter* will be frequently triggered. In our experiments, eight bits are used to represent this number. An example is given below to show how hot data are identified. Assume that

there are eight logical blocks, and eight bits are used to record the number of updates of each logical block. Suppose that the update times of these eight logical blocks are 255, 100, 3, 2, 6, 2, 8, and 8, respectively. When the *hot data filter* is triggered, 255 is the maximum value of an 8-bit unsigned number. Using Equation (2.2), the threshold T can be calculated as (255+100+3+2+6+2+8+8)/8 = 48. If the logical block with the update time 255 has already been chosen as hot data and all other blocks are cold data, then U_{max} is 100 as it is selected from all cold data. Because U_{max} is larger than 48, the *hot data filter* chooses the logical block with the update time 100 as a new piece of hot data.

Algorithm 1: Hot data filter

Input: n: The number of updated blocks

U_{all}: All update times

U_{max}: Maximum update times of cold data

E: The erasure times of physical blocks

P_y: The young pool

P_o: The old pool

Output: HDN: The logical block number identified as the hot data

```
1 T ← U_all/n
2 if U_max ≥ T then
3     HDN ←The logical block number with U_max
4     PB_oldest ← E_max{X|XϵP_y}
5     P_o ← PB_oldest
6     Return HDN
7 else
8     Return NULL to denote no hot data is detected
```

Algorithm 1 describes how the *hot data filter* works. Three input parameters are used in *hot data filter*: n, U_{all} and U_{max}, which denote the number of pieces of updated data, all

update times, and the maximum update times of cold data respectively. n and U_{all} are used to calculate filter threshold T. Then if U_{max} is equal to or larger than the threshold T, a new piece of hot data will be detected and returned, and a physical block with the maximum erase time from the young pool will be moved to the old pool. Otherwise, if U_{max} is smaller than threshold T, no hot data is detected and the algorithm returns $NULL$.

2.3.3 Lazy Heating Repair

The heating operation takes a few seconds, which are much longer than that of an erase operation [18], such as $1.5ms$ block erase time [32]. In addition, a heating operation can block the whole Die response in a flash chip and thus may seriously degrade the system response time. Therefore, the heating operation has become the new performance bottleneck in self-healing flash memory and an effective management method is urgently required. In this section, we propose a lazy heating repair technique to alleviate the long heating effect with the benefits of utilizing system idle time.

When a block reaches the heating point, the block is moved to a heating list instead of being heated immediately. Then, when the system is idle, the blocks in the heating list are selected to repair. By delaying the heating operation and employing the idle time to repair, the lazy heating repair technique can ease the long time heating effect. However, this strategy may introduce a new problem. If too many blocks are accumulated in the heating list, these blocks may be healed in a concentrated manner. In order to address this issue, the lazy heating repair is periodically triggered to clean the heating list. The heating period is adaptively adjusted according to two factors: the number of heating blocks in the heating list and the number of free blocks. If a lot of blocks are waiting to do heating repair in the heating list and a few free blocks are available, the heating repair operation is triggered intensively within a short heating period. On the other hand, a few blocks in the heating list or many free blocks will result in triggering the heating operation infrequently.

Figure 2.8 illustrates the procedure of the lazy heating repair scheme. Assume that the heating period is $\triangle t$ and a block is heated at $t1$. If the system is idle after $\triangle t1 (\triangle t1 < \triangle t)$

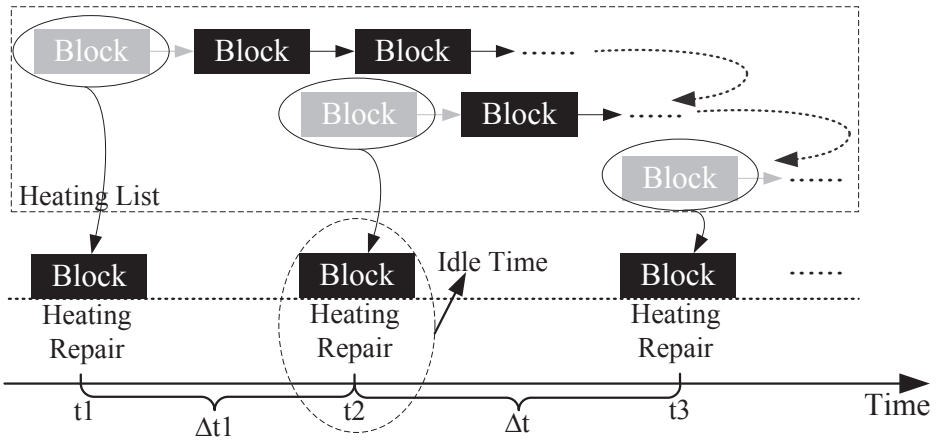


Figure 2.8. Illustration on the procedure of lazy heating repair.

time interval, a block in the heating list will be selected to repair at the idle time, such as $t2$ as shown in Figure 2.8. On the other hand, if the system is always busy during the $\triangle t$ time interval, a block in the heating list is chosen to repair after the $\triangle t$ time interval. As shown in Figure 2.8, the time interval between $t2$ and $t3$ is $\triangle t$ and the system is always busy during this time interval. Then, at $t3$, a block in the heating list is selected to repair, since the elapsed time has reached the heating period $\triangle t$. The adjustment of $\triangle t$ can influence the system performance and number of heating blocks. With the incremental of the heating period $\triangle t$, the lazy heating repair scheme can well utilize the system idle time to do heating and thus further eliminating the performance overhead caused by heating. However, this improvement benefits from keeping more worn-out blocks in the heating list. If only a few free blocks are available to response the write requests and a lot of blocks are kept in the heating list, the blocks in the heating list may be required to be healed in a concentrated manner. This may seriously degrade the system response time. In order to well utilize the lazy heating repair scheme, it is better to dynamically adjust the heating period according to the number of heating blocks in the heating list and the number of free blocks.

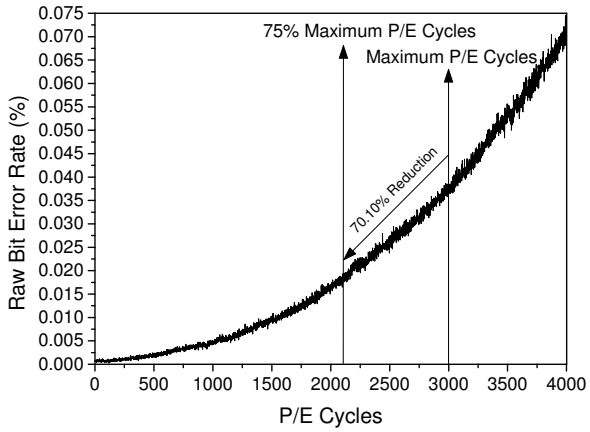


Figure 2.9. The raw bit error rate with different P/E cycles [42].

2.3.4 Early Heating

An early heating strategy is proposed to enhance the reliability for self-healing NAND flash memory. In the dispersed heating scheme, where write and erase operations are on a small portion of flash memory cells, the reliability problem might arise. We propose to utilize an early heating strategy to address this problem. The idea is to start the healing process earlier than the expected worn-out time.

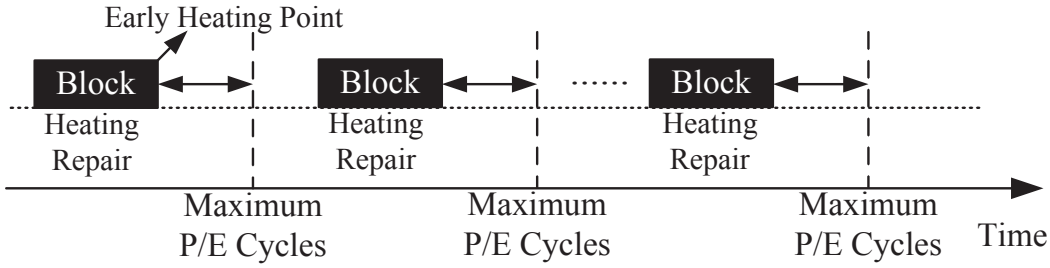


Figure 2.10. The early heating strategy for reliability.

Since the dispersed heating scheme utilizes a small number of flash memory cells in a concentrated manner, the reliability problem may arise. Figure 2.9 shows the raw bit error rate with different P/E cycles. These results are obtained via testing Micron's MLC flash 29F64GCBAAA with the capacity of 64 Gb and the maximum of 3000 P/E cycles. One key

observation form Figure 2.9 is that the raw bit error rate increases dramatically with flash memory cells reaching the expected maximum P/E cycles. If we do healing earlier than the expected maximum P/E cycles as shown in Figure 2.10, the raw bit error rate can be greatly reduced and thus the reliability of self-healing flash memory can be effectively enhanced. For example, the bit error rate can be reduced by 70.10% with the cost of 25% lifetime reduction.

In addition to address the reliability issue, the proposed early heating strategy can help avoid the concentrated heating problem. If a lot of blocks are worn out evenly, the early heating strategy can help avoid the concentrated healing problem by healing a small portion of flash memory cells earlier than the expected heating time. Thus, we can still guarantee only a small portion of flash memory cells are healed each time.

2.3.5 DHeating Working with NFTL

Figure 2.11 illustrates how the DHeating scheme works. For purpose of demonstration, NFTL [19] is selected as the FTL. NFTL uses a block-level address translation mechanism for coarse-grained address translation and is widely used in embedded systems. Note that our scheme is general and can work with other FTLs at block-level, page-level, or hybrid-level. In NFTL, a logical page number (LPN) is divided by the number of pages in a block to obtain its logical block number (LBN) and block offset, where the LBN is the quotient, and the block offset is the remainder of the division. A block-level mapping table maps the LBN into a physical block known as the primary block (PPBN). Each primary block is associated with some additional physical blocks known as replacement blocks (RPBN). A write operation to an LPN is mapped to a page in a primary block, and subsequent update operations to the same LPN are made on the corresponding replacement block with the same block offset.

Figure 2.11(a) shows the first stage of the dispersed heating, where all data update times are 0 and all physical blocks are young. After some data requests are issued, hot data LBN 0 is filtered from cold data with the help of a *hot data filter* (Figure 2.11(b)).

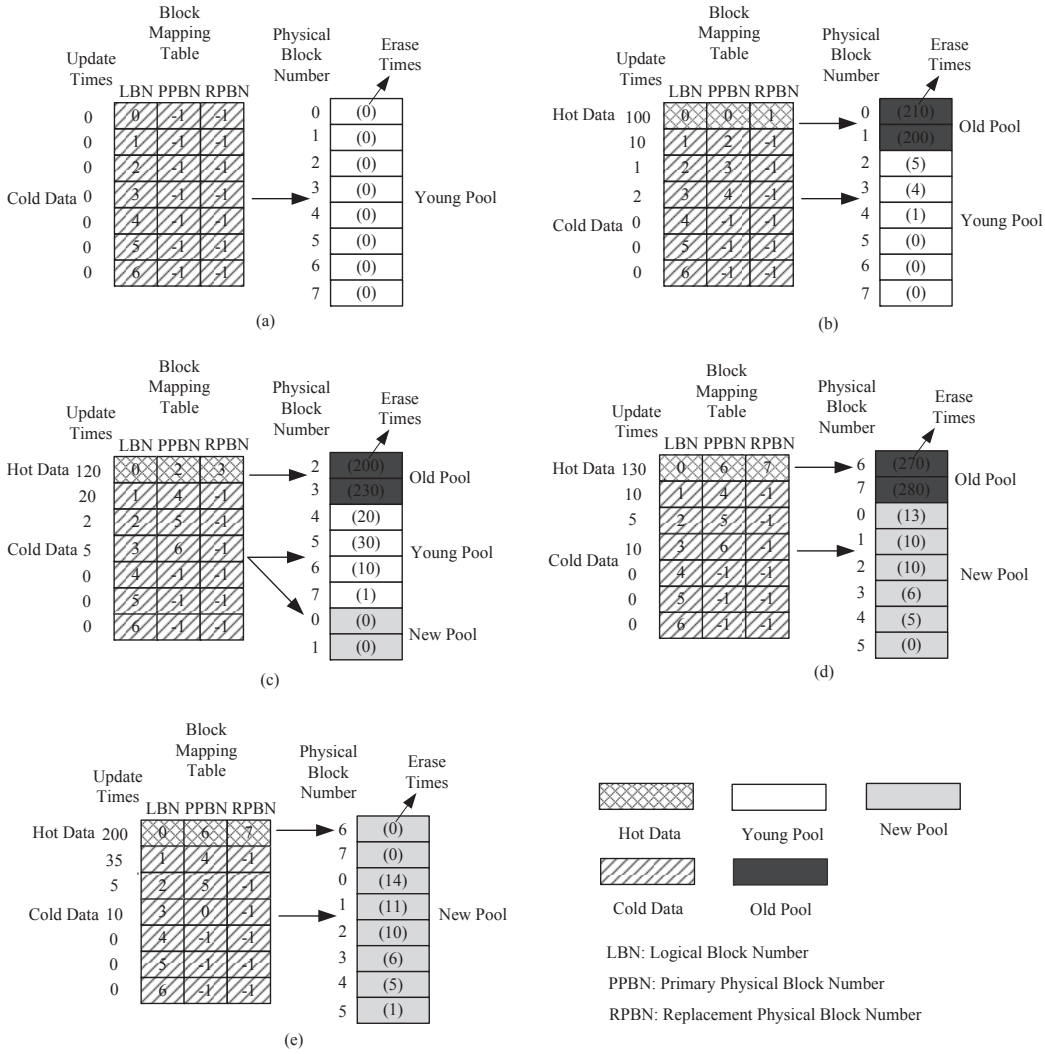


Figure 2.11. Illustration of the dispersed heating scheme working with NFTL [19].

Correspondingly, two old blocks (block 0 and block 1) with the erasure times of 210 and 200 respectively are stored in the old pool.

Figure 2.11(c) shows that block 0 and block 1 have been healed and moved from the old pool to the new pool. Moreover, two old blocks (block 2 and block 3) are moved from old pool to new pool. In Figure 2.11(d), when all the young blocks have been depleted; and only old blocks and new blocks are stored in the old pool and new pool respectively, we enter Stage 4. After the blocks in the old pool have healed, we enter the last stage as shown Figure 2.11(e).

2.3.6 Performance and Overhead Analysis

In this section, we analyze the system performance and overhead of DHeating by comparing it with the SWL scheme [19]. The system response time is an important metric to evaluate the performance of FTLs. It is the time period from the point when an operation is issued to the point when the operation has been completed. The inputs of an FTL are read and write operations. We conduct the analysis for the system response time of read and write operations. The symbols used in this analysis are listed below.

T_{rd}	The time to read one page
T_{wr}	The time to write one page
T_{erase}	The time to erase a block
N_{vpage}	The number of valid pages in one block

The performance overhead is reflected as extra valid page copies and block erasures, and the time is:

$$(T_{rd} + T_{wr}) \times N_{vpage} + T_{erase}. \tag{2.3}$$

Compared with SWL, DHeating only introduces small performance overhead. In SWL, cold data and hot data are swapped very frequently so as to achieve static wear leveling. The frequent swap operations result in heavy performance overhead. SWL will be

triggered to find old and young blocks, and swap valid data between the two different kinds of blocks when unevenness occurs. On the other hand, DHeating only does a data swap when young blocks become old or old blocks become new. Therefore, compared with the SWL scheme, DHeating can reduce valid page copies and block erasures, thereby improving the system response time.

DHeating requires more memory space than that of SWL to record the update times of each block. However, the memory space overhead is negligible since the information recorded is at the block level. For example, the block size of a 32Gb MLC NAND flash [32] memory chip is 512KB. If we use 2 bytes to record the erasure times for each block, only 16KB memory space is required.

With the early heating scheme, DHeating trades lifetime for reliability. However, as shown in the experiments, with only 5% of lifetime overhead, heating can still be dispersed very well. Indeed, the reliability is also improved in DHeating.

2.4 Evaluation

In this section, we present our experimental results with analysis. We compare and evaluate the proposed DHeating scheme with the baseline scheme [19] in terms of four metrics: the consecutive heating time intervals, the extra valid page copies, the lazy heating effect and the early heating effect. The performance evaluation is conducted on an embedded development board with a Samsung ARM11 processor and a 8 Gb NAND flash memory chip.

2.4.1 Experimental Setup

We conducted experiments on a hardware platform. Figure 2.12 (a) shows the top view of our hardware platform. The evaluation platform adopts an ARM11 processor core (Samsung S3C6410 [73]) with ARMv6 architecture. In this platform, the ARM processor core runs at 532MHz; and consists of a 16 KB instruction cache and a 16 KB data cache. The platform adopts the Linux kernel 2.6.38. The core board is equipped with 8 Gb of NAND flash

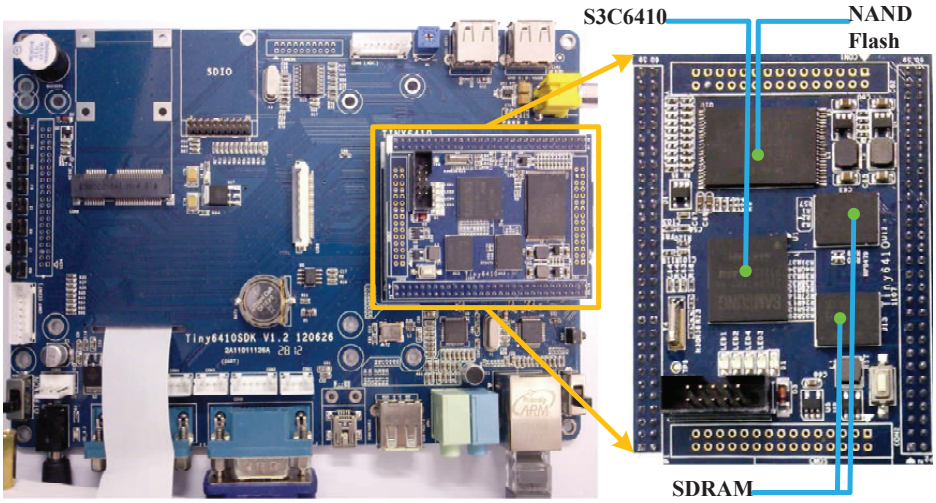


Figure 2.12. Experimental platform. (a) The top layer of our experimental platform. (b) The core development board.

memory and 256 MB of SDRAM. The physical interfaces, such as the RJ45 interface, are designed in the mother board. One pin connector is used to connect the core board with the mother board.

The evaluation framework of our work is shown in Figure 2.13. DHeating is implemented as a block device driver in the Linux kernel 3.5. DHeating functionally works as the wear-lever of the flash translation layer (FTL), and another trace driver module is implemented to trigger DHeating. In our evaluation, the trace driver module is also implemented as a block device driver. FTL can issue read/write operations to the memory technology device (MTD) layer, which can control the NAND flash memory chip. We utilize the universal serial device driver (i.e. a char device driver) to obtain and output the experimental results. For fair comparisons, the same configuration has been adopted for both the baseline scheme and the proposed DHeating scheme. We emulate the process of self-healing and assume that NAND flash memory will perform heating after a given lifetime.

We use real applications as benchmarks to evaluate the effectiveness of DHeating. Since the real environment varies significantly and the same application may generate different I/O requests with different running times even with the same configuration, we collect

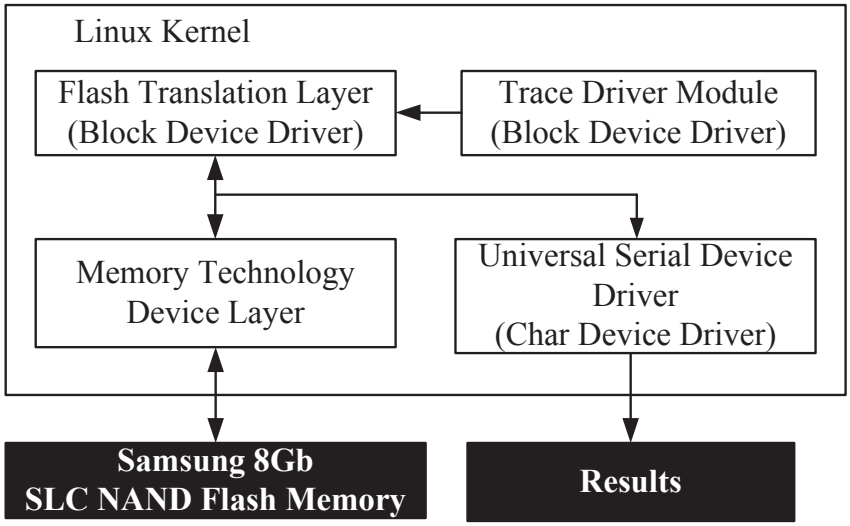


Figure 2.13. The evaluation framework of DHeating.

the I/O requests of real applications and use the collected traces to evaluate the techniques in order to make a fair comparison. The characteristics of traces are shown in Table 2.1. These applications are typical operations in our daily lifetime. They are mainly write dominant, and they can be used to accelerate the evaluation process and evaluate the performance of worn-out flash memory cells for NAND flash memory. We test each benchmark on the evaluation platform. The traces iteratively issue requests to the storage system.

2.4.2 Results and Discussion

In this section, we present the experimental results with analysis. We first present the heating impact of the self-healing flash memory. Then, we present the improvement in performance by comparison of DHeating scheme and the baseline scheme. Finally, we analyze the early heating effect.

We use SWL and DHeating to represent the results obtained from the work in [19] and the proposed DHeating scheme, respectively.

Table 2.1. The characteristics of the applications.

Benchmarks	Numbers of Request	Write (%)	Avg.Arr. Time (ms)	Avg.Req. Size (KB)
File Copy	645,895	90.83	8.19	3.17
Sensor	777,945	92.66	8.12	2.26
Multimedia	790,925	83.10	196.90	48.03
NFS	850,700	89.37	8.40	29.38
SD Card	987,970	94.95	38.30	49.87
FTP	518,575	97.4	72.97	20.05

Table 2.2. Average heating time interval.

Benchmarks	SWL Ave. Heating Time Int.(s)	DHeating Ave.Heating Time Int.(s)	DHeating/SWL
File Copy	162.663	3,143.191	19.323
Sensor	159.070	3,789.925	23.825
Multimedia	166.572	3,864.397	23.199
NFS	185.844	4,152.161	22.342
SD Card	176.371	4,831.884	27.395
FTP	169.512	2,514.422	14.833

Heating Dispersal

Table 2.2 shows the average consecutive heating intervals of two physical blocks. The consecutive heating interval denotes the frequency of heating. Therefore, the longer the consec-

utive heating interval that the NAND flash memory experiences, the better its performance and lifetime. Judging from the experimental results, our DHeating scheme can delay heating by up to 24 times compared to SWL. This shows the effectiveness of DHeating in maximizing the consecutive heating interval.

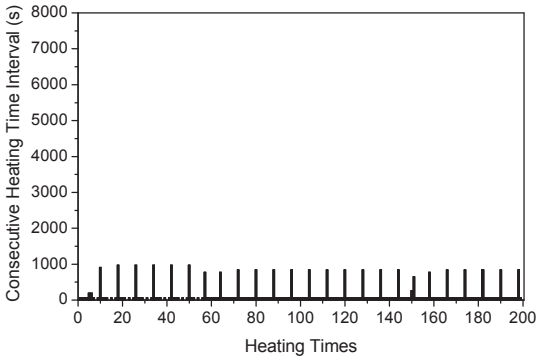
Figure 2.14 and Figure 2.15 present the experimental results of the consecutive heating time intervals of SWL and DHeating. From the results, we can see that the consecutive heating time interval for SWL is very short, and that our scheme can delay the consecutive heating time interval for much longer thant SWL can. In the evaluation, we assume that self-healing flash memory will trigger heating for every 100 program/erase (P/E) cycles.

Taking the benchmark *sensor* as an example, SWL finishes the 200th block heating in $31,655$ seconds, while our scheme finishes the 200th heating in $754,195$ seconds , which is 23 times longer. As the lifetime increases, DHeating will also significantly increase in improvement over SWL.

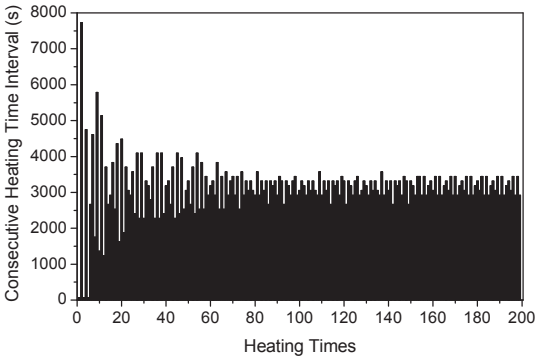
Performance Improvement

Previous wear-leveling schemes basically relied on the swapping of hot and cold data to balance the erase counts [14, 15, 20]; therefore, some system performance had to be sacrificed, such as extra valid page copies and extra block erasures. In our technique, we allocate hot data to old blocks and simply swap the data when young blocks become old blocks or old blocks become new blocks. Therefore, DHeating can incur much less extra overhead compared to previous wear-leveling schemes.

