OPTIMIZING BIG DATA SYSTEMS WITH NON-VOLATILE MEMORIES: FROM GRAPH COMPUTING TO FLASH-BASED SSD ARRAYS

HAN LEI

PhD

The Hong Kong Polytechnic University

2019

THE HONG KONG POLYTECHNIC UNIVERSITY

DEPARTMENT OF COMPUTING

Optimizing Big Data Systems with Non-volatile Memories: From Graph Computing to Flash-based SSD Arrays

Han Lei

A Thesis Submitted in Partial Fulfillment of
the Requirements for the Degree of
Doctor of Philosophy

February 2019

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)

Han Lei ________________________(Name of Student)

ABSTRACT

Big data has been exerting an increasingly pervasive and profound influence on everyday life. For example, social networks, such as Facebook and Twitter, produce huge volumes of data and analyze big data to learn relationships between users, which are usually linked as large-scale graphs. However, as a huge collection of data over a time frame for processing and managing, big data remains extraordinarily complex and large for current computing infrastructures, leading to high processing costs and high storage resource consumption. Graph processing is an important part of big data analysis. Processing large-scale graphs on traditional platforms including CPU, GPU and FPGA is inefficient due to the many random memory access. Moreover, high-variety information with various data characteristics has significantly boosted. Persistently storing them on SSD-based arrays for high-velocity incurs high disk replacement rates due to the limited lifetime of SSDs. In addition, employing erasure codes for data protection in storage systems consumes high computational resources, further exacerbating the inefficiency of big data processing. In this thesis, we optimize big data systems with non-volatile memories from several aspects, including improving the performance of large-scale graph processing, extending the lifetime of SSD arrays and flash chips, and improving the efficiency of erasure coding on SSD arrays.

In the first part, we focus on optimizing the computational performance of big data with an emerging metal-oxide resistive random access memory (ReRAM). In the case of large-scale graph traversal, processing breadth-first search (BFS) on traditional platforms issues many random and irregular memory accesses, especially on CPU-based and GPU-based platforms. This leads to a huge amount of data movement between memories and processors, so that processors are always waiting for memories and executing instructions slowly. Moreover, the off-chip main memory in traditional platforms is a major consumer

of energy. To weaken these limitations, we propose a novel ReRAM-based processing-in-memory architecture for BFS, called RPBFS. In RPBFS, the ReRAM-based memory banks are separated into graph banks and master banks. We design an efficient graph mapping scheme to distributively store a graph on multiple graph banks. To reduce data movement overhead, we design an efficient traversal scheme that can constrain a graph search inside the related graph banks through collaboration with a master bank. Moreover, we propose an analytical performance model for RPBFS, which can help us identify bottlenecks and provide optimization opportunities for our design. The experimental results show that the proposed schemes can significantly improve graph traversal performance and achieve high energy reductions compared with both CPU-based and GPU-based BFS implementations.

In the second part, we optimize the storage efficiency for big data systems with NAND-based flash memory and ReRAM, achieving lower operational cost. Flash-based SSD arrays are increasingly being deployed in data centers. Compared with hard disk drive arrays, SSD arrays drastically enhance storage density and I/O performance, and reduce power and rack space. However, SSDs suffer aging issues since a flash block can only be experienced by a limited number of program/erase (P/E) cycles. The ability of storage systems to maintain service in the time aspect is particularly relevant to operational cost, frequently replacing failed drives makes service unstable. To optimize this, first, we propose FreeRAID which applies approximate storage via the interplay of RAID and SSD controllers to improve the lifetime of SSD-based RAID arrays. Our basic idea is to reuse faulty blocks (which contain pages with uncorrectable errors) to store approximate data (which can tolerate more errors). FreeRAID integrates two key techniques: dual-space management, which can efficiently allocate independent space for normal and approximate data, and adaptive-FTL, which can dynamically switch FTL schemes for an SSD according to its lifespan stage. We conduct experiments and compare our FreeRAID with conventional RAID and FTL schemes. The experimental results show that we can significantly increase the lifetime of SSD-based RAID arrays. Second, we extend the lifetime optimization to embedded stor-

age systems. We propose Rebirth-FTL, a pure software management in the flash translation layer for the lifetime optimization. Rebirth-FTL efficiently and effectively manages two spaces, approximate space and normal space, with approximation-aware address mapping, coordinated garbage collection and differential wear leveling. We also develop a scheme to pass approximate information from userland to kernel space in Linux, which can collaborate with Rebirth-FTL to optimize the lifetime of flash memory. A lifetime model is also presented for lifetime analysis. We implement Rebirth-FTL on an embedded development board and a simulator. Evaluations across a wide variety of workloads show that Rebirth-FTL significantly outperforms conventional FTLs in lifetime extensions and satisfies the workloads quality. Third, erasure codes such as Cauchy Reed-Solomon codes have been gaining ever-increasing importance for fault-tolerance