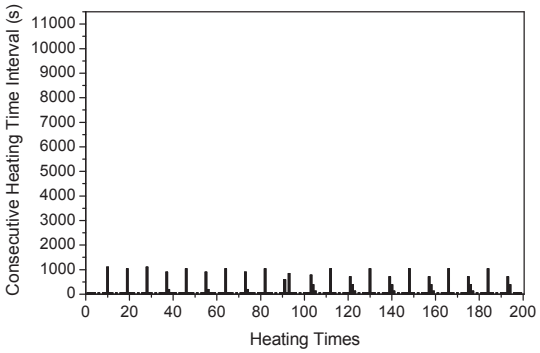
The extra valid page copies and the extra number of block erase counts are compared in Table 2.3 and Table 2.4, respectively. The results of the experiment show that DHeating performs significantly better than SWL in reducing the extra valid page copies and the extra number of block erase counts. Since the extra number of valid page copy operations will issue more write operations to blocks containing free pages, more erase operations will be incurred and the lifetime of the NAND flash memory will be shortened. DHeating can effectively reduce the number of extra valid page copy operations and the ex-



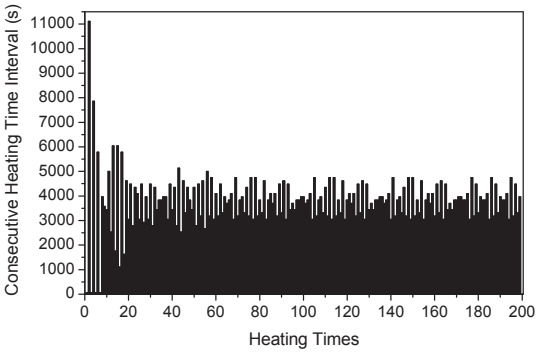
(a) SWL/Local File Copy.



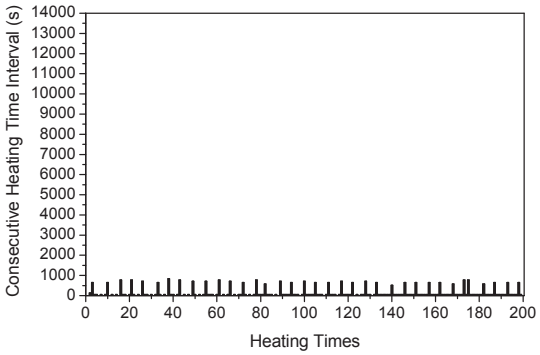
(b) DHeating/Local File Copy.



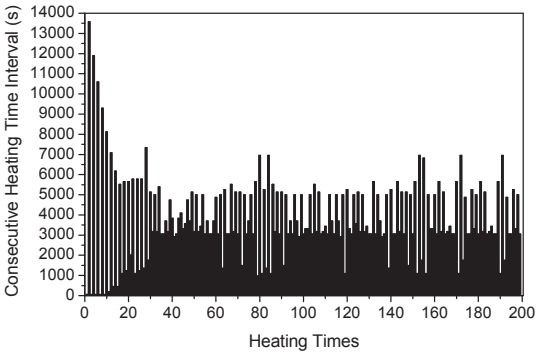
(c) SWL/Sensor.



(d) DHeating/Sensor.

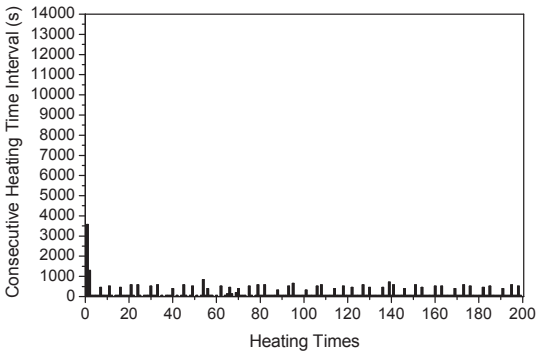


(e) SWL/Multimedia.

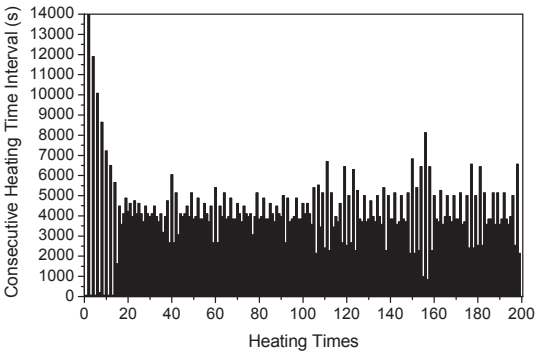


(f) DHeating/Multimedia.

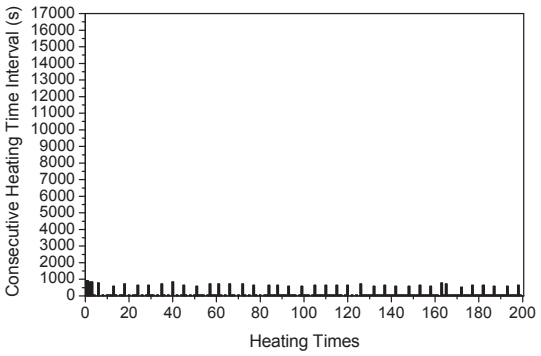
Figure 2.14. The consecutive heating time interval of SWL and DHeating over six applications (part1).



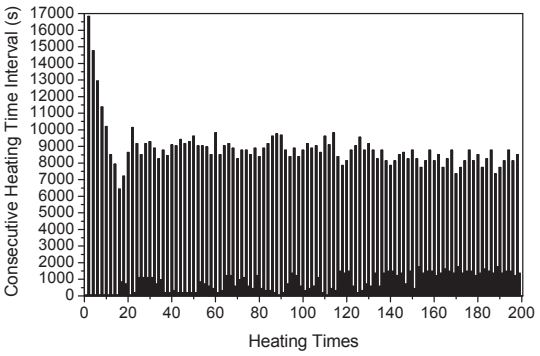
(g) SWL/NFS.



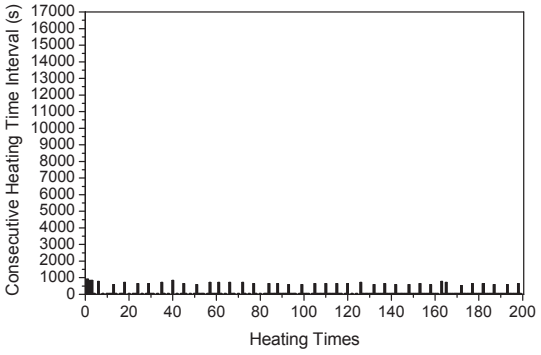
(h) DHeating/NFS.



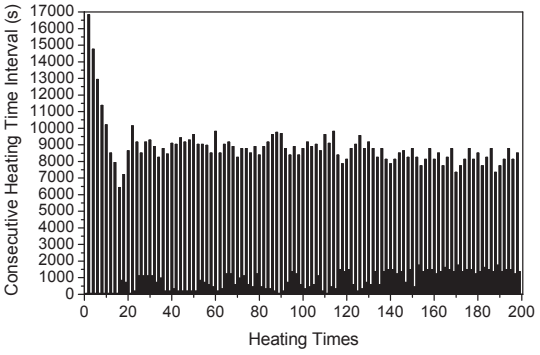
(i) SWL/SD Card.



(j) DHeating/SD Card.



(k) SWL/FTP.



(l) DHeating/FTP.

Figure 2.15. The consecutive heating time interval of SWL and DHeating over six applications (part2).

Table 2.3. Extra valid page copies.

Benchmarks	SWL	DHeating	DHeating over SWL (%)
File Copy	158,720	9,344	94.11
Sensor	209,920	10,496	95.00
Multimedia	158,720	7,936	95.00
NFS	206,036	9,472	95.40
SD Card	215,096	8,640	95.98
FTP	139,202	10,688	92.32

Table 2.4. Extra block erasures.

Benchmarks	SWL	DHeating	DHeating over SWL (%)
File Copy	3,262	192	94. 11
Sensor	3,800	190	95.00
Multimedia	3,600	180	95.00
NFS	4,010	184	95.41
SD Card	4,508	181	95.99
FTP	2,394	183	92.36

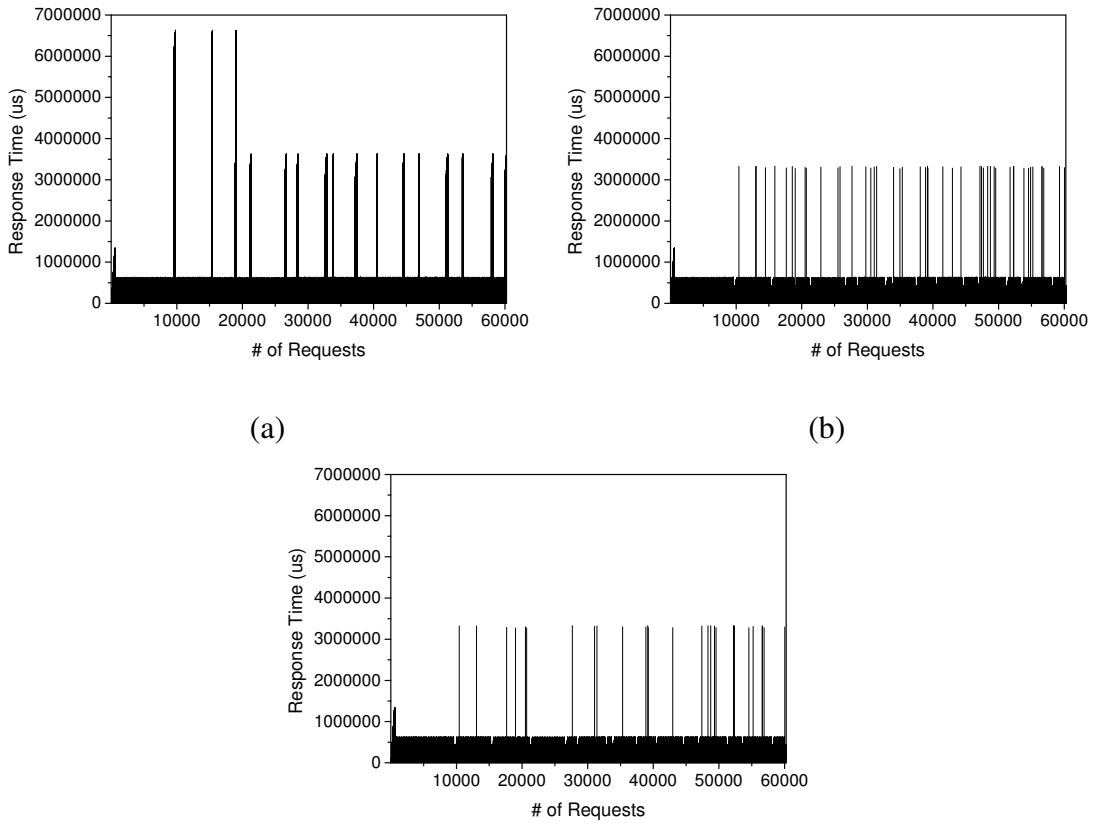


Figure 2.16. The lazy heating repair effect with running the $File\ Copy$ application. (a) The baseline scheme without lazy heating repair. (b) The lazy heating repair technique with $60s$ heating period. (c) The lazy heating repair technique with $600s$ heating period.

tra number of block erase counts, which are beneficial to a NAND flash memory storage system. To better understand the benefits of the proposed scheme, we further analyze the performance improvement. The flash page read and write latencies are $25us$ and $500us$, respectively, and the block erasure latency is $1.5ms$ [31, 49, 87, 104]. Taking benchmark *File Copy* as an example, by using Equation (2.3), we can calculate the performance improvement is $(25 + 500) \times 149,376 + 1,500 \times 3,070 = 830,274,000us$.

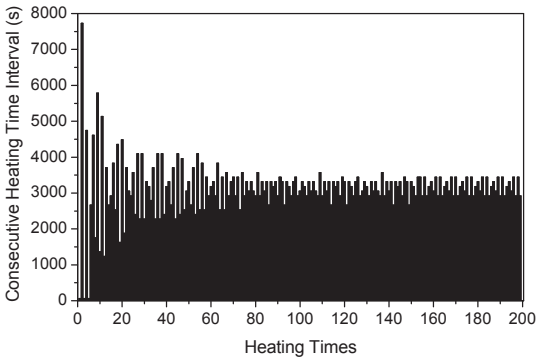
Lazy Heating Repair Effect

Lazy heating repair is proposed to eliminate the long heating time effect. In this experiment, the heating time is configured to $3s$. In order to study the heating period effect, the heating period is configured to $60s$ and $600s$, respectively. The system idle time is detected based on the pending requests. If the pending request queue is empty, the system is idle.

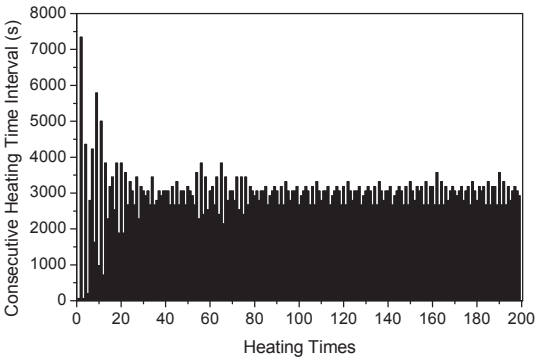
Figure 2.16 illustrates the lazy heating repair effect. The dispersed heating scheme without the lazy heating repair technique is adopted as the baseline scheme. In the baseline scheme, the self-healing flash memory heats the worn-out cells immediately. As a result, the response time is greatly degraded as shown in Figure 2.16 (a), such as $6632964us$ in the worst case. By employing the lazy heating scheme, the system response time can be effectively improved with the benefits of utilizing the system idle time as shown in Figure 2.16 (b) and Figure 2.16 (c). The benefits of the lazy heating repair scheme are reflected into two aspects, the worst case response time and the total heating overhead. Compared with the baseline scheme, the worst case response time is reduced by 49.77% and the total heating time overhead is improved by 98.84% on average. In addition, with the heating period prolonged, such as from $60s$ to $600s$, the lazy heating repair scheme works better and the improvement is enhanced by 43.49%. However, this improvement benefits from keeping more worn-out blocks in the heating list. If only a few free blocks are available to response the write requests and a lot of blocks are kept in the heating list, the blocks in the heating list may be required to be healed in a concentrated manner. This may seriously degrade the system response time. In order to well utilize the lazy heating repair scheme, it is better to dynamically adjust the heating period according to the number of heating blocks in the heating list and the number of free blocks.

Early Heating for Reliability

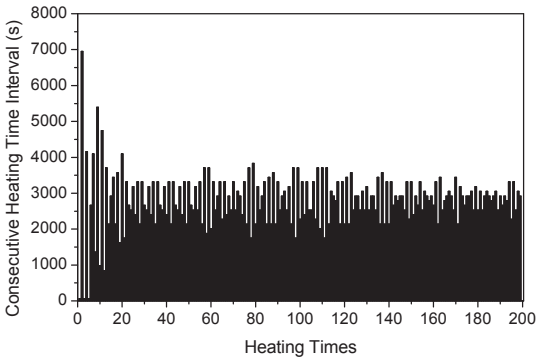
Early heating strategy can enhance the reliability at the expense of trading some flash memory lifetimes. With heating repair started earlier than the expected maximum P/E cycles, the reliability of flash memory can be effectively enhanced. For example, the bit error rate can



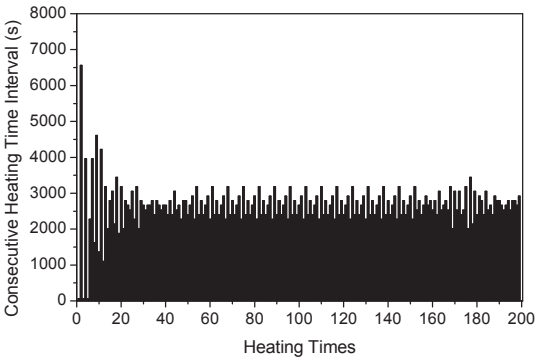
(a) Local File Copy/100%Lifetime.



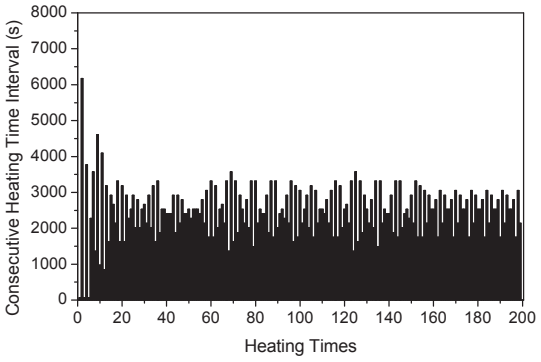
(b) Local File Copy/95%Lifetime.



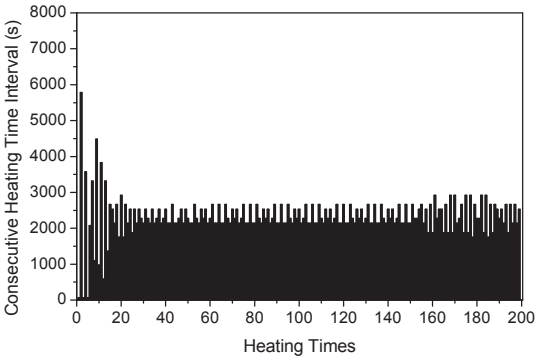
(c) Local File Copy/90%Lifetime.



(d) Local File Copy/85%Lifetime.



(e) Local File Copy/80%Lifetime.



(f) Local File Copy/75%Lifetime.

Figure 2.17. The consecutive heating time intervals of DHeating over the $File\ Copy$ application with different lifetimes.

be reduced by 70.10% with the cost of 25% lifetime reduction [42].

Table 2.5. A reliability comparison of early heating over different applications with different reduced lifetimes.

Benchmarks	DHeating 100% Life Time Ave. Heat. Time(s)	DHeating 95% Life Time Ave. Heat. Time(s)	Ave. Early Heat. Diff. (s)
File Copy	3,143.191	2981.181	162.010
Sensor	3,789.924	3,596.558	193.366
Multimedia	3,864.396	3,663.518	200.878
NFS	4,152.160	3,931.683	220.477
SD Card	4,831.884	4,569.271	262.613
FTP	2,514.422	2,383.442	130.980

Table 2.5 shows the experimental results for early heating with different lifetimes. It can be seen that, using 95% as the heating threshold, early heating can work very well and the average heating time interval only decreases by an average of about 5%. Figure 2.17 illustrates how different threshold values influence the consecutive heating time intervals of the application *File Copy*. With the heating threshold decrease, the average heating interval decreases accordingly. However, the proposed scheme can still work well with the decrease in the lifetime of the flash memory, and the reliability of the self-healing flash memory is enhanced.

2.5 Summary

In this chapter, we propose a scheme, called DHeating, to solve the concentrated heating problem and overcome the constraints of self-healing NAND flash memory. DHeating consists of three techniques: the dispersed heating technique, the lazy heating repair technique,

and the early heating technique. These three techniques are proposed based on different considerations and interact with each other. The dispersed heating technique is designed to avoid the concentrated heating problem. The lazy heating repair technique can address the long time heating issue and the early heating technique is proposed to enhance the reliability of self-healing flash memory. We conduct experiments on a set of representative I/O workloads collected from the embedded development board. The experimental results show that our proposed scheme not only solves the concentrated heating problem for NAND flash memory, but also improves the system response time and enhances the reliability of the self-healing flash memory.

CHAPTER 3

VIRTUALIZED FLASH FOR OPTIMIZING THE I/O PERFORMANCE IN MOBILE DEVICES

3.1 Introduction

Mobile devices have been an integral part of our daily life. Most of applications in mobile devices need to transfer data from the internet and buffer the content in local storage [59,83]. Over a long period, most of efforts in improving system I/O performance have been focusing on improving the network performance. Emerging wireless technologies, such as 802.11ad (7Gbps peak throughput), significantly improve the network throughput of mobile devices [50]. Therefore, local storage I/O is becoming one of the major performance bottlenecks in mobile devices [47, 50]. What is worse, the access speed of flash memory will become slower due to the introduction of MLC (multi-level cell) technology and high reliability error correction techniques [44, 77]. Therefore, it is more and more difficult to satisfy I/O demand with pure flash memory based storage in mobile devices.

Newly emerged NVM (non-volatile memory), such as PCM and STT-RAM [23, 97], provides a promising solution to improve the I/O performance for mobile devices, since it can provide fast and high-throughput memory operations. By placing fast NVMs along side flash memory at the bottom of the I/O stack and storing data in NVMs, the I/O performance can be improved. To manage this kind of heterogeneous storage system, an intuitive approach is to identify frequently accessed data and store them into NVM, while less accessed data are stored into flash memory. This simple approach may work for desktop or server applications. However it does not yield satisfiable performance for mobile devices. This is because the applications in mobile devices are switched much more frequently than those in desktops

or servers. While an application is accessing hot data in one moment, another application might be launched in the next moment and access other data. Therefore, we need a novel management method that is tailored for mobile devices.

One key observation here is that mobile device user behaviors are relatively repetitive everyday. Therefore, if we can exploit historical user behavior information to predict future application usage and relocate data before the application is launched, the system performance and the user experience can be greatly improved. In this chapter, we propose vFlash (Virtualized Flash), a transparent cross-layer scheme, to manage heterogeneous NVM and flash memory storage system. vFlash is a software layer that works on the bottom of the software I/O stack and makes use of application I/O features to transparently manage NVM.

In vFlash, two techniques, intra-app and inter-app, are integrated to utilize application behaviors to manage NVM. These two techniques are introduced in vFlash based on different considerations. Inter-app technique utilizes the historical locality information of applications, while intra-app considers the I/O access patterns for each application. At different time periods of each day, a user tends to regularly use some fixed applications, while let the other applications sleep. Based on this historical locality information, inter-app technique can determine which application should be served by NVM at a given time period of each day. At the same time, if too many applications are used at the same period, NVM spaces might be used up. Thus, intra-app can be adopted to decide which application data should be stored in the NVM with the first priority. The main contributions of this chapter include:

- To the best of our knowledge, this is the first work to take the advantage of user behaviors to improve the I/O performance in mobile devices.
- Inter-app and intra-app techniques are crafted to optimize application performance by exploiting the historical locality and I/O access patterns of applications.
- A cross-layer technique is proposed to transfer the application information from the application layer to the vFlash layer to enable optimization.

Experiments are conducted with various Android applications on an Android platform with an ARM Cortex-A9 processor and a 64Gb NAND flash memory chip. The proposed vFlash scheme is implemented beneath the block device driver in the Linux kernel. The experimental results show that vFlash can improve the I/O performance in Android mobile devices by more than 2.86 times compared with the stock Android 4.2 system.

The rest of this chapter is organized as follows: Section 3.2 presents the background of this work and our motivation for conducting this work. Section 3.3 introduces our vFlash strategies. The performance and overhead introduced by the proposed scheme are analyzed in Section 3.4. The experimental results are presented and discussed in Section 3.5. In Section 3.6, we summarize this chapter.

3.2 Background

In this section, we first present the I/O system architecture of a typical Android mobile device in Section 3.2.1. Then, we introduce the I/O performance bottleneck in Section 3.2.2 and two challenges of integrating NVMs in the storage system of mobile devices in Section 3.2.3, respectively.

3.2.1 The Android I/O System Architecture

We use the popular used Android system to study the I/O storage architecture in mobile devices. As shown in Figure 3.1, Android applications run on the Linux kernel, and each application runs with a Dalvik VM, which is a virtual machine running in Android mobile devices. Dalvik VM converts the Java bytecode to local host executable binary code [24,47]. In the Linux kernel, I/O requests issued from applications are passed through three I/O layers to the flash device, including the file system layer with disk cache, block device driver layer, and flash device driver [63, 106].

The software layers in the I/O stack have been well designed to handle the I/O requests, and the performance in the I/O stack is confined to NAND flash memory. In this

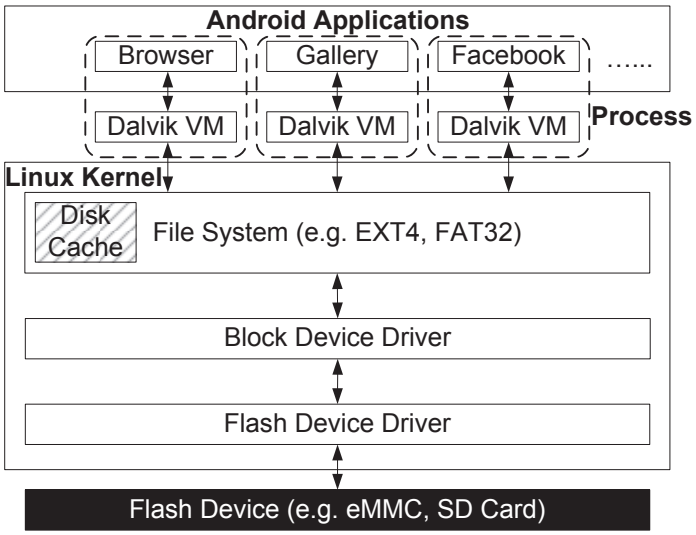


Figure 3.1. The I/O system architecture of Android mobile devices.

chapter, we focus on improving the flash device performance. In order to improve the flash memory performance, we propose to add small faster but expensive NVM to the I/O storage system. However, in order to do so, we need to transparently integrate NVM into the existing system such that all existing applications do not need to be modified.

3.2.2 I/O Bottleneck and Non-Volatile Memory

In mobile devices, NAND flash memory has been prevalently used as the secondary storage. However, with the rapid improvement of the computation and communication speed, NAND-flash-based I/O storage has become the new performance bottleneck, and thus seriously degrades the application performance in mobile devices. NAND flash memory has several constraints, such as erase-before-rewrite, limited lifetime, and long time overhead for garbage collection [17, 22, 28, 54, 86]. These constraints seriously affect the flash memory performance. Compared with flash memory, newly emerged NVM, such as PCM and STT-RAM, can well compensate the above constraints, benefiting from the high I/O performance, low standby power, and in-place update features [11, 97]. These good features make NVM a promising solution to effectively improve the I/O performance for NAND flash memory, and

thus enhance the performance of mobile device applications.

Table 3.1. The characteristics of NAND flash memory and non-volatile memory [31, 49, 87, 104].

Attributes	**NAND**	**PCM**	**STT-RAM**
Non-Volatility	Yes	Yes	Yes
Byte Addressability	No	Yes	Yes
Bit Alterability	No	Yes	Yes
Read Latency	$\sim$ 25 us	$\sim$50 ns	$\sim$ 1 ns
Write Latency	$\sim$ 500 us	$\sim$150 ns	$\sim$ 10 ns
Erase Latency	$\sim$ 1.5 ms	No	No
Endurance	$10^4 - 10^5$	$10^6 - 10^8$	$> 10^{15}$

Table 3.1 summarizes the characteristics of NAND flash memory and non-volatile memory. As shown in Table 3.1, non-volatile memory and NAND flash memory both have the non-volatile feature. However, non-volatile memory also suppports the byte addressability and bit alterability, which are not available in NAND flash memory. These two abilities determine that non-volatile memory can do in-place update, while NAND flash memory should do long time erase operations before update. Also, non-volatile memory presents much shorter read/write latency compared with NAND flash memory, even though its byte addressability has already made it superior to NAND flash on handling frequent data updates. Flash memory has limited endurance, which can only sustain $10^4 - 10^5$ writes before a failure occurs [13, 48, 97]. Compared with flash memory, non-volatile memory has much better endurance than that of NAND flash memory, such as more than 10^{15} for STT-RAM. Therefore, non-volatile memory is rapidly developed as a promising candidate for memory and/or storage in the design of embedded computing systems.

3.2.3 Transparent Integration

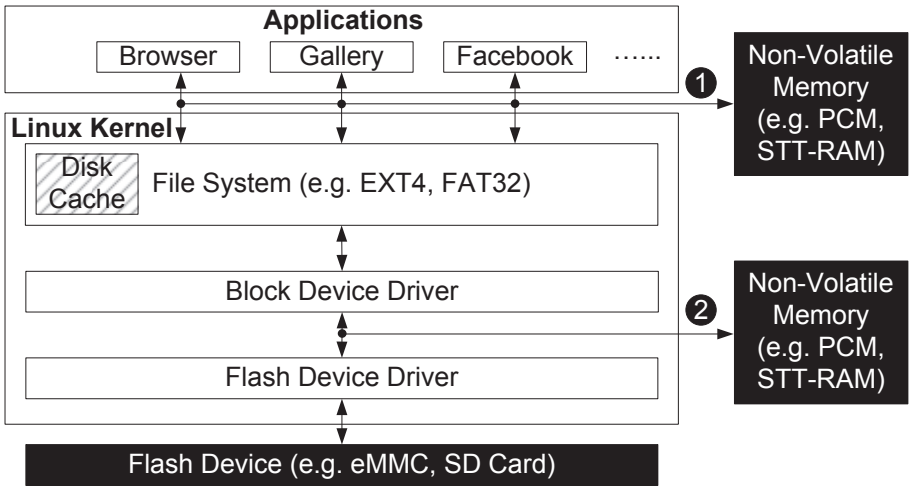


Figure 3.2. Two usage methods of non-volatile memory.

In general, two possible methods can be explored to improve the I/O performance by integrating NVM in mobile devices as shown in Figure 3.2. In the first method, NVM is exposed to applications, and can be directly operated by applications. This method usually involves modifications of the application itself, which is expensive or even impossible. In mobile devices, I/O operations are represented as read or write operations. If we use this method to manage NVM, at least all write operations need to be modified. For example, when an application issues a write request, it should determine the storing media, flash memory or NVM.

Therefore, in order to transparently manage NVM, it is more desirable to place NVM at the bottom of the I/O stack, lying beneath the standard block device driver as shown in Figure 3.2. Compared with the first usage method, the second management method is transparent to upper layers in the I/O stack. By using this method, we can effectively avoid application modifications. It is also the choice of this work. After NVM is integrated into the system, we need a software layer to manage the underlying heterogeneous storage system, and provide transparent support for upper layers. We will discuss the details of the proposed vFlash scheme which will achieve the desired goals.

3.3 vFlash Design

In this section, we introduce the vFlash scheme which can effectively improve the I/O performance in mobile devices. We first provide an overview in Section 3.3.1, and then present the detailed optimizations in Sections 3.3.2 and 3.3.3. An adaptive space preparation scheme is presented in Section 3.3.4.

3.3.1 Overview

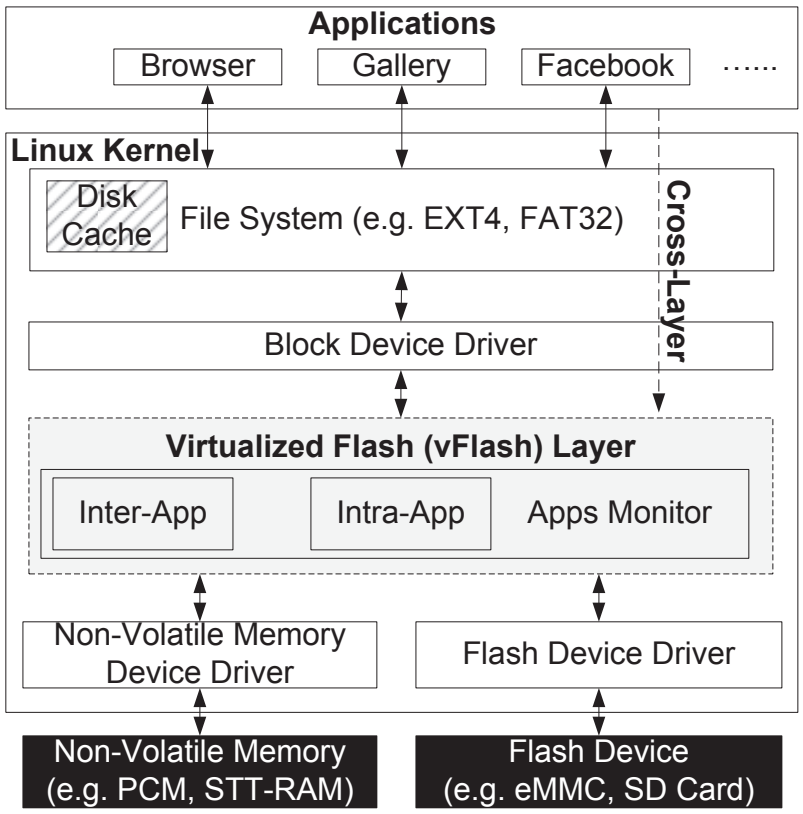


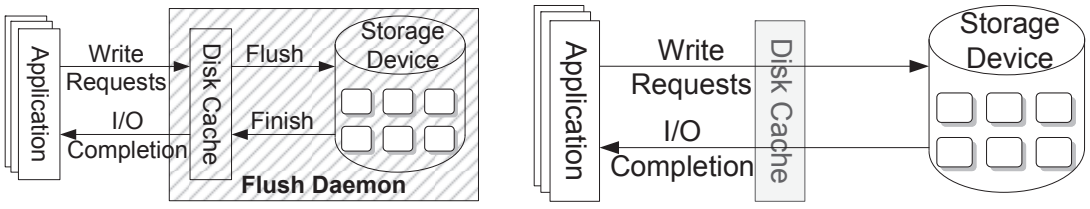
Figure 3.3. The storage architecture with NVM integration.

The objective of vFlash is to optimally manage the underlying heterogeneous storage system and provide transparent support for upper layers, such that the system performance of mobile devices can be improved. The basic idea in vFlash is to study the I/O behaviors of different applications and based on different I/O behaviors, vFlash adopts different storing

decisions. In this chapter, two techniques, intra-app and inter-app, are proposed to study application I/O behaviors. Inter-app utilizes the historical locality information of applications, while intra-app considers the I/O access patterns for each application.

Figure 3.3 illustrates the storage hierarchy by integrating NVM in mobile devices. In the storage hierarchy, a new software layer, vFlash, is proposed to manage the heterogenous storage system. In vFlash, the main component is apps monitor, whose function is to use application information to optimize the I/O performance. In order to achieve this goal, two techniques, inter-app and intra-app, are proposed in apps monitor. These two techniques are proposed based on different considerations. The inter-app technique is devised based on the consideration of the historical locality information. At different time periods of each day, a person tends to regularly use some fixed applications, while let the other applications sleep. Based on this historical locality information, inter-app technique can determine which application should be served by NVM at a given time period of each day. The intra-app technique is proposed based on the consideration of the I/O access patterns for each application. The frequent update requests introduce the most I/O burdens for NAND-flash-based storage. Therefore, the application that presents the frequent update I/O access pattern will have the higher priority to store data in NVM.

However, there is a challenge for vFlash to do so. Since vFlash is placed at the bottom of the I/O stack, the application information is lost when I/O requests reach this layer. Therefore, we need a mechanism to transfer the application information from applications to the proposed vFlash layer.



(a) The write-back policy.

(b) The write-through policy.

Figure 3.4. The disk cache effect.

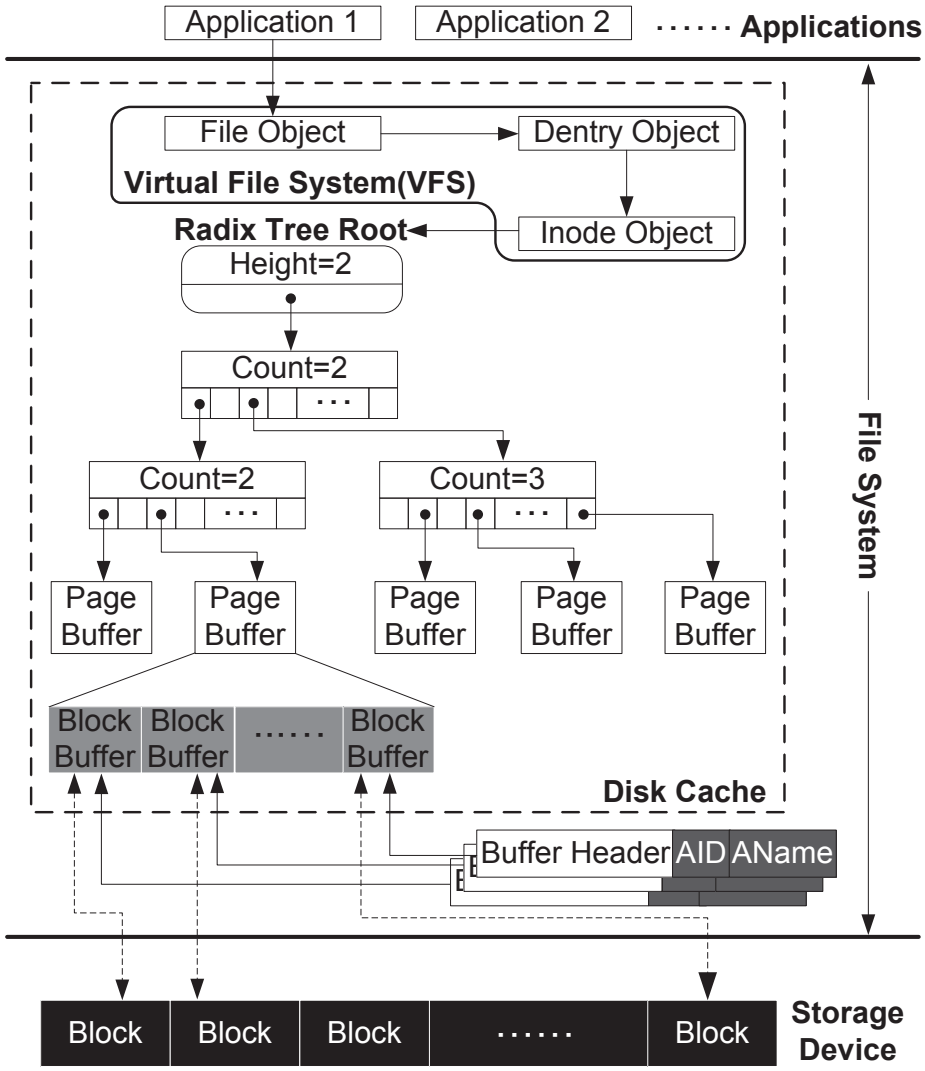


Figure 3.5. Disk cache in the file system.

3.3.2 Cross Layer Management

As we mentioned, by placing NVM at the bottom of the I/O stack, we can transparently manage NVM. However, the application information will be lost in the vFlash layer. In order to utilize the application information to better optimize the application performance, we need a cross-layer mechanism to transfer the application information from applications to the vFlash layer. We will analyze why the application information is lost in the vFlash layer, and how to transfer the application information from applications to the vFlash layer.

The main contributor to the loss of the application information is the disk cache. The disk cache is a buffer cache, allowing certain storage data to be kept in RAM without being directly written to the storage device as shown in Figure 3.4. In general, two classical caching policies, write-back and write-through, are employed to manage the disk cache. The default caching policy used in the file system is write-back, by which the applications get the I/O completion signal from the file system when data are stored in the disk cache. The write-back policy works with a flush daemon. A flush daemon is responsible for flushing the inconsistent RAM data to the storage device. If the data in the disk cache are needed to be flushed to the storage device, the flush daemon will send I/O requests to the vFlash layer. Therefore, vFlash can only obtain information of the flush process instead of the original process that writes this data. In contrast, a write-through cache performs to guarantee that all data are written to the cache and the storage device simultaneously. Therefore, vFlash can obtain information of the process that writes this data. However, the write-through policy severely degrades the overall system performance since every application needs to wait until data are written to the storage device. Thus, it is not a desirable solution for mobile device applications either.

In order to solve the above issue, we propose to transfer the application information from applications to the vFlash layer. We slightly modify the Linux kernel to record the application information in each I/O request. Therefore, even with the write-back caching policy, vFlash can still obtain the application information from the I/O request. In the following part of this section, we will analyze the underlying data structure of the disk cache in detail, and show how to transfer the application information from applications to vFlash.

Figure 3.5 illustrates the data structure of the disk cache. In Linux, files can have large sizes, even a few terabytes. When accessing a large file, the page cache may be filled with many pages. Sequentially scanning all of the pages would consume a lot of time. In order to efficiently lookup the data in the page cache, the Linux file system uses a search tree to organize the data in the page cache. The root of the search tree is represented by a radix_tree_root data structure. Nodes at the middle of the search tree store pointers pointing to the other nodes, and nodes at the bottom of the search tree store pointers pointing to page

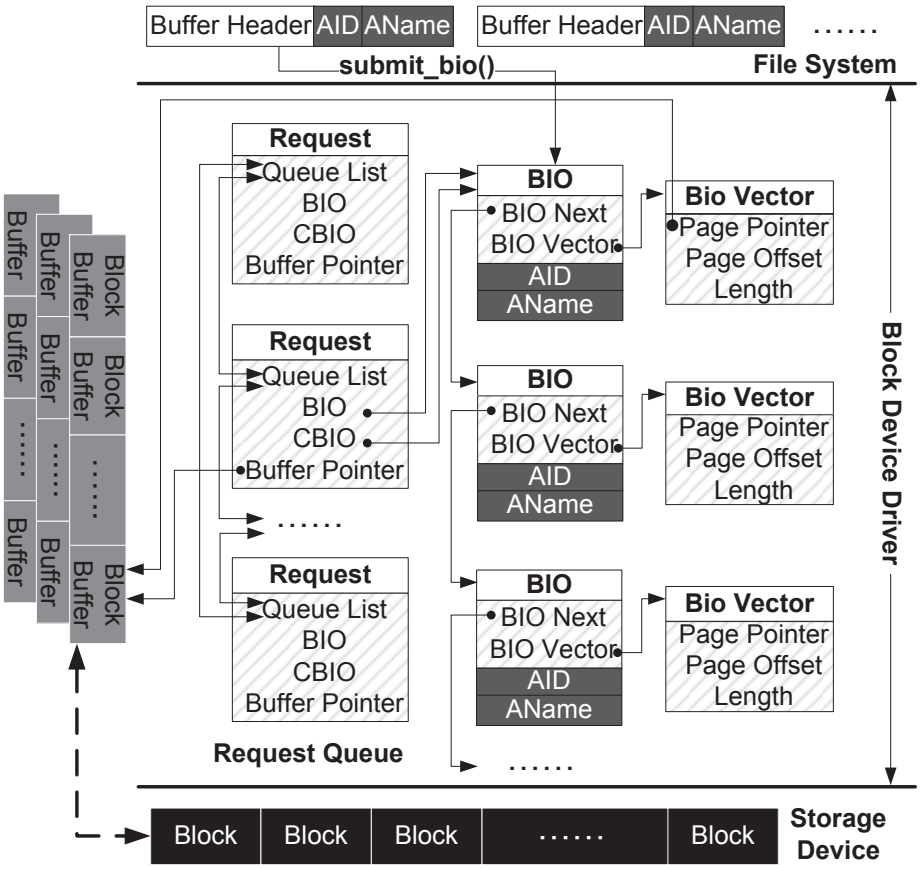


Figure 3.6. Data organization in the block device driver.

buffers. Each page buffer contains several block buffers, and each block buffer is pointed by a buffer header descriptor of type buffer_header. This descriptor contains the block buffer related information. We modify the buffer_header structure, and add the application identification (AID) and application name (AName) in the buffer_header structure. The application identification and application name are used to uniquely identify an application. When an application issues an I/O request, the Linux kernel will record the application information in AID and AName.

In the block device driver, the requests are linked into a request queue with the help of the *queue list* as shown in Figure 3.6. In the request structure, the accessed data are pointed by the *bio* field. The *cbio* and *buffer pointer* fields point to the first *bio* request that has not been transferred. In order to keep the application information in the block device driver, we

modify the *bio* data structure and add the AID and AName in the *bio* data structure. When the *buffer header* is converted to a block I/O request via a system function, *submit_bio()*, the AID and AName will be delivered from the *buffer header* to the *bio* structure. Finally, when vFlash receives an I/O request, it can use AID and AName to identify the application information from the *bio* data structure.

3.3.3 Inter-app and Intra-app Management

In the vFlash layer, apps monitor is proposed to capture and study the application I/O behaviors. Two techniques, inter-app and intra-app, are adopted in apps monitor. These two techniques utilize the historical locality and I/O access patterns to study the application I/O behaviors.

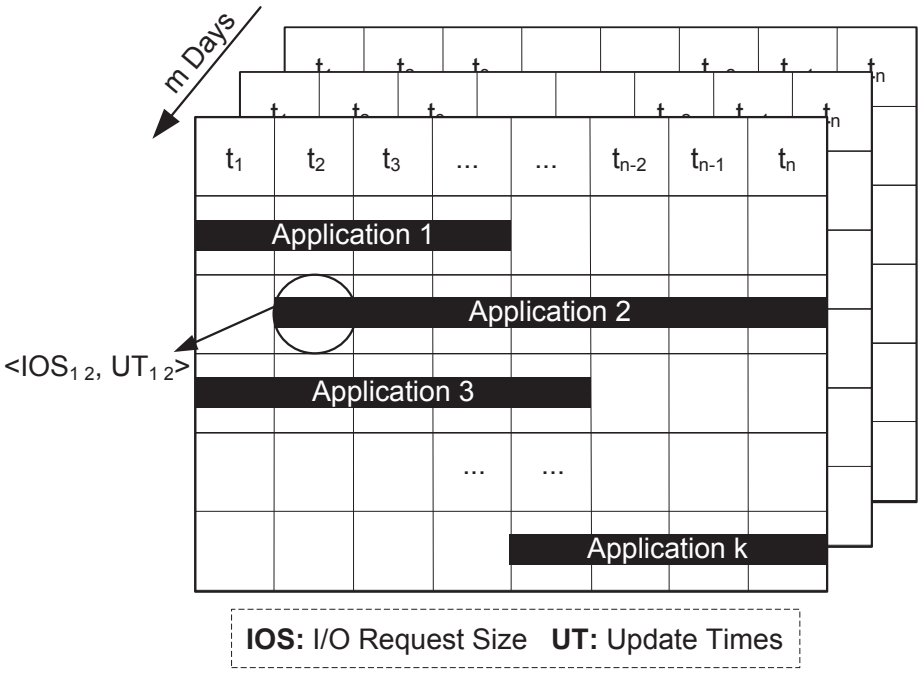


Figure 3.7. The data organization with the inter-app and intra-app technologies.

To collect the historical locality information, the inter-app technique uses several time tables to record the execution time for each application as shown in Figure 3.7. Each time table records the application execution time in one day, and one day is divided into

n ($n \geq 1$) time periods. To better predict the application behaviors, m ($m \geq 1$) tables are employed to record the application execution time for the up-to-date *m* days. With the help of the recorded historical locality information, we can predict which application will run in the next time period, and thus pre-allocate the required NVM spaces. IOS (I/O request size) is used to denote the required NVM spaces for one application, and will be discussed in the following part.

I/O Request Size: $IOS_{ij}(1 \leq i \leq m; 1 \leq j \leq n)$ is employed to represent the I/O request size for the ith day and jth time period issued from one application. The predicted IOS_{pre} of the jth time period can be calculated through Equation (3.1).

$$IOS_{pre} = \frac{\sum_{i=1}^{m} IOS_{ij}}{m}, \tag{3.1}$$

where $m \geq 1$. In this way, we can implement the inter-app technique. However, the inter-app technology can only predict which application will run and the required I/O spaces at a given time period. If too many I/O requests are generated at a given time period, free spaces in NVM may be used up. Under this condition, the intra-app technique is proposed to determine which application should have the first priority to allocate NVM spaces.

For intra-app, it applies I/O access patterns to optimize the I/O performance. In the NAND-flash-based storage system, the frequent update requests generate the most I/O burdens, since data updates can lead to the lengthy garbage collection operations in NAND flash memory. Therefore, if an application presents the frequent update feature, this application should be given the high priority to store data in NVM. In intra-app, UT (update times) is represented the data update times for each application and stored in the time table as shown in Figure 3.7. For each application, UT_{ij} represents the update times of the ith day at the jth time period. The predicted update times will be discussed in the following part.

Update Times: If NVM has sufficient spaces to allocate to each application, the UT information can be ignored. Otherwise, UT information can help to minimize the I/O burdens by first allocating the NVM spaces to applications with the frequent data update feature. The predicted UT_{pre} can be calculated through the following equation:

$$UT_{pre} = \frac{\sum_{i=1}^{m} UT_{ij}}{m}, \tag{3.2}$$

in which $m \geq 1$. In Equation (3.2), UT_{pre} is the average update times of m days in a given time period j. Then, UT_{pre} can be applied to predict the update frequency for each running application, and the inter-app technique is implemented.

3.3.4 Adaptive Space Preparation

For each time period, vFlash needs to prepare the NVM spaces in advance. The prepare operation may introduce the data migration and thus degrade the I/O performance. In order to eliminate the migration overhead while providing the NVM spaces for data caching, we propose an adaptive data preparation scheme. The adaptive data preparation scheme reclaims the NVM spaces by utilizing the system idle time and considering the data characteristics. The data are classified into three different categories, namely, read only data, write only data, and updated data. The read only data are not taken into account because of the following reasons. The read performance of NAND flash storage (e.g. $25us$) is much better than that of the write performance (e.g. $500us$) [31, 49]. In addition, If the data are frequently read, these data will be cached in the page cache of the file system. And, if the read only data are kicked out, these data are directly discarded by the file system without being flushed to the flash devices. However, the write data are needed to be flushed to the flash devices for the data persistence. So, the write data introduce much more I/O burdens compared with the read only data, and we mainly employ the NVM to cache the write data. When cleaning the data in NVM, the write only data are cleaned first, and then the updated data. The above cleaning procedure is elaborated based on the following considerations. Compared with the write only data, the updated data are more important since the data updates can generate the invalid data in flash memory and thus introduce the lengthy garbage collection. The next issue is to select an application to reclaim the NVM spaces. We adopt an execution time aware based method to perform the application selection with the consideration of the application temporal locality. The application with the maximum runtime distance (the time

interval between the current time period and the predicted execution time period) will be selected as a candidate and follow the above mentioned data characteristics to perform the cleaning procedure. If different applications present the same runtime distance feature, the application with the minimum UT_{pre} will be selected as a candidate and its data will be cleaned first.

Algorithm 2 describes how the adaptive space preparation scheme works. The input TT represents the time tables described in Section 3.3.3. In Algorithm 2, the adaptive space preparation scheme does not work under the following three conditions. The first condition is that the number of free spaces in NVM is larger than or equal to the predicted I/O request size. The second condition is that the system is busy, and the last condition is that all NVM spaces are free. If none of the above conditions is satisfied, the space preparation will be triggered. The adaptive space preparation scheme reclaims NVM spaces according to the system status, idle or busy. If the system is idle, the adaptive space preparation scheme continues to clean NVM spaces until the number of free spaces in NVM is larger than or equal to the required NVM spaces, or all the NVM spaces are cleaned. On the other hand, if the system is busy handling I/O requests, the adaptive space preparation scheme will wait until the system becomes idle. The adaptive space preparation also works with the consideration of the different data size. If the migrated data size is larger than the NAND flash page size, continuing migrating a large amount of data may heavily block the system response. To solve this issue, the adaptive space preparation scheme migrates data with a preemptive manner. That is, migrated data are divided into flash page size data, and each atomic migration procedure only contains one page size data. After migrating one page size data, the adaptive space preparation scheme will detect the system status. If the system becomes busy, the adaptive space preparation scheme will release the system resources and let the system response the I/O request first. By applying the preemptive migration, the performance overhead introduced by migrating a large amount of data can be significantly reduced.

Algorithm 2: Adaptive space preparation

Input: TT: Time tables

```
1  IOS_pre ← Calculate the I/O request size used in the
2  next time period according to TT
3  if The free spaces in NVM ≥ IOS_pre or The system is busy or
   All NVM spaces are free then
4      Adaptive space preparation does not work
5  else
6      while The system is idle and The NVM has occupied spaces and
       The free spaces in NVM < IOS_pre do
7          MD ← The write only data with the
8          maximum runtime distance
9          if MD is NULL then
10             APP_max ← The applications with the
11             maximum runtime distance
12             MD ← The updated data in the APP_max
13             with the minimum UT_pre
14         while MD data size > 0 do
15             PMD ← The flash page size data in MD
16             Move PMD from NVM to flash memory
17             Mark the PMD occupied NVM spaces as free
18             MD data size ← MD data size − PMD data size
19             if The system is busy then
20                 Wait the system to be idle
```

3.4 Performance and Overhead Analysis

In this section, we analyze the system performance improvement and the extra overhead introduced in the proposed scheme. The system response time is an important metric to evaluate the system performance. It is the time period from the point when an operation is issued to the point when the operation has been completed. The inputs of an I/O request are read and write operations. Thus, we conduct the analysis for the best-case and worst-case system response times of read and write operations. The symbols used in this analysis are listed in Table 3.2.

Table 3.2. The description of symbols used in analysis.

Symbols	**Description**
T_{rd}	The time to read one flash page
T_{wr}	The time to write one flash page
T_{erase}	The time to erase a flash block
T_{NVM_rd}	The time to read one flash page size data in NVM
T_{NVM_wr}	The time to write one flash page size data in NVM
N_{max_vpage}	The maximum number of valid pages in one block
N_{gc}	The number of garbage collection operations
N_{NVM_wr}	The number of NVM write requests in the unit of flash page size
N_{NVM_rd}	The number of NVM read requests in the unit of flash page size
N_{data_mig}	The number of flash page size data migrated from NVM to flash memory

The extra performance degradation is first reflected to the garbage collection opera-

tions, including extra valid page copies and block erasures, and the time is:

$$((T_{flash\_rd} + T_{flash\_wr}) \times N_{max\_vpage} + T_{flash\_erase}) \times N_{gc} \tag{3.3}$$

$$T_{flash\_erase} \times N_{gc} \tag{3.4}$$

In original mobile devices, data updates generate invalid pages in NAND flash memory. The generation of these invalid pages finally results in triggering the garbage collection operation, which needs to copy the valid pages and erase the dirty blocks. The worst-case and best-case overhead of garbage collection operations can be calculated through Equation (3.3) and Equation (3.4), respectively. The timing difference between the worst-case and best-case is reflected from the valid page copy procedure. If one block contains maximum number of valid pages denoted by N_{max_vpage}, the worst-case will occur and plenty of valid pages are needed to be copied from dirty blocks as shown in Equation (3.3). On the other hand, if a block has no valid pages, only block erasure operations occur as described in Equation (3.4). Since NVM supports in-place updates, the data updates can be effectively absorbed by first storing data in NVM. As a result, the proposed vFlash scheme can effectively improve the system response time by reducing the garbage collection overhead.

$$\begin{aligned}&(T_{flash\_wr} - T_{NVM\_wr}) \times N_{NVM\_wr} + \\ &(T_{flash\_rd} - T_{NVM\_rd}) \times N_{NVM\_rd}\end{aligned} \tag{3.5}$$

The proposed scheme not only reduces the garbage collection overhead but also can enhance the read and write speed. Equation (3.5) describes the read and write performance gain by integrating NVM. N_{NVM_wr} stands for the number of page size write requests issued to NVM, and N_{NVM_rd} represents the number of page size read requests issued to NVM. Without the proposed NVM integration scheme, these requests are directly issued to flash devices. Given that the I/O speed of flash memory is orders of magnitude slower than that of NVM, we can achieve great I/O performance gain as described in Equation (3.5).

$$(T_{NVM\_rd} + T_{flash\_wr}) \times N_{num\_mig} \tag{3.6}$$

The proposed scheme may introduce the data migration overhead. The performance degradation introduced by data migration can be represented in Equation (3.6). T_{NVM_rd} and T_{flash_wr} represent the time to read one page size data in NVM and the time to write one page in flash memory, respectively. N_{num_mig} denotes the number of page size data migrated from NVM to flash memory. The migration operations first read data from NVM and then store the read data into flash memory. However, the migration overhead does not introduce much performance degradation to applications in the vFlash scheme. This is because the proposed adaptive space preparation utilizes the system idle time to do migration.

The proposed scheme uses time tables to record the I/O request size and update times for each application. This table consumes some RAM spaces. However, the RAM cost for the time tables is acceptable. Assume we use one hour as a time interval, and we record seven days. A person installs 100 applications in his/her mobile device, and we use 8 bytes to record the IOS and UT. Then, in the worst case, the RAM cost for the time table is $100 * 24 * 7 * 8 = 131.25KB$. Since usually these 100 applications do not always run in one day, the RAM space consumes introduced by the time tables should be less that of the worst case. In addition to the time tables, vFlash also needs a mapping table to record the flash memory address when data are stored in NVM. This RAM space consumption is also acceptable since even with a small capacity of NVM (e.g. 16MB used in the experiment) we can achieve great performance improvement. Assume the capacity of NVM is 16MB, and one flash page size is 4KB. If each mapping item consumes 8 bytes, the RAM space overhead is $\frac{16*1024}{4} \times 8 = 32KB$ in the worst case (all the NVM spaces are utilized).

3.5 Evaluation

In this section, we present our experimental results with analysis. We compare and evaluate the proposed vFlash scheme with the baseline schemes in terms of the I/O performance and

energy consumption. The baseline schemes adopt the stock Android 4.2 system and the NVM integration with the LRFU [55] management. LRFU management is designed with the consideration of the least recently used (LRU) and least frequently used (LFU) policies. The performance evaluation is conducted on an Android platform with a Samsung ARM Cortex-A9 quad-core processor and a 64Gb NAND flash memory chip.

3.5.1 Experimental Setup

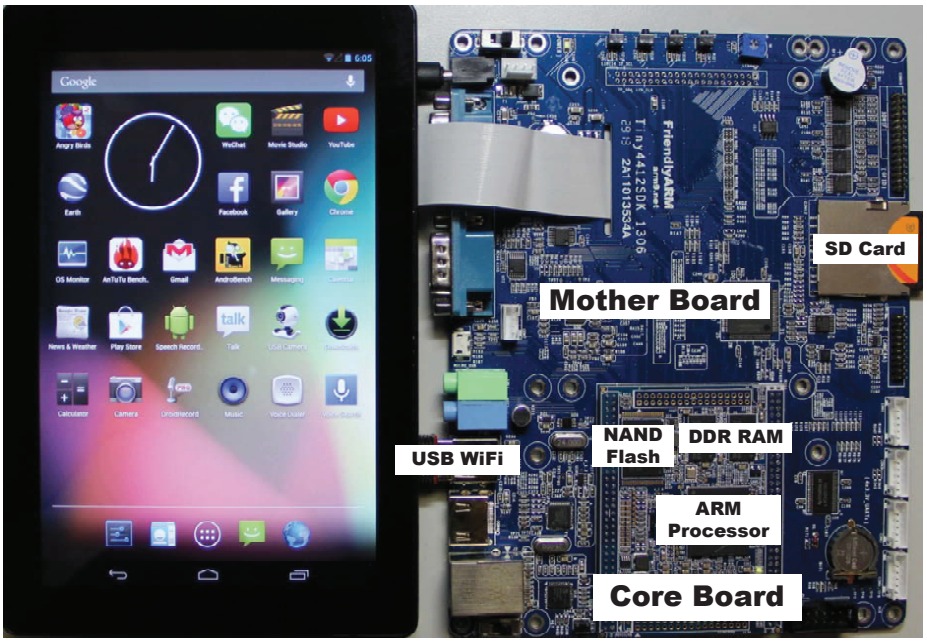


Figure 3.8. Experimental Android platform.

A hardware platform is used to evaluate the baseline schemes and proposed vFlash scheme. Figure 3.8 shows the top view of the hardware platform. The evaluation platform contains a core board and a mother board, which are connected via a one-to-one pin connector. The core board includes an ARM Cortex-A9 quad-core processor (Samsung Exynos4412 [74]) with ARMv7 architecture. In this platform, the ARM processor core runs at 1.4GHz and consists of a 32KB instruction cache and a 32KB data cache. In addition to the processor, the core board is also equipped with 64Gb NAND flash memory, 128Gb SD card, and

1GB DDR3 RAM. We use a small portion of RAM to simulate NVM, since the I/O accessing patterns of NVM are similar to RAM [39]. That is, RAM and NVM are all byte addressable and support in-place updates. The physical interfaces, such as the USB port, are designed in the mother board, and a WiFi module is connected through the USB interface.

Table 3.3. The Android applications and their usage.

Applications	**Using Scenarios**
Google Earth	Search Places and View Streets
Browser	Browse Websites and Search Pictures
WeChat	Chat with Friends
YouTube	Watch Online Movies
Facebook	View Friends' Status
Gmail	Receive Emails and Send Emails
Gallery	View Pictures
Download	Download Applications
Install	Install Applications
Media Player	Watch Movies and Listen to Music
Angry Birds	Play the Game and Select Levels

Android 4.2 with the Linux kernel 3.5 is ported to this platform. The proposed scheme vFlash is implemented between the block device driver and the lower physical device driver (e.g. NAND flash memory device driver) in the Linux kernel. Since vFlash needs to utilize application information to optimize the I/O performance, we modify the ext4 file system and block device driver to transfer the application information from applications to the vFlash layer. For fair comparisons, the same configuration has been adopted for the base-

line and vFlash schemes. We have covered a wide range of applications to demonstrate the effectiveness of the proposed scheme. In this experiment, we employ 11 different kinds of applications, such as Google Earth for navigation, Browser for web browser, and Facebook for social network. These applications and their using scenarios are shown in Table 3.3. These applications are frequently used in people's daily life and iteratively issue I/O requests to the storage system. Three different persons are selected to evaluate the proposed scheme, and each person uses this board for one month. In the time table, the time interval is configured to 1 hour, and vFlash records up-to-date 7 days' data to predict the IOS and UT.

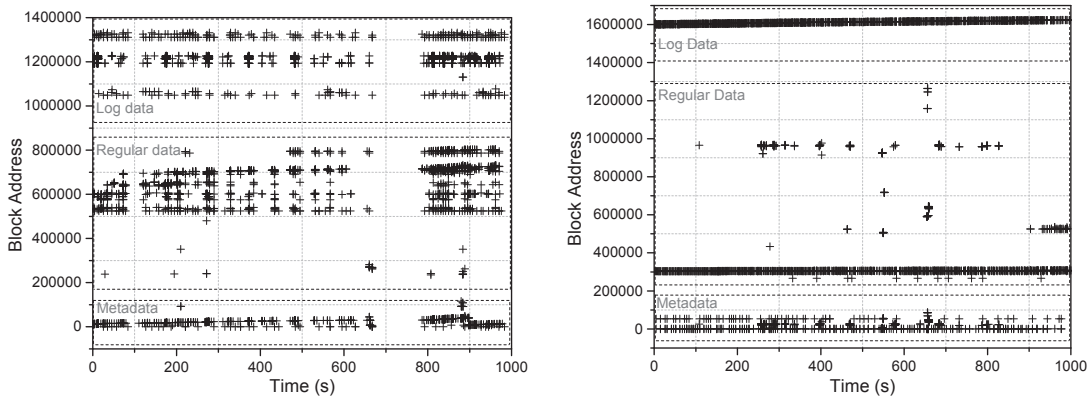
3.5.2 Results and Discussion

In this section, we present the experimental results with analysis. We first give the offline data to deeply analyze the application data characteristics in the Android environment. Then, we show the performance and energy consumption improvement by comparing the baseline schemes with the proposed vFlash scheme. Finally, the effect of the adaptive space preparation scheme is discussed.

We utilize PCM as an example of NVM to study the benefits of the proposed scheme. The response time of NAND flash device is directly measured with the help of the Linux kernel function $do\_gettimeofday$. The other results are obtained by capturing I/O requests to PCM and NAND flash memory and calculating with the models of NAND flash memory and PCM. The read latency and write latency of PCM are configured to be $50ns$ and $150ns$, respectively. In order to study the energy consumption, the read and write energy of PCM are configured to be $1J/GB$ and $6J/GB$, respectively. The read and write energy of NAND flash memory are set to $1.5J/GB$ and $17J/GB$, respectively. NAND flash memory also supports the erase operation which is not available in PCM. The energy consumption of the erase operation is configured to be $10J/GB$ [39, 41, 52, 87, 104, 105]. In the experiment, the PCM capacity is configured to 16MB. This configuration is based on the consideration of the PCM price which is much higher than that of flash device. In the future, the NVM price may drop. So we will explore how the proposed scheme is influenced with increasing the PCM

capacity in the future work.

Offline Data Analysis



(a) WeChat. (b) Facebook.

Figure 3.9. The I/O characterization with running (a) the WeChat application, and (b) the Facebook application.

We select two representative applications, WeChat and Facebook, as examples to analyze the application I/O behaviors. Figure 3.9(a) and Figure 3.9(b) compare the I/O characterization with running the WeChat application and the Facebook application, respectively. These results are obtained by modifying the proposed vFlash layer in the Linux kernel and outputting the captured I/O requests to a trace file. As shown in Figure 3.9, there are three different kinds of data, including metadata, log data, and regular data. Metadata and log data are generated by the ext4 file system and usually can be easily identified according to the data block address [21,96], which is an address from the file system's perspective and different from the NAND flash block address. The metadata usually present the frequent update feature and should be migrated to NVM with the lowest priority. Different from metadata, log data are sequentially written to the storage device with the infrequently accessed feature. Therefore, the log data will be migrated to the flash storage with the first priority. The regular data are generated by applications and the data access patterns are highly dependent on applications. For example, compared with the WeChat application, regular data generated by the Facebook application present the frequent update feature, so these data should be migrated

to NVM with the lower priority.

Performance Improvement

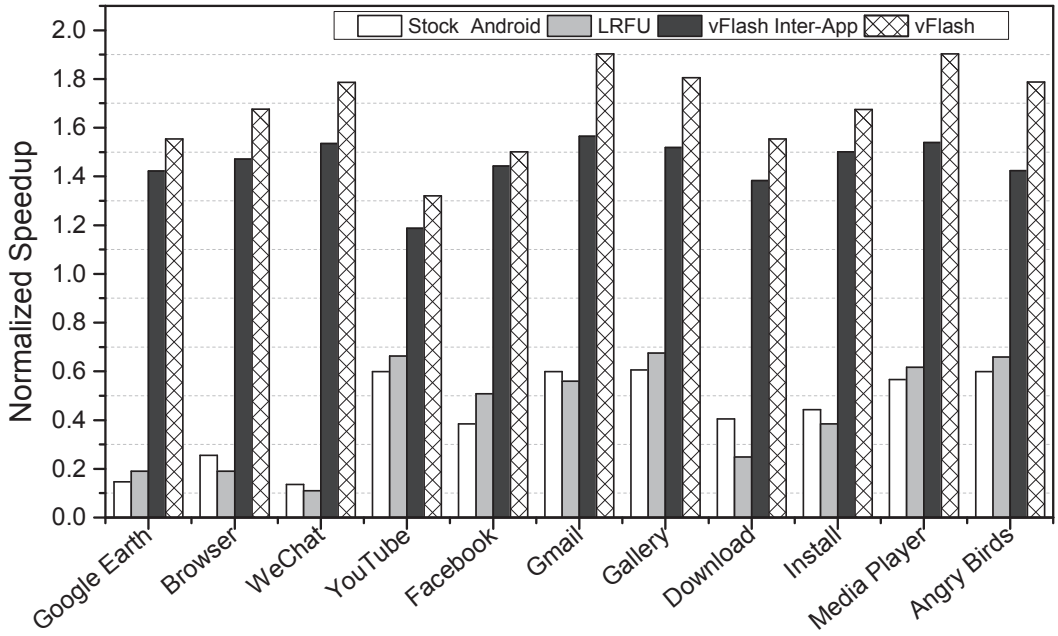


Figure 3.10. The I/O performance by comparing the baseline schemes with the vFlash scheme (the first user).

The normalized speedup is adopted to quantitatively analyze the performance improvement [103]. We first obtain the I/O throughput that is the number of megabytes processed per second for each application, and then calculate the average I/O throughput of all the applications. The normalized speedup is obtained by dividing the I/O throughput by the average throughput.

Figure 3.10, Figure 3.11, and Figure 3.12 present the experimental results of the I/O performance for the baseline and vFlash schemes with three different users. Several phenomena can be observed from the experimental results. First, although different users may have different usage behaviors, the proposed vFlash scheme can significantly improve the I/O performance compared with the baseline schemes for all these three users. For example, for three different users, the I/O performance by employing vFlash outperforms that of the

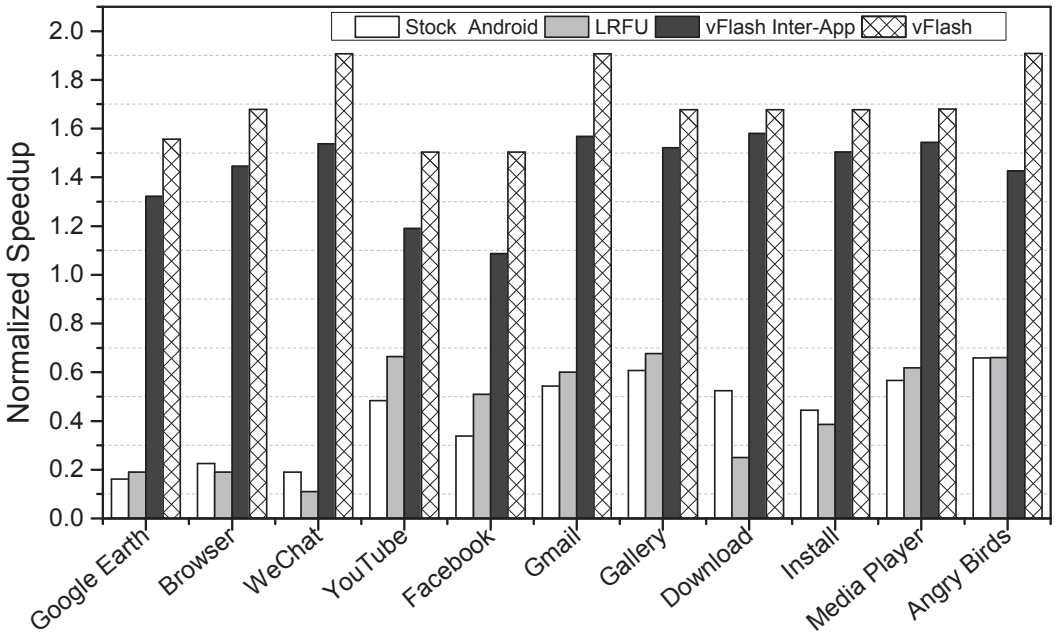


Figure 3.11. The I/O performance by comparing the baseline schemes with the vFlash scheme (the second user).

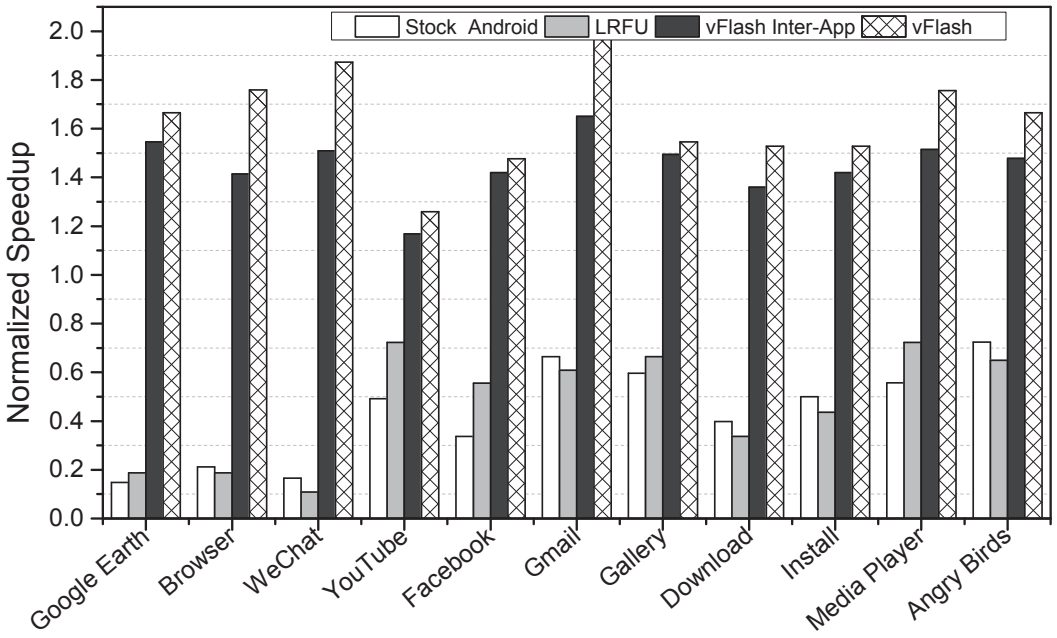


Figure 3.12. The I/O performance by comparing the baseline schemes with the vFlash scheme (the third user).

stock Android system up to 9.54 times (2.89 times on average), 8.63 times (2.93 times on average), and 10.27 times (2.76 times on average), respectively. These results demonstrate that the user behaviors are relative fixed for each person, and the vFlash scheme can well utilize the user behaviors to optimize the I/O performance with NVM integration. The second observation is that the LRFU management policy could not better manage the NVM resources under the mobile environment. The LRFU scheme can only enhance the I/O performance to 3.92% on average for all the three users compared with the stock Android system. This is because applications in mobile devices are switched much more frequently than those in desktops or servers. While an application is accessing hot data in one moment, another application might be launched in the next moment and access other data. If we do not use the cross-layer technique to transparently mange NVM spaces, previous generated data, which may become hot data later, are easily to be kicked out by the data generated by the other applications. Moreover, mobile applications (e.g. WeChat) tend to generate many new data. If the spaces are not prepared well, the NVM spaces can be quickly used up. This finally results in aggressively cleaning the NVM spaces for the new I/O requests, and the performance even becomes worse than that of the stock Android system (e.g. Browser shown in Figure 3.10). Last, the results show that the proposed inter-app and intra-app techniques play different roles in addressing the storage performance improvement and can well cooperate with each other. The experimental results illustrate that with intra-app technique, the proposed vFlash scheme can further improve the I/O performance to 15.77% on average for all the three users compared with the inter-app technique. Compared with the stock Android system, the proposed scheme reduces 30,250 block erasure times. The typical block erasure latency is $1.5ms$ [31, 49, 87, 104]. By using Equation (3.4), we can calculate that the performance overhead introduced by block erasures is reduced by $30,250 \times 1.5 = 45,375ms$.

Energy Consumption

Energy consumption is another important metric to evaluate the effectiveness of the proposed scheme. The results shown in Figure 3.13, Figure 3.14, and Figure 3.15 illustrate the normalized energy consumption of the baseline and vFlash schemes. From the experimental

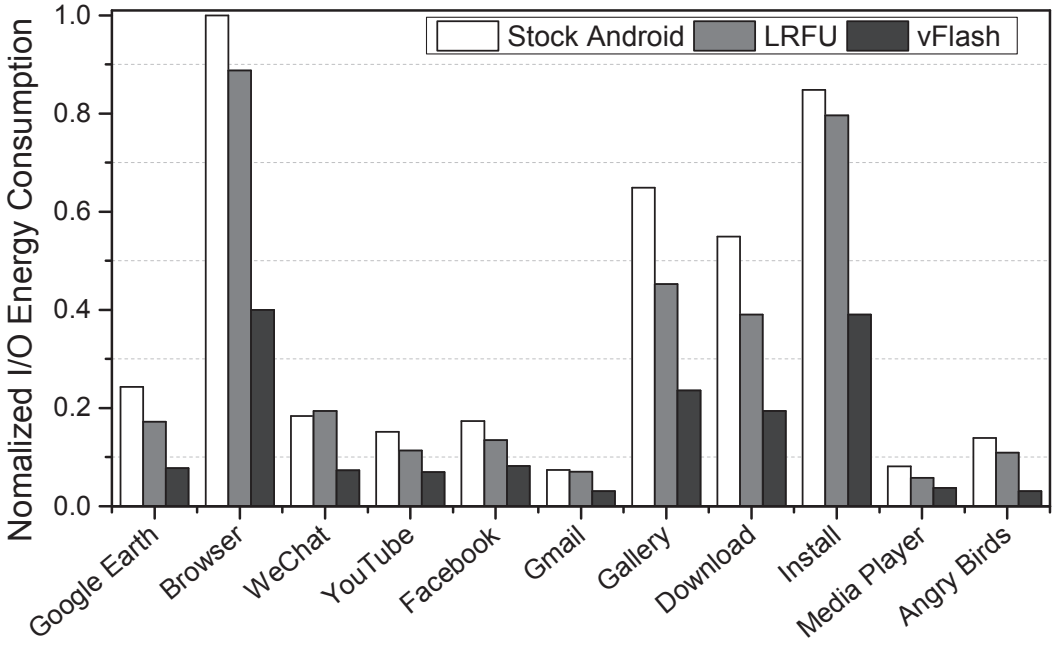


Figure 3.13. Illustration of the I/O energy consumption (the first user).

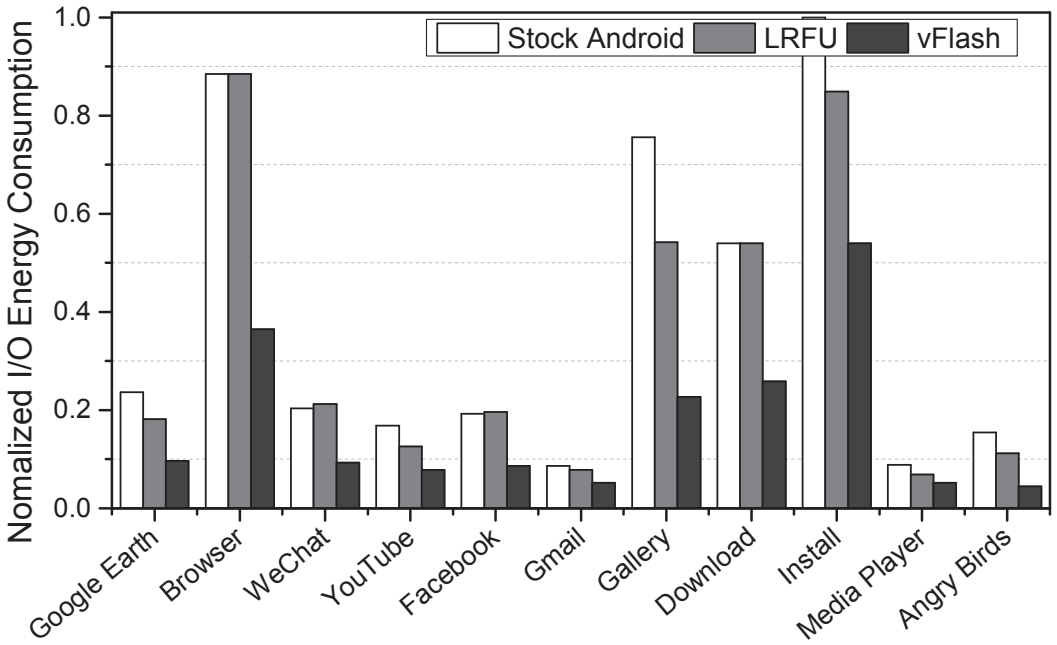


Figure 3.14. Illustration of the I/O energy consumption (the second user).

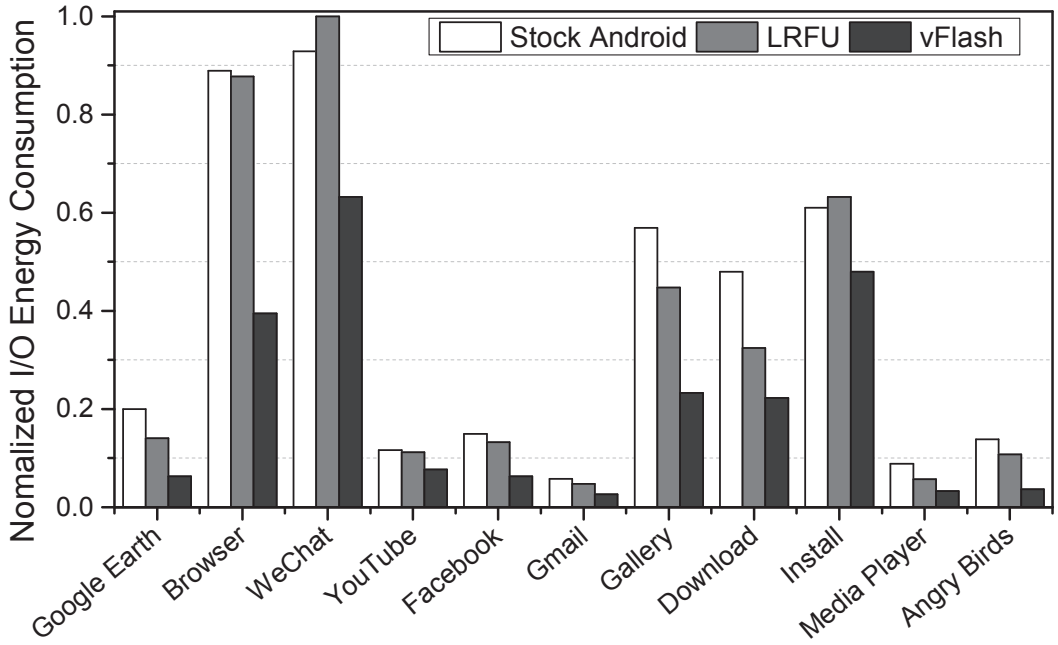


Figure 3.15. Illustration of the I/O energy consumption (the third user).

results, we can discover that vFlash can effectively reduce the I/O energy consumption compared with the baseline schemes. For example, in Figure 3.13, the proposed vFlash scheme can achieve I/O energy consumption reduction to 60.36% and 51.98% on average compared with the stock Android system and LRFU, respectively. The energy consumption reduction for the vFlash scheme mainly benefits from two aspects. The first aspect is that the read and write energy consumption of PCM is much lower that that of NAND flash memory. The second aspect is that the proposed vFlash scheme can effectively reduce the data updates in NAND flash memory, which finally results in the effectiveness reduction of the total number of garbage collection operations in the flash storage.

Adaptive Space Preparation Effect

Since the adaptive space preparation needs to migrate data from NVM to flash storage, this strategy may introduce the performance overhead. Figure 3.16 illustrates the normalized performance degradation caused by the adaptive space preparation. These results are measured

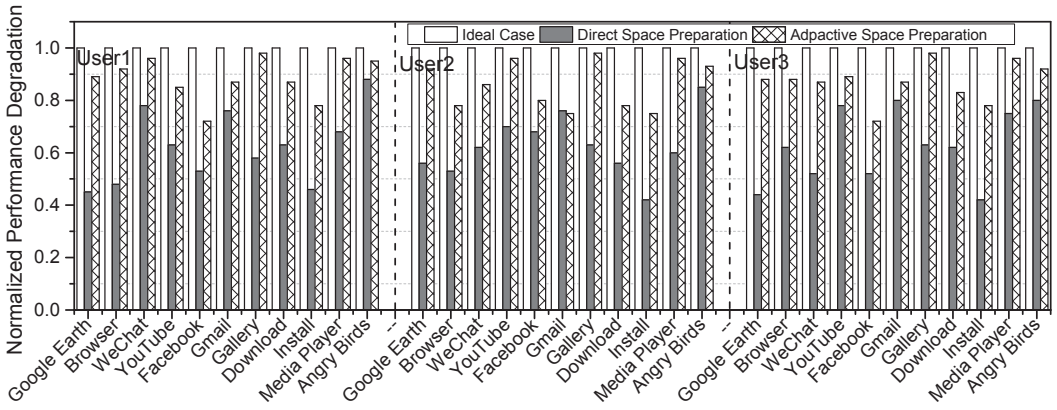


Figure 3.16. The performance degradation caused by the space preparation.

by monitoring the delays of the I/O requests caused by the data migration. The ideal case assumes that no request is blocked by the data migration, and the normalized performance is always 1. The direct space preparation performs the data preparation without utilizing the system idle time. From Figure 3.16, we can observe that the proposed adaptive data migration introduces about 12.72% performance degradation on average compared with the ideal case, while it can achieve 39.33% performance improvement on average compared with the direct space preparation. These results demonstrate that the data migration overhead introduced by the adaptive space preparation scheme is acceptable, and can be greatly eliminated with the help of utilizing the system idle time.

3.6 Summary

In this chapter, we propose a unified NVM and flash memory architecture to improve the I/O performance for mobile devices. We also present a software layer, called vFlash, to manage this hybrid storage architecture. The basic idea in vFlash is to cross-layer transparent utilize NVM resources to minimize the application modifications while maximize the I/O performance for NAND-flash-based mobile devices. We conduct experiments on a set of realistic I/O workloads of Android applications running in a hardware platform. The experimental results show that the proposed scheme can significantly enhance the I/O performance and

reduce the energy consumption for mobile devices.

CHAPTER 4

IMAGE-CONTENT-AWARE I/O OPTIMIZATION FOR MOBILE VIRTUALIZATION

4.1 Introduction

Mobile virtualization is gaining more and more popularity in mobile devices, since it can provide multiple platforms with a single device and can defend against increasing security threat. Major virtualization providers, such as VMWare [84] and Citrix [27], have started to provide their mobile virtualization solutions for mobile devices. By allowing a physical device to host multiple virtual machines (VMs), mobile virtualization can reduce development cost, shorten time-to-market with code reuse, and enhance privacy and security with VM isolation.

While mobile virtualization provides all the benefits, it also introduces considerable overhead with additional software stack layers. Particularly, storage accessing overhead is increased. In mobile devices, NAND-flash-based devices, such as NAND flash memory/SD cards, are widely used as secondary storage. NAND flash has well-known constraints such as erase-before-rewrite, limited lifetime, and long time overhead for garbage collection [38, 46]. However, mobile virtualization was not designed to take the constraints of NAND flash into consideration. Without any optimization, VMs place extra burden on the NAND flash storage system in mobile platform. When a guest OS creates a file system, a VM image file contains both the guest file system metadata and regular file data. In a VM image file, the metadata will be updated much more frequently compared with regular data because most file operations to regular files will also lead to the same metadata being updated. The same thing occurs in the host operating system, since the frequent updating of the image file also

leads to the host metadata being updated frequently. The frequent updating of the same metadata will lead to frequently triggering of out-of-space update and garbage collection in flash storage system. Therefore, both the system performance is degraded and NAND flash's lifetime is shortened [12, 43, 56, 79, 94]. A promising solution to overcome this drawback is to identify the metadata and store it in a faster and more endurable NVM (non-volatile memory). In this way, we can greatly reduce unnecessary updates to NAND flash, which will further result in less garbage collection overhead and improving the I/O performance and flash lifetime.

Wu et al. [96] explored how to classify file system metadata from regular data and used dedicated fine-grained page-level mapping to manage metadata. Their work showed promising performance and lifetime improvement in flash system. Their work can effectively identify metadata for the host OS file system. However, it cannot identify the metadata in a guest OS file system. In the virtual machine environment, guest OS file system metadata are stored in a disk image file. So, the guest file system metadata are seen as the regular file data in the mobile virtualization platform, and the unique challenge is how to separate the guest file system metadata from regular data in the mobile virtualization platform. More specifically, the most important challenging issue is how to identify the guest file system metadata data via the IBA (image block address), which is translated through the VBA (virtual block address) via a two-level mapping table. Without additional support, it is difficult for the host OS to directly identify file system metadata in a guest OS disk image file.

In this chapter, we propose an image-content-aware scheme to effectively identify both the host and guest metadata in a mobile virtualization platform. In order to achieve this, the proposed image-content-aware scheme identifies guest OS file system metadata according to the I/O request address in the guest OS file system, and uses a padding flag to the virtual machine to trace the metadata. After the metadata are successfully identified, both the guest metadata and host metadata are stored in a small faster and endurable NVM, such as phase change memory. Smart data management techniques are proposed to manage flash memory and the auxiliary NVM. With proposed management techniques, we can greatly eliminate the frequent update effects to the NAND-flash-based storage caused by the file

system metadata in a VM image file and in the host file system. The main contributions of this chapter include:

- To the best of our knowledge, this is the first work to identify the file system metadata from regular data in a guest OS VM image file under mobile virtualization.

- We propose to utilize a small faster and more endurable NVM to store file system metadata to reduce the flash memory I/O traffic with negligible extra cost.

- Smart data management techniques are proposed to manage flash memory and the additional NVM.

The proposed techniques are evaluated on an embedded platform with an ARM Cortex-A15 processor and a 64Gb NAND flash memory chip. The experimental results show that the proposed scheme not only improves the system response time, but also enhances the lifetime in NAND flash memory.

The rest of this chapter is organized as follows: Section 4.2 presents the background of this work, and Section 4.3 introduces the motivation for conducting this work. Section 4.4 detailed describes the proposed scheme. In Section 4.5, we analyze the performance and overhead of the proposed scheme. The experimental results are presented in Section 4.6. Finally, Section 4.7 summarizes this chapter.

4.2 Background

In this section, we introduce the background of mobile virtualization. We first introduce the properties of NAND flash memory and PCM in Section 4.2.1, and then present the storage architecture in Section 4.2.2. Finally, we analyze the metadata in Section 4.2.3.

Table 4.1. The characteristics of NAND flash memory and PCM [18, 31, 76, 104].

Attributes	**NAND**	**PCM**
Non-Volatility	Yes	Yes
Byte Addressability	No	Yes
Bit Alterability	No	Yes
Read Energy	1.5 J/GB	1 J/GB
Write Energy	17 J/GB	6 J/GB
Erase Energy	10 J/GB	No
Read latency	$\sim$25 us	$\sim$50 ns
Write latency	$\sim$500 us	$\sim$1 us
Erase latency	$\sim$1.5 ms	No
Endurance	$10^4 - 10^5$	$10^6 - 10^8$

4.2.1 NAND Flash Memory and PCM

Table 4.1 summarizes the characteristics of NAND flash memory and PCM. As shown in Table 4.1, PCM and NAND flash memory both have the non-volatile feature; however, PCM also has the byte addressability property and bit alterability property, which are not available in NAND flash memory. These two properties determine that PCM can support in-place update, while NAND flash memory should do erase operations before update. Also, PCM presents much shorter read/write latency and lower energy consumption compared with NAND flash memory, even though its byte addressability has already made it superior to NAND flash on handling reads/writes of small random data such as the file system metadata. Due to the heating process, each PCM cell only can sustain $10^6 - 10^8$ writes before a failure occurs. The read/write operations have asymmetric performance and energy consumption [48, 78, 97]. However, PCM still has much better endurance than that of NAND flash memory. Therefore, PCM is rapidly developed as a promising candidate for memory and/or storage in the design of embedded computing systems due to its shock-resistance, non-volatility, byte addressability, and low energy consumption. The capacity and price con-

strains of PCM in the market make it impossible to fully replace NAND flash memory as the storage devices in mobile devices.

4.2.2 Storage Architecture

Figure 4.1 shows a typical storage architecture of mobile virtualization. Similar to traditional virtualization environment, three different software components are packaged above the storage media: host operating system, VM, and guest operating system [37, 60]. In this chapter, the VM is a more general concept, which can be represented as the virtual machine or virtual machine monitor (VMM). In the full virtualization environment, the VM denotes the virtual machine. In the hardware-assisted virtualization environment, the VM is used to denote the virtual machine monitor (VMM). VM acts as a bridge that connects the guest operating system and the host operating system. The function of VM is to emulate hardware devices that execute programs like a physical machine, such as emulated storage media shown in Figure 4.1. With the help of VM, different kinds of mobile operating systems, such as Google Android operating system and Apple iOS operating system, can simultaneously run in one mobile devices. These different mobile operating systems are called guest operating systems and run as multiple threads in the host operating system. Each guest operating system has its own disk space represented as a VM image file, and each VM image file keeps its own metadata, such as block bitmap in ext file system. Metadata show the frequent updating feature that both does harm for the NAND flash based storage media and shortens the flash's lifetime.

4.2.3 Metadata Analysis

In this part, we use a case to study the file system metadata. We employ a widely used VM image format qcow2 [1] and ext4 file system [4] to show how the file system metadata works in a VM image file. The qcow2 image format is one of the most popular used image formats in mobile virtualization, which can be supported by many virtual machines, such as QEMU [8]. The ext4 file system is the fourth extended filesystem, and it has become the

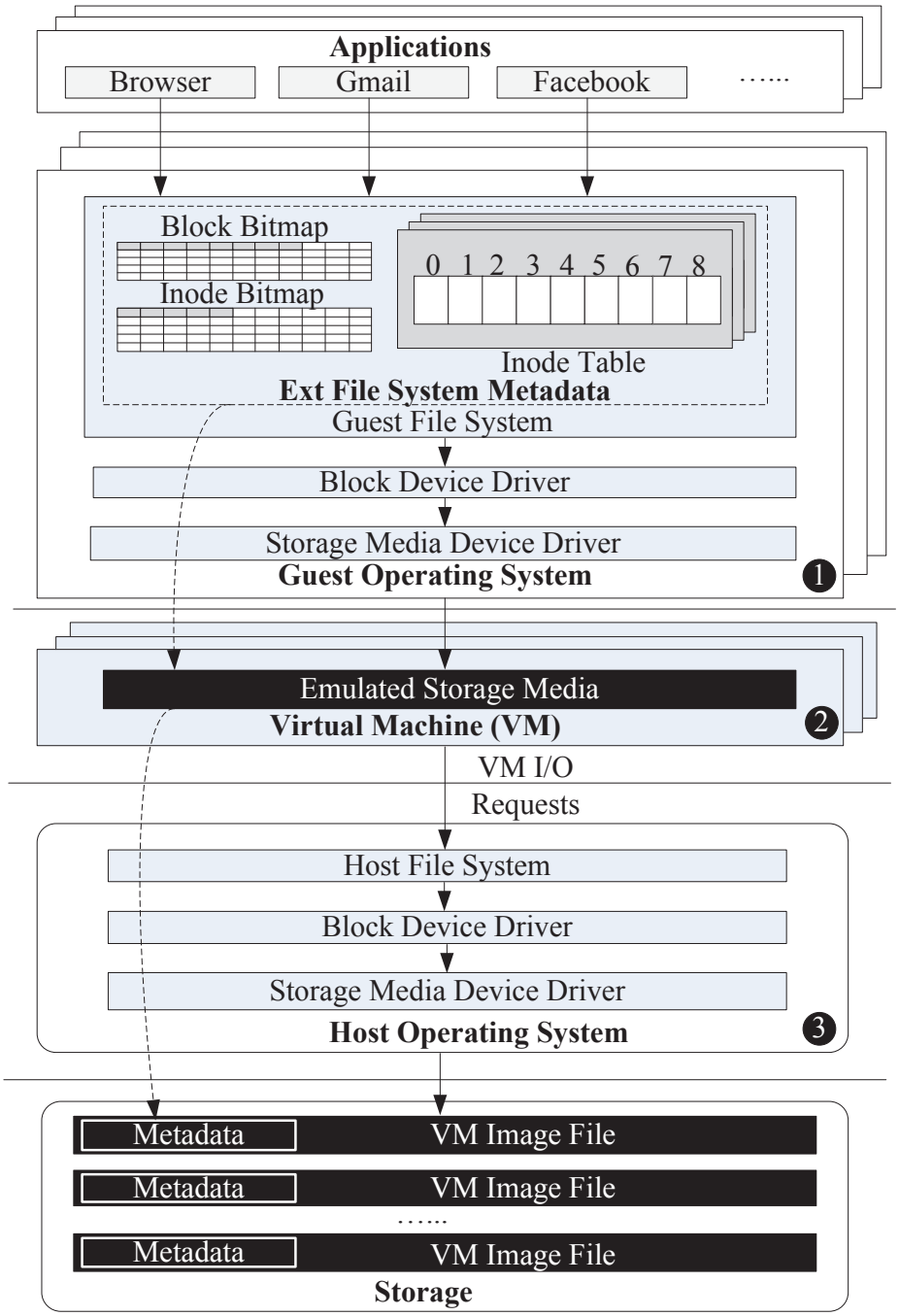


Figure 4.1. The storage architecture of mobile virtualization.

default file system on popular Linux based embedded operating systems, such as Android operating system.

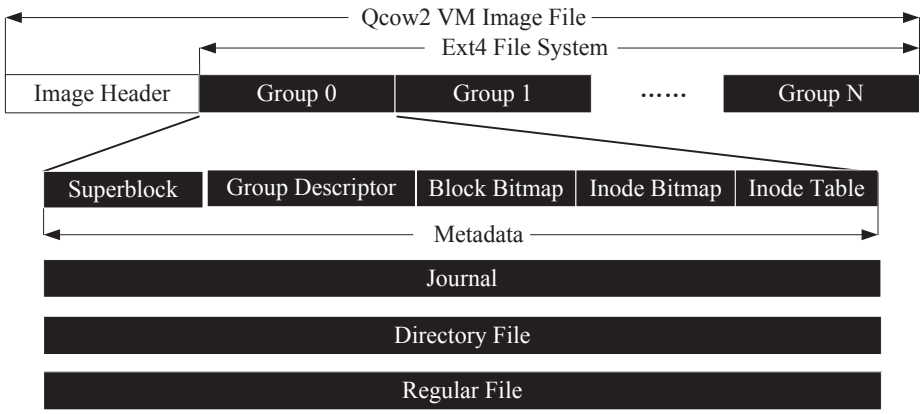


Figure 4.2. The qcow2 VM image file format with the ext4 file system.

As shown in Figure 4.2, qcow2 file begins with an image header, which contains image format related information. The virtual machine uses these information to identify the image format and loads the image file content. For example, the first four bytes in an image file contain a magic number, and virtual machine uses this magic number to identify the image format. The other image space excepting the image header is exposed to the user program and managed by the ext4 file system. In the ext4 file system, the image space is split into several groups, and each group contains four parts, including file system metadata, journal, directory file, and regular file. File system metadata are all in fixed, well-known locations [96], and they mainly consist of five important elements, including superblock, group descriptor, block bitmap, inode bitmap, and inode table.

Figure 4.3 and Figure 4.4 illustrate the detailed file creation and deletion process, respectively. If a VM creates a new file under the root directory, such as *a.dat*, the following procedures will occur. First, ext4 file system reads the inode table to find the root directory location as shown in step (2). In ext4 file system, the location of the root directory is pointed by the second inode and the address of root directory can be obtained from inode table. After getting the root directory disk location, such as block 100, ext4 file system will read

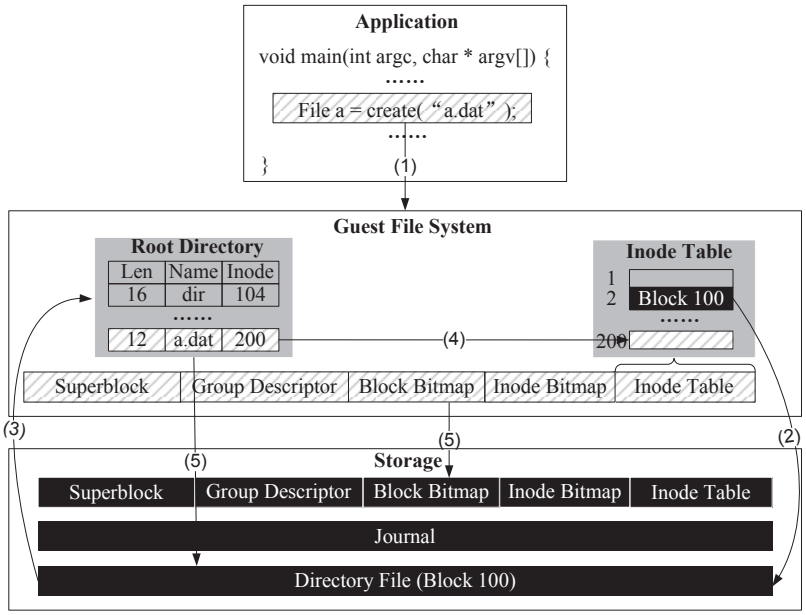


Figure 4.3. A file creation procedure in a VM image file.

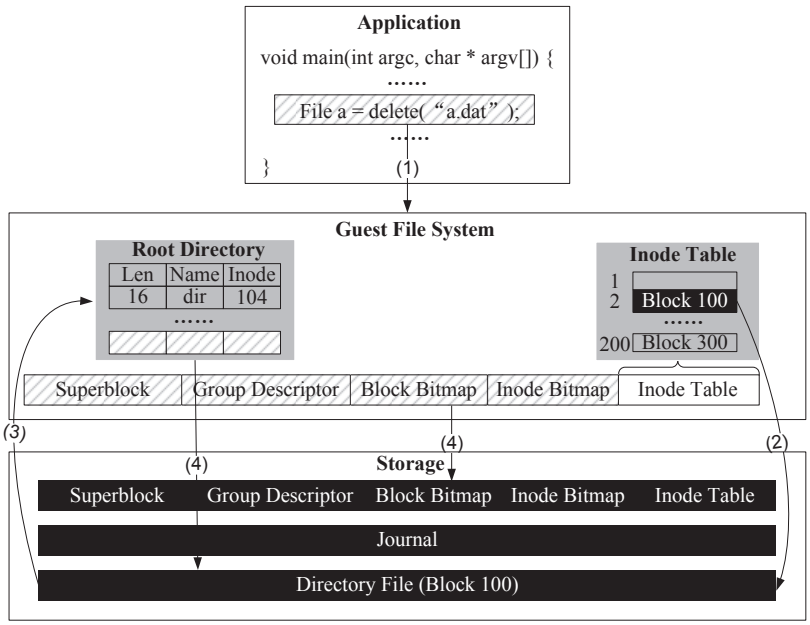


Figure 4.4. A file deletion procedure in a VM image file.

the root directory content as shown in step (3). Then, file system adds a new directory entry in the root directory and allocates an empty inode to the new directory entry in step (4). The new directory entry mainly contains three parts: *len, name and inode*, which denote the length of this directory entry, file name and the number of the inode that this directory entry points to, respectively. Since an empty inode is allocated, the content of superblock, group descriptor and inode bitmap are updated correspondingly. The updates of block bitmap and the contents of the inode table depend on the storing location of the new directory entry. If the new directory entry is allocated to a new block, the contents of block bitmap and inode table are updated. Otherwise, these two contents are untouched. Figure 4.3 (b) shows the file deletion operation which is a reverse operation of the file creation operation.

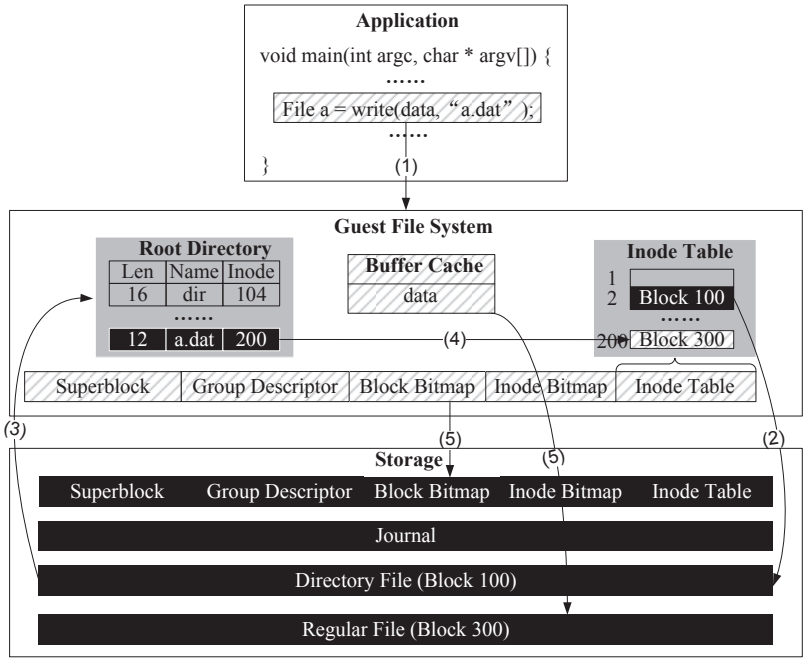


Figure 4.5. A file content increase procedure in a VM image file.

Figure 4.5 and Figure 4.6 illustrate a file content increase and shrink process, respectively. These two processes all need five steps. In step (2), ext4 file system uses the root directory inode, more specially inode 2, to retrieve the root directory content. Then, in step (3), ext4 file system gets the required file directory entry, and uses the inode table of the directory entry to obtain the file content inode (the inode 200 in the example). With the help

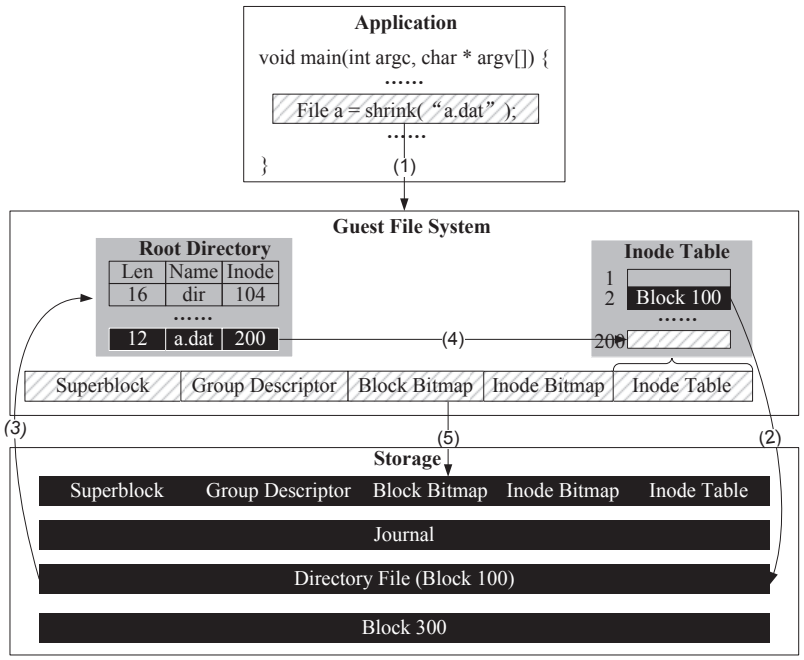


Figure 4.6. A file content shrink procedure in a VM image file.

of the file content inode, ext4 file system can locate the physical blocks which contain the real file content as shown in step (4). Then, the real file content increase or shrink operation will be executed by adding or reducing the file contents. The file content increase or shrink procedure requires to allocate new physical blocks or recycling discarded blocks. As a result,superblock, group descriptor, block bitmap and the contents of the inode table are updated as shown in step (5).

The above analysis shows that the frequent update of metadata is the intrinsic feature of the file system, since the file system metadata are responsible for recording the changes of the regular data. Normally, each file operation is accompanied with a file system metadata updating procedure.

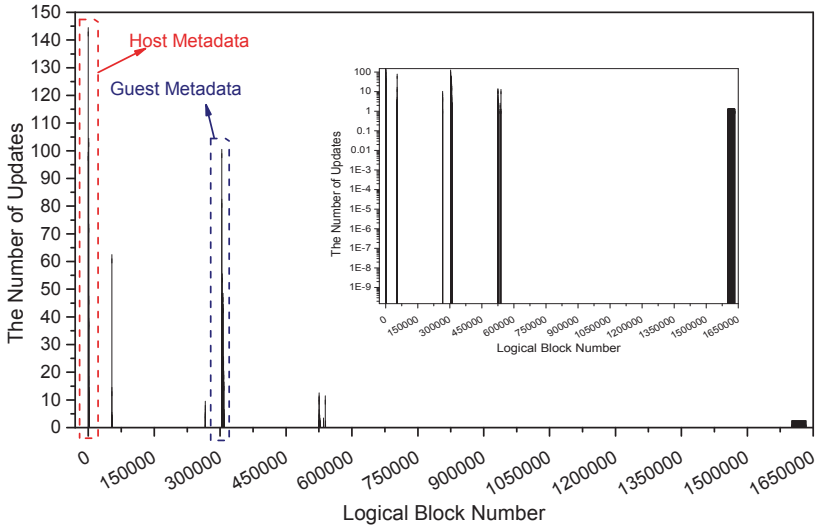


Figure 4.7. The number of the logical block updates when running the facebook application.

4.3 Motivation

4.3.1 Metadata Effect

We utilize the ARM Cortex-A15 dual-core processor platform [2] with the original stock Android system 4.2 running in QEMU virtual machine to analyze the data access patterns, especially for metadata stored in the fixed lower address of the whole file system. Immediately after the preconditioning is complete, we set the facebook application to issue 1.65 million 512B sized write requests. We collect each write request and the collected data are periodically retrieved and written to a log file. Figure 4.7 shows the distributions of number of logical block updates for the facebook application. The X-axis represents the logical block number, and the Y-axis represents the calculated number of block updates. The graph has a smaller graph embedded, which presents the whole data range with a log scaled Y-axis.

Several important results can be observed from the graph. First, the maximum number of updates of host file system metadata is 143, which is about 101x more frequently than the facebook's average number of updates of 1.41. Meanwhile, the number of guest file system metadata updates is 103, which is more than 73x times frequent than that of the average number of updates. These numbers show that both the guest file system metadata and host file system metadata are updated much more frequently than that of the regular data.

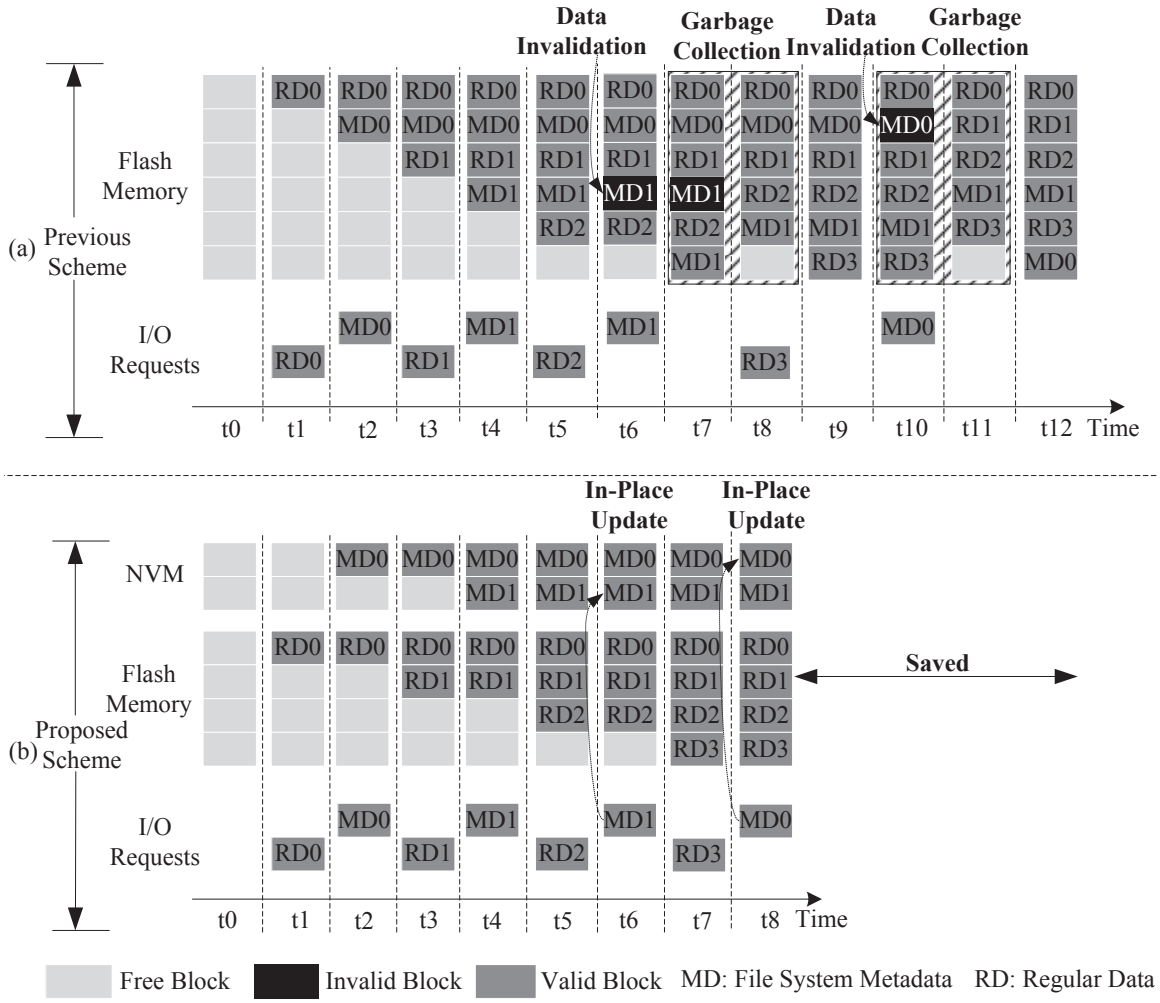


Figure 4.8. A motivational example of I/O optimization in mobile virtualization.

Second, the updated metadata size only occupies 0.42% of the whole captured write request size. That means the frequently updated metadata only occupy a very small portion of the write requests for the facebook application.

The frequent update of metadata is the intrinsic feature of the file system. Normally, each regular data write request is accompanied with a file system metadata write request, since the file system metadata are responsible for recording the changes of the regular data. Taking ext4 file system as an example, when a new file is created, an available inode, which is part of ext4 file system metadata, should be allocated for the new file. That is, the ext4 file system will select an available inode from the inode table, and then change the status of this

inode to be used. The changing status process is a filesystem metadata updating process.

4.3.2 Motivational Example

This work is motivated by the fact that frequent updates of the file system metadata in a VM image file do harm to the NAND flash based storage media in mobile devices. The objective of this work is to reduce the negative effects of file system metadata on NAND flash based storage media so as to improve the system performance and prolong the lifetime of NAND flash based storage media. Therefore, we propose to classify the file system metadata from the VM image file, and utilize the non-volatile memory to store these metadata to eliminate the metadata effects on NAND flash memory.

A motivational example is illustrated in Figure 4.8. In this example, we assume that all I/O requests are write requests, as we mainly concern the frequent update effects caused by file system metadata in this chapter. We divide the I/O requests into two kinds of requests, file system metadata (MD) requests and regular data (RD) requests. The MDi and RDi ($i \geq 1$) denote the ith file system metadata request and regular data request issued from a VM, respectively, where i represents the request address. If the addresses of two requests are the same, the later coming request is an update request of the early coming request, such as MD0 at time $t2$ and $t10$ in Figure 4.8 (a).

Figure 4.8 (a) shows the default scheme used in the original mobile system. This scheme does not use non-volatile memory to cache the file system metadata in a VM image file. Figure 4.8 (b) illustrates our proposed scheme that uses non-volatile memory to cache the file system metadata. Normally, each regular data write request is accompanied with a file system metadata write request, since the file system metadata are responsible for recording the changes of the regular data.

From Figure 4.8 (a), we can observe that the frequent updates of metadata will incur block invalidation and garbage collection operations in NAND flash memory. These two operations seriously affect the system response time, especially the garbage collection operation. Garbage collection operation contains two parts including moving the valid data into an

empty block, and erasing the invalid block. It is particularly time consuming if the cleaning block contains many valid data. Metadata usually updates a small portion of blocks and lead to a lot of valid data. For example, in Figure 4.8 (a), no valid space exists at time $t8$, since metadata MD1 is updated, resulting in generating the invalid data at time $t6$. When the I/O request RD3 is issued to NAND flash memory at time $t8$, the garbage collection is triggered to reclaim the invalid data MD1. At that time, in order to perform the erase operation, five valid data (RD0, RD1, RD2, MD0 and MD1) are required to move to a new empty block, which consumes a lot of time.

Therefore, in order to eliminate the frequent update effect introduced by metadata, we propose to utilize the non-volatile memory to store the file system metadata in a VM image file. As shown in Figure 4.8 (b), the file system metadata MD0 and MD1 are stored in the non-volatile memory, while the regular file data are kept in flash memory. Since non-volatile memory supports in-place update, all the updates of file system metadata can be efficiently tackled by non-volatile memory. As a result, we can effectively improve the system performance and prolong the NAND flash lifetime with the proposed scheme.

Although the I/O performance can be enhanced by putting the frequent-update metadata in the PCM memory, this method can introduce the lifetime degradation of the PCM memory. However, compared with the NAND flash memory, the lifetime of PCM is much longer. For example, the lifetime of PCM is 106$\sim$108, while the lifetime of NAND flash memory is 104$\sim$105. Here, we use PCM as one kind of NVM, for the other kinds of NVM (e.g. STT-RAM), the lifetime of other NVMs is even longer than that of PCM. So, normally, the lifetime of PCM is more than 100 times longer than that of NAND flash memory. Meanwhile, our experimental results show that the update frequency of metadata is about 100 times more than that of the regular data in the mobile virtualization environment. So, putting the frequent-update metadata in NVM does not lead to PCM being worn out much earlier than that of the NAND flash storage, and it is reasonable to put the frequent-update metadata in PCM for the lifetime and performance enhancement for the NAND-flash-based storage system.

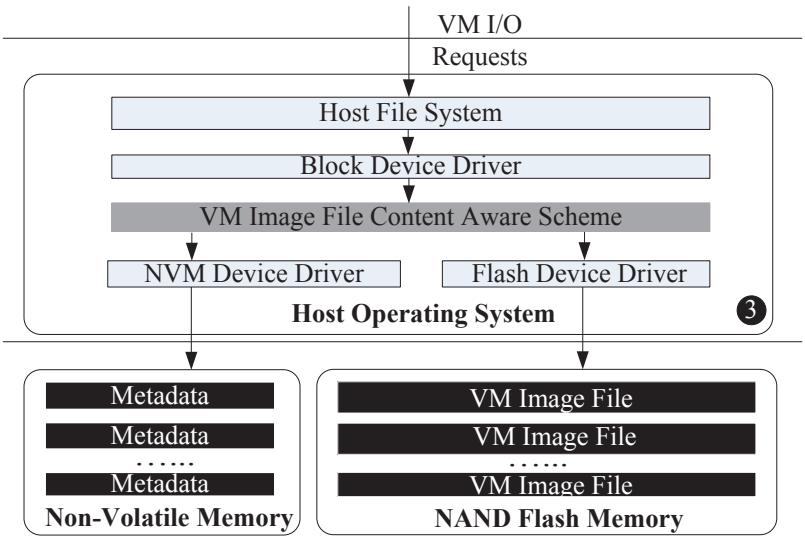


Figure 4.9. Three possible ways of integrating NVM into virtualization environment.

4.4 VM Image File Content Aware Design

In this section, we introduce the proposed scheme to effectively avoid the frequent update of file system metadata in a VM image file. We first give an overview in Section 4.4.1, and then present the proposed scheme in Section 4.4.2 and Section 4.4.3.

4.4.1 Overview

In order to achieve the results shown in the motivational example, we propose to store frequently updated metadata in the non-volatile memory to avoid the bad effects (long time garbage collection and lifetime degradation) to NAND flash memory. When a virtual machine issues an I/O request, the proposed scheme will use a metadata monitor to check the data type based on the address of the request. If the data are file system metadata, these data will be stored or fetched from non-volatile memory. On the other hand, the regular data will bypass to NAND flash memory. As a result, the I/O response time can be improved and the NAND flash lifetime can be prolonged.

By utilizing NVM to optimize the I/O performance, we need to integrate NVM in

the current mobile virtualization environment. There are three possible ways of integrating NVM as shown in Figure 4.1. In the first method, NVM can be directly operated by the guest operating system. This method involves modifications of the guest operating system, virtual machine, and host operating system. These modifications can be summarized as follows. From the guest operating system's perspective, each guest operating system should be modified to support the proposed scheme. Meanwhile, virtual machine should provide the emulated NVM device, and host operating system should expose the NVM access interface, such as *NVM_Read* and *NVM_Write*. So, these modifications are expensive, and even impossible by considering the real implementation. For the second method, NVM is managed by virtual machine. This method also requires a lot of software modifications in terms of modifying the virtual machine and the host operating system. Virtual machine should be renovated to support the proposed scheme, while the host operating system should provide the NVM access interface.

Therefore, in order to reduce the software modifications, it is more desirable to integrate the proposed scheme into the host operating system. More specifically, we propose to put the proposed scheme at the bottom of the I/O stack, lying beneath the standard block device driver in host operating system as shown in Figure 4.9. Compared with the first two usage methods, the last management method is transparent to upper layers in the I/O stack. The design is implemented in both the host OS and the virtual machine layer. In the virtual machine layer, we propose a simple but effective method to identify the guest OS file system metadata. More specifically, we use a padding flag to denote the data type. In the host OS file system, with the help of the padding flag, we design and implement the proposed VM image file content aware scheme. By using this method, we can effectively avoid substantial software modifications. It is also the choice of this work. We will discuss the details of the proposed scheme which will achieve the desired goals.

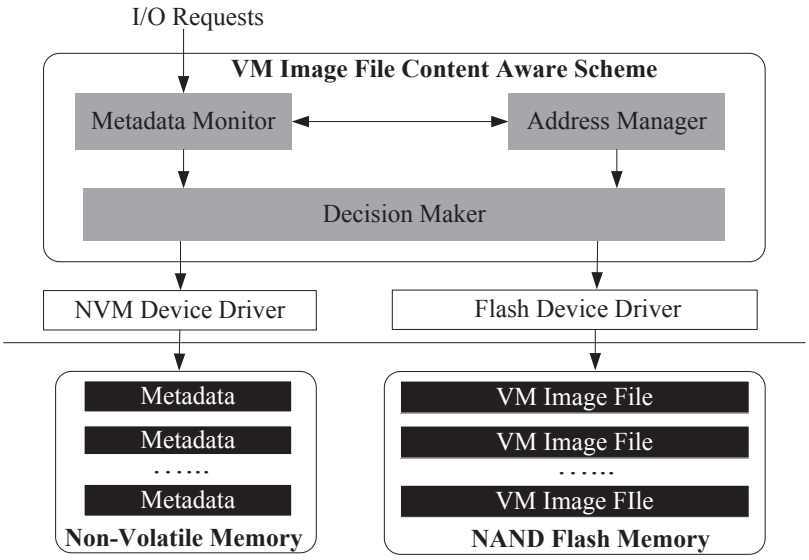


Figure 4.10. VM image file content aware scheme.

4.4.2 VM Image File Content Aware Scheme

As shown in Figure 4.10, the proposed scheme consists of three components (metadata monitor, decision maker and address manager). These three components interact with each other and play different roles in the proposed scheme. The function of metadata monitor is to classify the file system metadata and regular file data in a VM image file. Decision maker determines the storing media, NAND flash memory or non-volatile memory. Address manager is responsible for managing the storage usage of non-volatile memory.

We have introduced the function of the three components in the VM image file content aware scheme; and we will describe the detailed working procedures of these three components. The metadata monitor identifies the location of the file system metadata in the flash storage device by analyzing the metadata layout in a VM image file during the system start-up. Therefore, whenever the metadata monitor receives an I/O request, it should be able to distinguish metadata from regular file data according to the accessed address issued from a VM. If the accessed address belongs to the regular data, the regular data can be directly bypass to NAND flash memory. On the other hand, metadata will be delivered to decision maker. Decision maker will consult the address manager and allocate the required address

to the metadata requests.

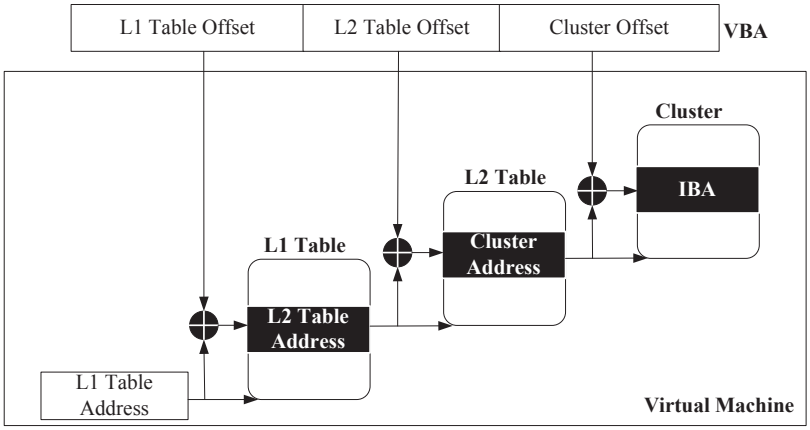


Figure 4.11. Two level address mapping in qcow2 image file.

The metadata monitor is responsible for classifying the file system metadata from regular file data. However, it is difficult for the host OS file system to identify the guest file system metadata mainly because of lacking of the VM context in the host file system. The reason for this context loss is that the guest metadata address is translated to the regular image file block address via a two level mapping table as shown in Figure 4.11. So, in the host file system, it is difficult to identify the guest file system metadata according to the translated address. The two levels of tables are called L1 table and L2 table. There is only one L1 table, and each entry of the L1 table links to an L2 table. Each entry of an L2 table points to a cluster, which is the unit for space allocation in an image and contains multiple blocks (e.g., a 64KB unit consists of sixteen 4KB blocks). Accordingly, a VBA consists of three components: L1 table offset, L2 table offset, and cluster offset. L1 table offset and L2 table offset indicate specific entries in the corresponding L1 and L2 tables, respectively, and the cluster offset specifies the block in the corresponding cluster, or the IBA the VBA is mapped to. These two mapping tables translate the virtual block address (VBA) into an image block address (IBA), which is actually also a logic block address in the disk image. Therefore, it is difficult to directly identify the metadata based on the translated VM I/O request address.

4.4.3 Metadata Identification

Metadata Identification in the Guest File System

One key observation here is that the metadata and regular data can still be identified before VBA is translated. Therefore, we can use techniques proposed in [96] to identify metadata and regular data before the address is translated.

Algorithm 3: Metadata identifier

Input: VBA: The virtual block address

Output: DT: The data type

```
1  MT ← the table recordered the metadata address
2  DT ← Regular data
3  counter ← 0
4  while MT[counter] is available do
5      MVBA ← The virtual address in MT[counter]
6      MLen ← The data length in MT[counter]
7      if MVBA ≤ VBA ≤ (MVBA + MLen) then
8          DT ← Metadata
9          Return
10     counter ← counter + 1
```

Algorithm 3 describes the metadata identifier used to identify I/O requests as metadata requests or regular requests. The metadata identifier maintains a run length structure for each metadata file in the main memory. Each I/O request contains information like $MVBA$ and $MLength$, where $MVBA$ is the first virtual block address belonging to the metadata, and $MLength$ is the number of consecutive $MVBA$. By comparing the request address, the metadata identifier determines the request type, which is a metadata request or a regular data

request. If the request is to access the content of a file or directory, the request is identified as the regular data request. Although many popular commercial file systems seldom change the location of their metadata, the proposed metadata identifier analyzes and updates the changed VBA regions of metadata during the systems startup, so as to prevent the loss of accuracy in metadata tracking in the long run and to prevent the adoption of the costly online monitoring mechanism. In the ext file system, all of the VBA regions of metadata could be reconstructed by parsing its Superblock and Group Description Table, because the layout of an ext2 partition is determined when it is formatted [96]. Each block group usually consists of the same number of blocks with the same number of metadata blocks in the beginning of the block group, and each block group is allocated one after another. Therefore, the *MVBA* region of the metadata in each block group could be easily derived by several arithmetic operations, instead of maintaining a run-length structure for each block group. Once the Superblock or the root sector of the storage is modified, the calculation parameter for the *MVBA* region of metadata in each block group could be revised accordingly.

Metadata Information Delivered in the Virtual Machine

After the data are successfully identified, we modify the VM and append a padding flag P in the I/O request as shown in Figure 4.12. Then, when metadata monitor receives an I/O request, the metadata monitor can identify the metadata based on the appended padding information.

Figure 4.12 shows the padding flag strategy to classify metadata from regular data. We add a metadata identifer in virtual machine. From [96], we know that both metadata and regular data are located in fixed, well-known VBA. By looking at VBA, we can easily identify them and use metadata identifer to give different padding flags, such as 1 for metadata and 0 for regular data. The detailed working procedure is discussed in the following part. When I/O requests are delivered to the VM, the VBA will be translated to an image block address (IBA) with the two level mapping table. After the IBA is found with the help of the two level mapping table, the padding flag will be appended by metadata identifer. Then, both

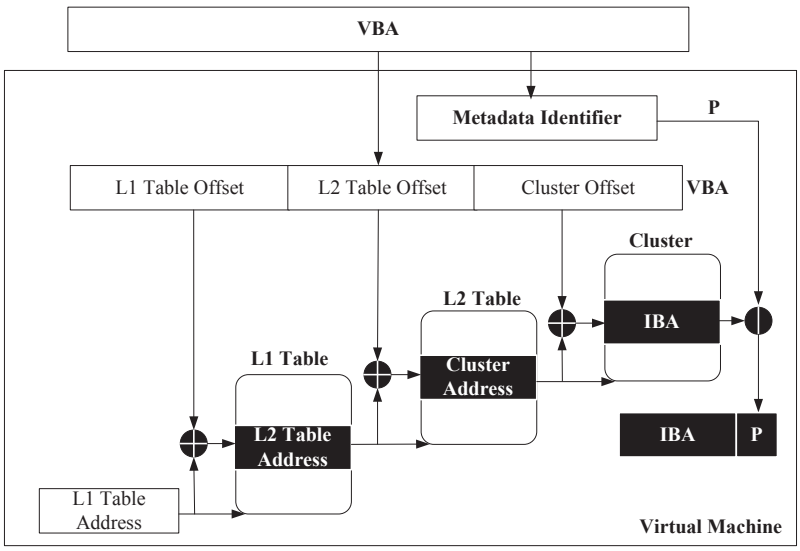


Figure 4.12. Padding flag strategy to classify metadata from regular data.

the IBA and padding flag P will be transferred to host operating system. After VBA with padding flag P is transferred to the VM image file content aware layer, metadata monitor can utilize the padding flag P to check whether this data is metadata or regular data.

4.5 Performance and Overhead Analysis

In this section, we analyze the system performance of the proposed scheme by comparing it with the original mobile virtualization. The system response time is an important metric to evaluate the system performance. It is the time period from the point when an operation is issued to the point when the operation has been completed. The inputs of an I/O request are read and write operations, and the basic read and write unit is one page. Thus, we conduct the analysis for the best-case and worst-case system response times of read and write operations. The symbols used in this analysis are listed below.

T_{flash_rd}	The time to read one page in flash memory
T_{flash_wr}	The time to write one page in flash memory
T_{flash_erase}	The time to erase a block in flash memory
T_{NVM_rd}	The time to read one page size data in NVM
T_{NVM_wr}	The time to write one page size data in NVM
T_{ecc_en}	The time to encode data in flash memory
T_{ecc_de}	The time to decode data in flash memory
N_{max_vpage}	The maximum number of valid pages in one block
N_{gc}	The number of garbage collection operations
N_{md_wr}	The number of metadata write requests
N_{md_rd}	The number of metadata read requests

The extra performance degradation is first reflected to garbage collection operations, including extra valid page copies and block erasures, and the time is:

$$((T_{flash\_rd} + T_{flash\_wr}) \times N_{max\_vpage} + T_{flash\_erase}) \times N_{gc} \tag{4.1}$$

$$T_{flash\_erase} \times N_{gc} \tag{4.2}$$

In the mobile virtualization environment, the frequent metadata updates in a VM image file leads to generate many invalid pages in NAND flash memory. The generation of these invalid data finally results in triggering the garbage collection operation, which needs to copy the valid pages and erase the dirty blocks. The worst-case and best-case overhead of garbage collection operations can be calculated through Equation (4.1) and Equation (4.2), respectively. The timing difference between the worst-case and best-case is reflected from the valid page copy procedure. If one block contains maximum number of valid pages denoted by N_{max_vpage}, the worst-case will occur and plenty of valid pages are needed to copy from dirty blocks as shown in Equation (4.1). On the other hand, if a block has no valid pages, only block erasure operations occur as described in Equation (4.2).

$$(T_{flash\_wr} - T_{NVM\_wr}) \times N_{md\_wr} + (T_{flash\_rd} - T_{NVM\_rd}) \times N_{md\_rd} \tag{4.3}$$

$$T_{ecc\_en} \times N_{md\_wr} + T_{ecc\_de} \times N_{md\_rd} \tag{4.4}$$

The proposed scheme not only reduces the garbage collection overhead but also can enhance the read and write speed. Equation (4.3) describes the read and write performance gain by utilizing NVM. N_{NVM_wr} stands for the number of NVM write requests, and N_{NVM_rd} represents the number of NVM read requests. Without the proposed NVM integration scheme, these requests are issued to flash devices. Given that the I/O speed of flash memory is orders of magnitude slower than that of NVM, we can achieve great performance gain for I/O operations as shown in Equation (4.3). In addition, the proposed scheme can further improve the data coding time. Since the reliability of NVM is much better than that of NAND flash memory, metadata stored in NVM usually use simple coding or non-coding method to guarantee the data reliability. The proposed scheme stores the metadata in NVM resulting in eliminating the encoding/decoding time for these metadata as shown in Equation (4.4).

We use an example to illustrate the benefits of the proposed scheme. Assume that the flash page size is 4KB and each block contains 512 pages. Given that the flash page read and write latency are *25us* and *500us*, respectively, and the block erase time is *1.5ms*. For the NVM, we use the parameters of PCM as an example to quantitatively analyze the performance improvement, and the read latency and write latency for accessing one cache line size (e.g. 64bytes) data are configured to be *50ns* and *1us*, respectively. If 800 garbage collection operations are reduced, the performance improvement with the garbage collection reduction is $((0.025ms+0.5ms)\times 63+1.5ms)\times 800 = 27660ms$ in the best case, and $800\times 1.5ms = 1200ms$ in the worst case. If 50,000 metadata read requests and 50,000 metadata write requests are received, the response time can be reduced by $(0.5ms - 0.001ms \times 64) \times 50,000 + (0.025ms - 0.00005ms \times 64) \times 50,000 = 22890ms$.

4.6 Evaluation

In this section, we present our experimental results with analysis. We compare and evaluate the proposed scheme with the baseline scheme in terms of three metrics: the lifetime, the I/O performance and the energy consumption. The baseline scheme is the original stock Android 4.2 system. The performance evaluation is conducted on a real Android based VM platform with a Samsung ARM Cortex-A15 dual-core processor and a 64 Gb NAND flash memory chip.

4.6.1 Experimental Setup

We conducted experiments on a real Android platform [3]. The evaluation platform adopts an ARM Cortex-A15 dual-core processor (Samsung Exynos5250 [2]) with ARMv7 architecture. In this platform, the ARM processor core runs at 1.7GHz, and it consists of a 32 KB instruction cache and a 32 KB data cache. The platform adopts Android 4.2 with the Linux kernel 3.9. The core board is equipped with 64 Gb of eMMC flash memory, 128Gb SD card and 2GB DDR3 RAM. We use a small portion of RAM to simulate PCM, since the I/O accessing patterns of PCM are similar to RAM [39]. That is, RAM and PCM are all byte addressable and support in-place update. The physical interfaces, such as the USB port, are designed in the mother board, and we connect a WiFi module through a pin to pin connector. Also, one pin connector is used to connect the core board with the mother board.

The scheme is implemented in the host operating system. In evaluation, QEMU emulates the hardware devices, and the popular used mobile operating system, Android 4.2 with the Linux kernel 3.9, is running on the QEMU. When an application accesses the storage in QEMU, QEMU issues the I/O requests to the host operating system. Host operating system can issue read/write operations to the flash device driver, which can control the NAND flash memory chip and SD Card. We utilize the universal serial device driver to obtain and output the experimental results. For fair comparisons, the same configuration has been adopted for both the baseline scheme and our proposed scheme.

Table 4.2. The Android applications and their usage.

Applications	**Using Scenarios**
Google Earth	Search Places and View Streets
Browser	Browse Websites and Search Pictures
WeChat	Chat with Friends
YouTube	Watch Online Movies
Facebook	View Friends' Status
Gmail	Receive Emails and Send Emails
Gallery	View Pictures
Download	Download Applications
Install	Install Applications
Media Player	Watch Movies and Listen to Music
Angry Birds	Play the Game and Select Levels

We use real Android applications as benchmarks to evaluate the effectiveness of the proposed scheme. These applications are running in the guest Android operating system. The applications and their using scenarios are shown in Table 4.2. These applications are typical operations in our daily lifetime. They can be used to accelerate the evaluation process, and evaluate the I/O performance for mobile virtualization. We test each application on the evaluation platform. The applications iteratively issue requests to the storage system. In the experiments with 2 and 3 VMs, these VMs are running simultaneously, which can help understand the scalability of the proposed design.

4.6.2 PCM and NAND Flash Memory Models

We utilize PCM as an example of NVM to study the benefits of the proposed scheme. The response time of NAND flash device is directly measured with the help of the Linux kernel function $do\_gettimeofday$. The other results are obtained by capturing I/O requests to PCM and NAND flash memory and calculating with the models of NAND flash memory and PCM. The read latency and write latency of PCM are configured to be $50ns$ and $1us$, respectively. In order to study the energy consumption, the read and write energy of PCM are configured to be $1J/GB$ and $6J/GB$, respectively. The read and write energy of NAND flash memory are set to $1.5J/GB$ and $17J/GB$, respectively. NAND flash memory also supports the erase operation which is not available in PCM. The energy consumption of the erase operation is configured to be $10J/GB$ [16, 25, 39, 53, 101].

PCM also has much better endurance than that of NAND flash memory. Each PCM cell can sustain $10^6 - 10^8$ writes before a failure occurs, while NAND flash memory can only support $10^4 - 10^5$ program/erase operations. That is why we can utilize PCM to improve the NAND flash memory lifetime.

4.6.3 Results and Discussion

In this section, we present the experimental results with analysis. We first present the improvement in performance by comparison of the baseline scheme and the proposed scheme. Then, we discuss the lifetime enhancement for flash memory. Finally, we analyze the I/O energy consumption for the two schemes.

The performance and power models of NAND flash memory and PCM are presented in Section 4.6.2. We get the experimental results by capturing and calculating the number of I/O requests to PCM and NAND flash memory, respectively as shown in Table 4.3 and Table 4.4. Then, we can obtain the I/O performance and I/O energy consumption with the help of PCM and NAND flash memory models.

Table 4.3. The I/O requests with different numbers of VMs (part 1).

Applications	**I/O Requests**				
	All Read (1 VM)	All Write (1 VM)	PCM Read (1 VM)	PCM Write (1 VM)	Avg. Req. Size (KB)
Google Earth	33,272	122,301	26,625	97,842	4.91
Browser	140,423	13,652	98,301	9,562	21.53
WeChat	1,123	124,215	995	113,722	4.23
YouTube	2,787	3,100	2,240	2,040	8.35
Facebook	38,831	41,242	31,074	32,919	6.85
Gmail	1,652	55,110	1,132	41,408	4.26
Gallery	32,210	878	25,568	745	89.65
Download	7,318	7,344	5,790	5,995	232.94
Install	9,725	24,622	7,291	16,475	145.95
Media Player	15,260	5,024	12,221	4,023	12.55
Angry Birds	7,791	522	6,241	390	58.32
Applications	All Read (2 VMs)	All Write (2 VMs)	PCM Read (2 VMs)	PCM Write (2 VMs)	Avg. Req. Size (KB)
Google Earth	22,582	82,994	18,072	66,393	4.85
Browser	94,851	9,222	75,488	7,438	20.56
WeChat	810	92,582	642	74,064	4.23
YouTube	1,889	1,908	1,512	1,581	8.56
Facebook	29,113	30,942	23,301	24,775	6.23
Gmail	1,072	35,681	860	28,542	4.89
Gallery	22,377	612	17,904	429	100.23
Download	4,392	4,435	3,521	3,524	212.53
Install	6,901	17,424	5,521	13,596	156.23
Media Player	7,747	2,543	6,195	2,023	15.63
Angry Birds	5,779	365	4,635	210	68.56

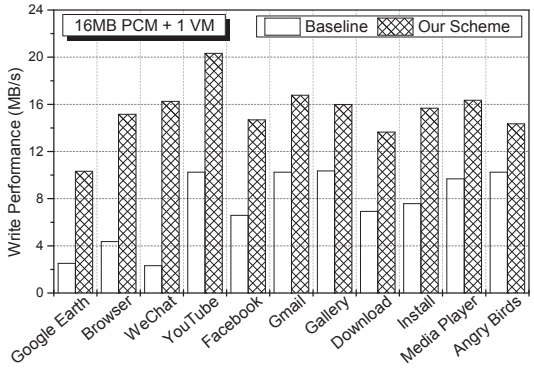
Table 4.4. The I/O requests with different numbers of VMs (part 2).

Applications	All Read (3 VMs)	All Write (3 VMs)	PCM Read (3 VMs)	PCM Write (3 VMs)	Avg. Req. Size (KB)
Google Earth	9,342	34,312	7,457	27,465	5.62
Browser	51,354	4,989	41,081	3,992	22.35
WeChat	351	35,373	252	28,306	5.63
YouTube	772	805	621	684	8.79
Facebook	23,234	24,667	18,587	19,714	5.94
Gmail	426	15,259	367	12,213	4.56
Gallery	18,402	506	14,742	410	112.35
Download	2,878	2,903	2,320	2,132	256.32
Install	3,510	8,814	2,802	7,083	186.32
Media Player	3,972	1,310	3,185	1,052	18.62
Angry Birds	2,791	272	2,235	142	75.62

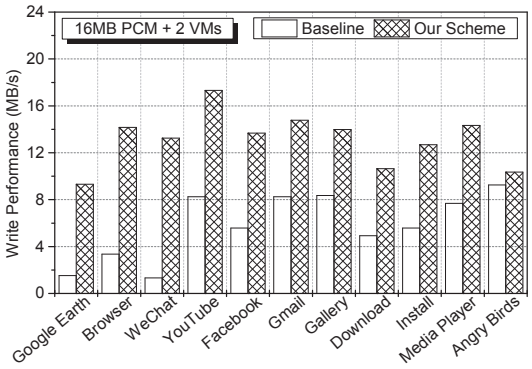
Performance Improvement

Figure 4.13 illustrates the write performance and read performance of the baseline and the proposed scheme. We run one to three VMs to study the effects with running different number of VMs in mobile devices.

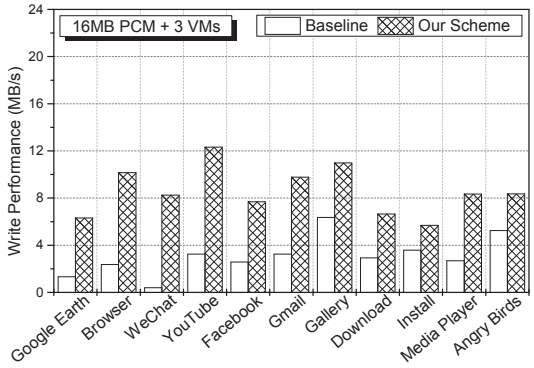
From the experimental results, we can discover that the read and write performance can be effectively improved compared with the baseline scheme. We take applications running with one VM as an example (shown in Figure 4.13 (a) and Figure 4.13 (d)) to illustrate the benefits of the proposed scheme. Compared with the baseline scheme, the proposed scheme can improve the write and read performance to more than 45.21% and 12.48% on average, respectively. In addition, with the number of VMs increase (from one to three), the read and write performance decrease in both the baseline scheme and the proposed scheme. This phenomenon is introduced by the mobile resources preemption between different VMs, which significantly degrades the system performance. However, the proposed can still work



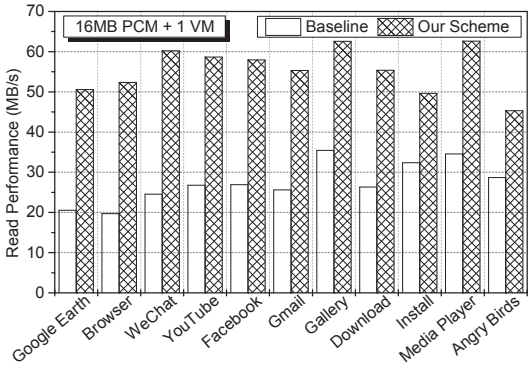
(a) Write performance with one VM.



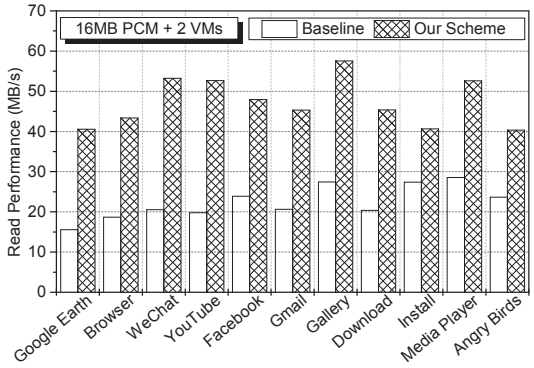
(b) Write performance with two VMs.



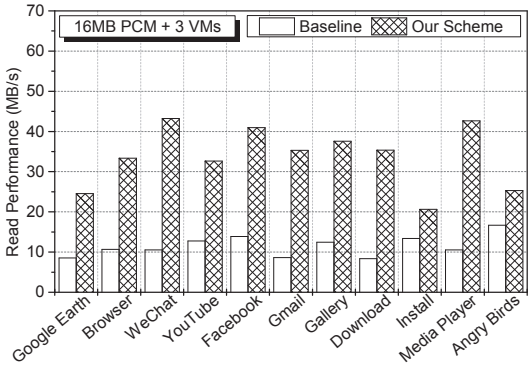
(c) Write performance with three VMs.



(d) Read performance with one VM.



(e) Read performance with two VMs.

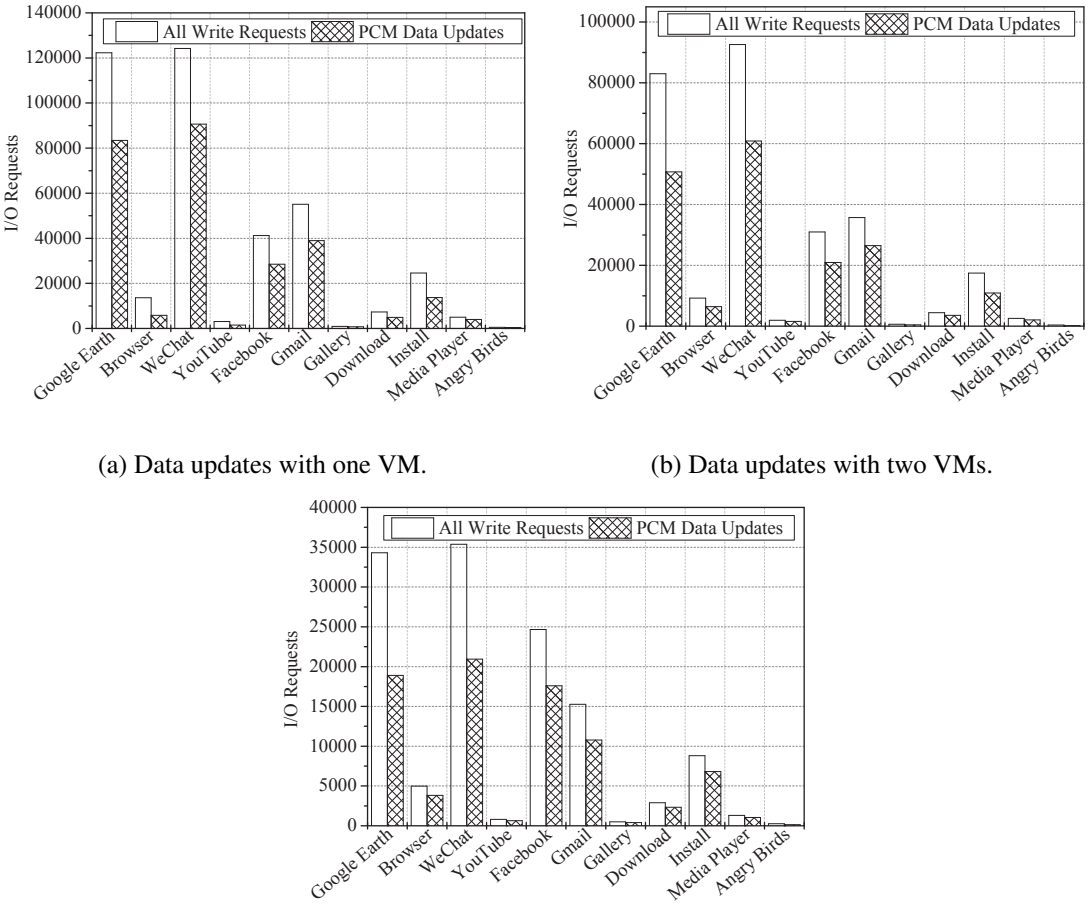


(f) Read performance with three VMs.

Figure 4.13. The I/O performance with different number of VMs.

better than the baseline scheme even working with more VMs as shown in Figure 4.13. All these good performance achievements benefit from storing the file system metadata in PCM, since PCM has better I/O performance and supports in-place update compared with NAND flash memory.

Lifetime Enhancement



(a) Data updates with one VM.

(b) Data updates with two VMs.

(c) Data updates with three VMs.

Figure 4.14. Comparison of all write requests and data updates in PCM with different number of VMs.

The NAND flash lifetime is mainly affected by data updates, since data updates generate invalid pages, and lead to more block erasures. Figure 4.14 shows the number of data updates in PCM with running different number of VMs. In the figure, the white bar stands for the number of write requests to flash memory and PCM, while the bar with cross lines represents the number of data updates in PCM. From the figure, the proposed scheme can effectively reduce the number of data updates to NAND flash memory, and thus prolong the lifetime of flash memory. We take the Google Earth application running with one to three VMs as an example to show the effectiveness of the proposed scheme. Compared with the baseline scheme, the proposed scheme can reduce 63.88% data updates on average.

For all the evaluated applications, the proposed scheme can reduce 67.03% data updates on average. These experimental results demonstrate that a large number of write requests is to update metadata, and the proposed scheme can effectively reduce the data updates to NAND-flash-based mobile devices.

IO Energy Consumption

Table 4.5. The I/O energy consumption with different numbers of VMs.

Applications	**I/O Energy** (mJ)					
	Baseline (1 VM)	Our Scheme (1 VM)	Baseline (2 VMs)	Our Scheme (2 VMs)	Baseline (3 VMs)	Our Scheme (3 VMs)
Google Earth	39,976.99	19,469.08	26,880.12	13,051.03	12,835.64	6,248.78
Browser	36,390.66	23,685.27	23,684.58	14,077.31	14,058.57	8,303.45
WeChat	34,221.20	13,907.76	25,416.23	12,264.84	12,946.16	6,236.32
YouTube	1,811.80	1,061.35	1,171.68	559.11	507.70	235.00
Facebook	19,872.60	9,974.46	13,568.85	6,785.26	10,341.63	5,167.28
Gmail	15,284.98	7,853.82	11,345.00	5,480.38	4,523.44	2,183.34
Gallery	21,627.64	14,453.08	16,841.61	11,584.54	15,546.78	10,424.81
Download	120,713.49	59,522.57	66,516.70	33,611.97	52,505.67	28,410.47
Install	241,294.86	138,237.27	182,770.31	91,924.22	110,300.14	53,867.36
Media Player	5,214.71	2,773.58	3,290.45	1,758.96	2,045.03	1,069.96
Angry Birds	4,574.16	2,925.52	3,919.95	2,679.69	2,561.54	1,768.59

The energy consumption is another important metric to measure the effectiveness of the proposed scheme. With the versatile mobile applications appearing in mobile devices, people have more and more relied on mobile devices, and the energy consumption becomes an important issue for battery-driven mobile devices.

Table 4.5 summarizes the measured energy consumption. From the experimental

results, we can discover that the proposed scheme can achieve less I/O energy consumption compared with the baseline scheme. For example, compared with the baseline scheme, the proposed scheme can reduce about 44.99% I/O energy consumption on average when one VM is running. With increasing the number of VMs (from one VM to three VMs), the I/O energy consumption decreases for both the baseline scheme and the proposed scheme. This is because we get the experimental results by running the applications with the same time period. The decreasing of the I/O performance leads to less I/O requests generated from the VMs. The energy reduction for the proposed scheme benefits from two aspects. The first aspect is that the I/O energy consumption of PCM is lower than that of NAND flash memory. By storing metadata in PCM, we can directly reduce the I/O energy consumption. The second aspect is indirectly reflected by reducing the garbage collection operation. Garbage collection includes valid pages copy and block erasure operations that both need to consume a lot of energy. By storing frequently updated metadata in PCM, we can reduce the garbage collection operations, and thus can effectively save the energy consumption.

4.7 Summary

In this chapter, we propose an image-content-aware scheme to improve the system performance, and prolong the lifetime for NAND-flash-based mobile devices. The basic idea of the proposed scheme is to analyze the VM image file content, and store both the guest and host file system metadata in small but faster and endurable non-volatile memory. We conduct experiments on a set of representative Android applications in an embedded development board. The experimental results show that the proposed scheme not only improves the system performance for mobile virtualization, but also extends the lifetime, and reduces the I/O energy consumption for mobile devices.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

In this thesis, we investigate emerging storage techniques for I/O optimization in mobile devices, which can provide comprehensive solutions and generate optimal storage optimization for resource-constrained mobile devices. Specifically, we proposed three schemes to optimize I/O performance.

- For the first scheme, we propose a scheme, called DHeating, to solve the concentrated heating problem, enhance the reliability, and eliminate the lengthy heating operation for self-healing NAND flash memory. We conduct experiments on a set of realistic I/O workloads collected from our embedded development board. The experimental results show that the proposed scheme not only solves the concentrated heating problem for NAND flash memory, but also improves the average system performance and enhances the reliability of the self-healing flash memory.

- For the second scheme, we propose a unified NVM and flash memory architecture to improve the I/O performance for mobile devices. We also propose a software, called vFlash, to manage the novel architecture. The basic idea of vFlash is to cross-layer transparent utilize NVM to minimize the application modifications and maximize the I/O performance for NAND-flash-based mobile devices. We conduct experiments on a set of I/O workloads of Android applications running in a representative hardware platform. The experimental results show that the proposed scheme can greatly enhance the I/O performance.

- For the third scheme, we propose a VM image file content aware scheme to improve the system performance and extend the life time for NAND-flash-based mobile devices. The basic idea in our scheme is to analyze the VM image file format, and store guest OS metadata and hot regular data in faster and more endurable NVM. We conduct experiments on a set of Android applications in an embedded development board. The experimental results show that the proposed scheme not only improves the system performance for mobile virtualization, but also greatly reduces the data updates to NAND flash memory.

5.2 Future Work

The work presented in this thesis can be extended in different directions in the future.

- First, we will continue to study self-healing NAND flash memory. Since the data access pattern of applications may influence the endurance and heating of self-healing flash memory, how to effectively identify the characteristics of the access pattern of applications and employ this access patter to guide the design of the self-healing flash management scheme is a possible topic for exploration.

- Second, we will continue to study the non-volatile memory technologies and apply these technologies to the other fields. The non-volatile memory based processor [89–91] is one direction that we can explore.

- Third, the newly emerged other memory technologies, such as memristor [7,57,58,61, 82,92], racetrack memroy [98,99,102], and scratch pad memory [45], present different features. It is also an interesting topic to research these different features and utilize these memory technologies to serve the current computer system.

- Finally, we will continue to investigate the performance and energy issues in the mobile virtualization environment. The mobile virtualization environment presents many

new challenges in effectively running multiple operation systems in the resource-constrained mobile systems. How to optimize the virtualization to satisfy the mobile devices would be an important direction for us to explore.

REFERENCES

[1] The QCOW2 image format. *https://people.gnome.org/ markmc/qcow-image-format.html*, 2008.

[2] Exynos5250. *http://www.samsung.com/global/business/semiconductor/file/product/Exynos_5_Dual_User_Manaul_Public_REV1.00-0.pdf*, 2012.

[3] Arndale board exynos5250. *http://www.arndaleboard.org/wiki/index.php/Main_Page*, 2014.

[4] Ext4 disk layout. *https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout*, 2014.

[5] Everything you need to know about slc, mlc, & tlc nand flash. *http://www.mydigitaldiscount.com/everything-you-need-to-know-about-slc-mlc-and-tlc-nand-flash.html*, 2015.

[6] YAFFS: Yet another flash file system. *http://www.yaffs.net/*, 2016.

[7] H. Akinaga and H. Shima. Resistive random access memory (ReRAM) based on metal oxides. *Proceedings of the IEEE*, 98(12):2237–2251, 2010.

[8] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference (ATC '05)*, pages 41–46, 2005.

[9] Simona Boboila and Peter Desnoyers. Write endurance in flash drives: Measurements and analysis. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST '10)*, pages 1–14, 2010.

[10] Yu Cai, E.F. Haratsch, O. Mutlu, and Ken Mai. Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis. In *Design, Automation Test in Europe Conference Exhibition (DATE '12)*, pages 521–526, 2012.

[11] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*, pages 385–395, 2010.

[12] Bing-Jing Chang, Yuan-Hao Chang, Hung-Sheng Chang, Tei-Wei Kuo, and Hsiang-Pang Li. A pcm translation layer for integrated memory and storage management. In *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '14)*, pages 6:1–6:10, 2014.

[13] Da-Wei Chang, Hsin-Hung Chen, Dau-Jieu Yang, and Hsung-Pin Chang. BLAS: Block-level adaptive striping for solid-state drives. *ACM Transactions on Design Automation of Electronic Systems*, 19(2):21:1–21:29, 2014.

[14] Li-Pin Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC '07)*, pages 1126–1130, 2007.

[15] Li-Pin Chang and Li-Chun Huang. A low-cost wear-leveling algorithm for block-mapping solid-state disks. In *Proceedings of the 2011 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '11)*, pages 31–40, 2011.

[16] Li-Pin Chang, Yo-Chuan Su, and I-Chen Wu. Plugging versus logging: Adaptive buffer management for hybrid-mapping ssds. *ACM Transactions on Embedded Computing Systems (TECS '15)*, 14(2):29:1–29:21, 2015.

[17] Li-Pin Chang and You-Chiuan Su. Plugging versus logging: A new approach to write buffer management for solid-state disks. In *Proceedings of the 48th Design Automation Conference (DAC '11)*, pages 23–28, 2011.

[18] Yu-Ming Chang, Yuan-Hao Chang, Jian-Jia Chen, Tei-Wei Kuo, Hsiang-Pang Li, and Hang-Ting Lue. On trading wear-leveling with heal-leveling. In *Proceedings of the 51st Annual Design Automation Conference (DAC '14)*, pages 83:1–83:6, 2014.

[19] Yuan-Hao Chang, Jen-Wei Hsieh, and Tei-Wei Kuo. Endurance enhancement of flash-memory storage, systems: An efficient static wear leveling design. In *Proceedings of the 44th ACM/IEEE Design Automation Conference (DAC '07)*, pages 212 –217, 2007.

[20] Yuan-Hao Chang, Jen-Wei Hsieh, and Tei-Wei Kuo. Improving flash wear-leveling by proactively moving static data. *IEEE Transactions on Computers*, 59:53 –65, 2010.

[21] Yuan-Hao Chang, Po-Liang Wu, Tei-Wei Kuo, and Shih-Hao Hung. An adaptive file-system-oriented FTL mechanism for flash-memory storage systems. *ACM Transactions on Embedded Computing Systems*, 11(1):9:1–9:19, 2012.

[22] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '09)*, pages 181–192, 2009.

[23] Yiran Chen, Hai Li, Xiaobin Wang, Wenzhong Zhu, Wei Xu, and Tong Zhang. A 130 nm 1.2 V/3.3 V 16 kb spin-transfer torque random access memory with nondestructive self-reference sensing scheme. *IEEE Journal of Solid-State Circuits*, 47(2):560–573, 2012.

[24] Ben Cheng and Bill Buzbee. A jit compiler for android's dalvik vm. In *Google I/O developer conference*, 2010.

[25] Sheng-Wei Cheng, Yu-Fen Chang, Yuan-Hao Chang, Hsin-Wen Wei, and Wei-Kuan Shih. Warranty-aware page management for pcm-based embedded systems. In *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD '14)*, pages 734–741, 2014.

[26] Y.-T. Chiu. Forever flash. *IEEE Spectrum*, 49(12):11 –12, December 2012.

[27] Citrix. Xenmobile. *https://www.citrix.com/products/xenmobile/overview.html*, 2015.

[28] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From ARIES to MARS: Transaction support for Next-generation, Solid-state Drives. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 197–212, 2013.

[29] Peter Desnoyers. Analytic models of ssd write performance. *ACM Transactions on Storage (TOS)*, 10(2):8:1–8:25, 2014.

[30] G. Dhiman, R. Ayoub, and T. Rosing. PDRAM: A hybrid PRAM and DRAM main memory system. In *46th ACM/IEEE Design Automation Conference (DAC '09)*, pages 664–669, 2009.

[31] Xiangyu Dong and Yuan Xie. AdaMS: Adaptive MLC/SLC phase-change memory design for file storage. In *16th Asia and South Pacific Design Automation Conference (ASP-DAC '11)*, pages 31–36, 2011.

[32] Samsung Electronics. K9LBG08U0M. *http://www.samsung.com*.

[33] Z. Fan, D. H. C. Du, and D. Voigt. H-ARC: A non-volatile memory based cache policy for solid state drives. In *30th Symposium on Mass Storage Systems and Technologies (MSST '14)*, pages 1–11, 2014.

[34] A. P. Ferreira, B. Childers, R. Melhem, D. Moss, and M. Yousif. Using PCM in next-generation embedded space applications. In *16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '10)*, pages 153–162, 2010.

[35] A. P. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, and D. Moss. Increasing PCM main memory lifetime. In *Design, Automation Test in Europe Conference Exhibition (DATE '10)*, pages 914–919, 2010.

[36] Congming Gao, Liang Shi, Kaijie Wu, C.J. Xue, and E.H.-M. Sha. Exploit asymmetric error rates of cell states to improve the performance of flash memory storage systems. In *2014 32nd IEEE International Conference on Computer Design (ICCD '14)*, pages 202–207, 2014.

[37] Yongqiang Gao, Haibing Guan, Zhengwei Qi, Yang Hou, and Liang Liu. A multi-objective ant colony system algorithm for virtual machine placement in cloud computing. *Journal of Computer and System Sciences*, 79(8):1230 – 1242, 2013.

[38] Yong Guan, Guohui Wang, Yi Wang, Renhai Chen, and Zili Shao. BLog: Block-level log-block management for nand flash memorystorage systems. In *Proceedings of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '13)*, pages 111–120, 2013.

[39] Jie Guo, Jun Yang, Youtao Zhang, and Yiran Chen. Low cost power failure protection for MLC NAND flash storage systems with PRAM/DRAM hybrid buffer. In *Design, Automation Test in Europe Conference Exhibition (DATE '13)*, pages 859–864, 2013.

[40] J. Hu, C. J. Xue, Q. Zhuge, W. C. Tseng, and E. H. M. Sha. Towards energy efficient hybrid on-chip scratch pad memory with non-volatile memory. In *2011 Design, Automation Test in Europe (DATE '11)*, pages 1–6, 2011.

[41] Jingtong Hu, Qingfeng Zhuge, Chun Jason Xue, Wei-Che Tseng, and Edwin H.-M. Sha. Software enabled wear-leveling for hybrid PCM main memory on embedded systems. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE '13)*, pages 599–602, 2013.

[42] Min Huang, Zhaoqing Liu, and Liyan Qiao. Asymmetric programming: A highly reliable metadata allocation strategy for MLC NAND flash memory-based sensor systems. *Sensors*, 14(10):18851–18877, 2014.

[43] Po-Chun Huang, Yuan-Hao Chang, Tei-Wei Kuo, Jen-Wei Hsieh, and Miller Lin. The behavior analysis of flash-memory storage systems. In *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC '08)*, pages 529–534, 2008.

[44] Po-Chun Huang, Yuan-Hao Chang, Kam-Yiu Lam, Jian-Tao Wang, and Chien-Chin Huang. Garbage collection for multiversion index in flash-based embedded databases. *ACM Transactions on Design Automation of Electronic Systems*, 19(3):25:1–25:27, 2014.

[45] Ilya Issenin, Erik Brockmeyer, Miguel Miranda, and Nikil Dutt. DRDU: A data reuse analysis technique for efficient scratch-pad memory management. *ACM Transactions on Design Automation of Electronic Systems*, 12(2), 2007.

[46] Jaeyong Jeong, Sangwook Shane Hahn, Sungjin Lee, and Jihong Kim. Lifetime improvement of NAND flash-based storage systems using dynamic program and erase scaling. In *12th USENIX Conference on File and Storage Techniques (FAST '14)*, pages 61–74, 2014.

[47] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O stack optimization for smartphones. In *USENIX Annual Technical Conference (ATC '13)*, pages 309–320, 2013.

[48] Lei Jiang, Youtao Zhang, and Jun Yang. ER: Elastic reset for low power and long endurance MLC based phase change memory. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED '12)*, pages 39–44, 2012.

[49] Adwait Jog, Asit K. Mishra, Cong Xu, Yuan Xie, Vijaykrishnan Narayanan, Ravishankar Iyer, and Chita R. Das. Cache revive: Architecting volatile STT-RAM caches for enhanced performance in CMPs. In *Proceedings of the 49th Annual Design Automation Conference (DAC '12)*, pages 243–252, 2012.

[50] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting storage for smartphones. *ACM Transactions on Storage*, 8(4):14:1–14:25, 2012.

[51] Hyojun Kim and Seongjun Ahn. BPLRU: A buffer management scheme for improving random writes in flash storage. In *6th USENIX Conference on File and Storage Technologies (FAST '08)*, volume 8, pages 1–14, 2008.

[52] Yuan-Hung Kuan, Yuan-Hao Chang, Po-Chun Huang, and Kam-Yiu Lam. Space-efficient multiversion index scheme for PCM-based embedded database systems. In *51st ACM/EDAC/IEEE Design Automation Conference (DAC '14)*, pages 147:1–147:6, 2014.

[53] Yuan-Hung Kuan, Yuan-Hao Chang, Po-Chun Huang, and Kam-Yiu Lam. Space-efficient multiversion index scheme for PCM-based embedded database systems. In *Proceedings of the 51st Annual Design Automation Conference (DAC '14)*, pages 147:1–147:6, 2014.

[54] Tei-Wei Kuo, Yuan-Hao Chang, Po-Chun Huang, and Che-Wei Chang. Special issues in flash. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '08)*, pages 821–826, 2008.

[55] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12):1352–1361, 2001.

[56] Sungjin Lee, Taejin Kim, Ji-Sung Park, and Jihong Kim. An integrated approach for managing the lifetime of flash-based SSDs. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '13)*, pages 1522–1525, 2013.

[57] Boxun Li, Yu Wang, Yiran Chen, H.H. Li, and Huazhong Yang. ICE: Inline calibration for memristor crossbar-based computing engine. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE '14)*, pages 1–4, 2014.

[58] Boxun Li, Yuzhi Wang, Yu Wang, Yiran Chen, and Huazhong Yang. Training itself: Mixed-signal training acceleration for memristor-based neural network. In *IEEE/ACM 19th Asia and South Pacific Design Automation Conference (ASP-DAC '14)*, pages 361–366, 2014.

[59] Xueliang Li, Guihai Yan, Yinhe Han, and Xiaowei Li. Smartcap: Using machine learning for power adaptation of smartphone's application processor. *ACM Transactions on Design Automation of Electronic Systems*, 20(1):8:1–8:16, 2014.

[60] Qian Lin, Zhengwei Qi, Jiewei Wu, Yaozu Dong, and Haibing Guan. Optimizing virtual machines using hybrid virtualization. *Journal of Systems and Software*, 85(11):2593 – 2603, 2012.

[61] Chenchen Liu and Hai Li. A weighted sensing scheme for ReRAM-based cross-point memory array. In *2014 IEEE Computer Society Annual Symposium on VLSI (ISVLSI '14)*, pages 65–70, 2014.

[62] D. Liu, T. Wang, Y. Wang, Z. Shao, Q. Zhuge, and E. Sha. Curling-PCM: Application-specific wear leveling for phase change memory based embedded systems. In *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC '13)*, pages 279–284, 2013.

[63] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of linux file system evolution. volume 10, pages 3:1–3:32, 2014.

[64] Hang-Ting Lue, Pei-Ying Du, Chih-Ping Chen, Wei-Chen Chen, Chih-Chang Hsieh, Yi-Hsuan Hsiao, Yen-Hao Shih, and Chih-Yuan Lu. Radically extending the cycling endurance of flash memory (to>100M cycles) by using built-in thermal annealing to self-heal the stress-induced damage. In *2012 IEEE International Electron Devices Meeting (IEDM '12)*, pages 9.1.1–9.1.4, 2012.

[65] Chris Mellor. TLC flash gets tender loving care from DensBits. *http://www. theregister.co.uk/2012/05/02/densbit_tlc/*, 2012.

[66] Micron. 3D NAND flash memory flyer. *https://www.micron.com/resource-details/1cea2229-16db-483b-8e39-48fc7ec29022*, 2016.

[67] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: random write considered harmful in solid state drives. In *10th USENIX Conference on File and Storage Technologies (FAST '12)*, page 12, 2012.

[68] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*, pages 471–484, 2014.

[69] H. Park, S. Yoo, and S. Lee. Power management of hybrid DRAM/PRAM-based main memory. In *48th ACM/EDAC/IEEE Design Automation Conference (DAC '11)*, pages 59–64, 2011.

[70] Youngwoo Park, Sung Kyu Park, and Kyu Ho Park. Linux kernel support to exploit phase change memory. In *Linux Symposium*, volume 2010, pages 217–224, 2010.

[71] Wu Qi, Guiqiang Dong, and Tong Zhang. Exploiting heat-accelerated flash memory wear-out recovery to enable self-healing SSDs. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '11)*, pages 1–5, 2011.

[72] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '42)*, pages 14–23, 2009.

[73] Samsung. S3C6410. *http://www.samsung.com/global/business/semiconductor/product/application/detail?productId=7115&iaId=835*.

[74] Samsung. Exynos4412. http://www.samsung.com/global/business/semiconductor/file/product/Exynos_4_Quad_User_Manaul_Public_REV100-0.pdf, 2012.

[75] Nak Hee Seong, Dong Hyuk Woo, and Hsien-Hsin S. Lee. Security refresh: Prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*, pages 383–394, 2010.

[76] Zili Shao, Yongpan Liu, Yiran Chen, and Tao Li. Utilizing PCM for energy optimization in embedded systems. In *Proceedings of the 2012 IEEE Computer Society Annual Symposium on VLSI (ISVLSI '12)*, pages 398 –403, 2012.

[77] L. Shi, K. Wu, M. Zhao, C.J. Xue, D. Liu, and E.H. Sha. Retention trimming for lifetime improvement of flash memory storage systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(1):58–71, 2016.

[78] Liang Shi, Keni Qiu, Mengying Zhao, and C.J. Xue. Error model guided joint performance and endurance optimization for flash memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD '14)*, 33(3):343–355, 2014.

[79] Liang Shi, Chun Jason Xue, Jingtong Hu, Wei-Che Tseng, Xuehai Zhou, and Edwin H.-M. Sha. Write activity reduction on flash main memory via smart victim cache. In *Proceedings of the 20th symposium on Great lakes symposium on VLSI (GLSVLSI '10)*, pages 91–94, 2010.

[80] Guangyu Sun, Yongsoo Joo, Yibo Chen, Dimin Niu, Yuan Xie, Yiran Chen, and Hai Li. A hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement. In *2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA '10)*, pages 1–12, 2010.

[81] Guangyu Sun, Xiaoxia Wu, and Yuan Xie. Exploration of 3D stacked L2 cache design for high performance and efficient thermal control. In *Proceedings of the 14th ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED '09)*, 2009.

[82] Yuan Heng Tseng, Chia-En Huang, C.H. Kuo, Y.-D. Chih, and Chrong Jung Lin. High density and ultra small cell size of contact ReRAM (CR-RAM) in 90nm CMOS logic technology and circuits. In *2009 IEEE International on Electron Devices Meeting (IEDM '09)*, pages 1–4, 2009.

[83] Chia-Heng Tu, Hui-Hsin Hsu, Jen-Hao Chen, Chun-Han Chen, and Shih-Hao Hung. Performance and power profiling for emulated android systems. *ACM Transactions on Design Automation of Electronic Systems*, 19(2):10:1–10:25, 2014.

[84] VMware. MVP. *http://www.vmware.com/ap/company/acquisitions/trango.html*, 2008.

[85] Chundong Wang and Weng-Fai Wong. Observational wear leveling: An efficient algorithm for flash memory management. In *Proceedings of the 49th ACM/EDAC/IEEE Design Automation Conference (DAC '12)*, pages 235–242, June 2012.

[86] Chundong Wang and Weng-Fai Wong. SAW: System-assisted wear leveling on the write endurance of NAND flash devices. In *Proceedings of the 50th Design Automation Conference (DAC '13)*, pages 164:1–164:9, 2013.

[87] Rujia Wang, Lei Jiang, Youtao Zhang, and Jun Yang. SD-PCM: Constructing reliable super dense phase change memory under write disturbance. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, pages 19–31, 2015.

[88] Y. Wang, Y. Liu, Y. Liu, D. Zhang, S. Li, B. Sai, M.F. Chiang, and H. Yang. A compression-based area-efficient recovery architecture for nonvolatile processors. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE '12)*, pages 1519–1524, 2012.

[89] Yiqun Wang, Yongpan Liu, Shuangchen Li, Xiao Sheng, Daming Zhang, Mei-Fang Chiang, Baiko Sai, X.S. Hu, and Huazhong Yang. PaCC: A parallel compare and compress codec for area reduction in nonvolatile processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(7):1491–1505, 2014.

[90] Yiqun Wang, Yongpan Liu, Shuangchen Li, Daming Zhang, Bo Zhao, Mei-Fang Chiang, Yanxin Yan, Baiko Sai, and Huazhong Yang. A 3us wake-up time nonvolatile processor based on ferroelectric flip-flops. In *Proceedings of the ESSCIRC (ESSCIRC '12)*, pages 149–152, 2012.

[91] Yiqun Wang, Yongpan Liu, Yumeng Liu, Daming Zhang, Shuangchen Li, Baiko Sai, Mei-Fang Chiang, and Huazhong Yang. A compression-based area-efficient recovery architecture for nonvolatile processors. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '12)*, pages 1519–1524, 2012.

[92] Yu Wang, Boxun Li, Rong Luo, Yiran Chen, Ningyi Xu, and Huazhong Yang. Energy efficient neural networks for big data analytics. In *Design, Automation and Test in Europe Conference and Exhibition (DATE '14)*, pages 1–2, 2014.

[93] D. Woodhouse. Jffs: The journaling flash file system. In *Ottowa Linux symposium*, 2012.

[94] Chin-Hsien Wu and Tei-Wei Kuo. An adaptive two-level management for the flash translation layer in embedded systems. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '06)*, pages 601 –606, 2006.

[95] G. Wu, H. Zhang, Y. Dong, and J. Hu. CAR: Securing PCM main memory system with cache address remapping. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS '12)*, pages 628–635, 2012.

[96] Po-Liang Wu, Yuan-Hao Chang, and Tei-Wei Kuo. A file-system-aware ftl design for flash-memory storage systems. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '09)*, pages 393–398, 2009.

[97] Chun Jason Xue, Youtao Zhang, Yiran Chen, Guangyu Sun, J. Jianhua Yang, and Hai Li. Emerging non-volatile memories: opportunities and challenges. In *Proceedings of the 7th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '11)*, pages 325–334, 2011.

[98] Chao Zhang, Guangyu Sun, Weiqi Zhang, Fan Mi, Hai Li, and Weisheng Zhao. Quantitative modeling of racetrack memory, a tradeoff among area, performance, and power. In *2015 20th Asia and South Pacific Design Automation Conference (ASP-DAC '15)*, pages 100–105, 2015.

[99] Chao Zhang, Guangyu Sun, Xian Zhang, Weiqi Zhang, Weisheng Zhao, Tao Wang, Yun Liang, Yongpan Liu, Yu Wang, and Jiwu Shu. Hi-fi playback: Tolerating position errors in shift operations of racetrack memory. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*, pages 694–706, 2015.

[100] W. Zhang, N. K. Jha, and L. Shang. Low-power 3D NANO/CMOS hybrid dynamically reconfigurable architecture. *ACM Journal on Emerging Technologies in Computing Systems*, 6(3):10.1–10.32, 2010.

[101] Xianwei Zhang, Youtao Zhang, Bruce R. Childers, and Jun Yang. Exploiting dram restore time variations in deep sub-micron scaling. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE '15)*, pages 477–482, 2015.

[102] Yue Zhang, Chao Zhang, J.-O. Klein, D. Ravelosona, Guangyu Sun, and Weisheng Zhao. Perspectives of racetrack memory based on current-induced domain wall motion: From device to system. In *2015 IEEE International Symposium on Circuits and Systems (ISCAS '15)*, pages 381–384, 2015.

[103] Jishen Zhao, O. Mutlu, and Yuan Xie. FIRM: Fair and high-performance memory control for persistent memory systems. In *47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '14),*, pages 153–165, 2014.

[104] Mengying Zhao, Lei Jiang, Liang Shi, Youtao Zhang, and C.J. Xue. Wear relief for high-density phase change memory through cell morphing considering process variation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(2):227–237, 2015.

[105] Mengying Zhao, Lei Jiang, Youtao Zhang, and Chun Jason Xue. SLC-enabled wear leveling for MLC PCM considering process variation. In *Proceedings of the 51st Annual Design Automation Conference (DAC '14)*, pages 36:1–36:6, 2014.

[106] Kan Zhong, Xiao Zhu, Tianzheng Wang, Dan Zhang, Xianlu Luo, Duo Liu, Weichen Liu, and Edwin Sha. DR. Swap: Energy-efficient paging for smartphones. In *Proceedings of the 2014 International Symposium on Low Power Electronics and Design (ISLPED '14)*, pages 81–86, 2014.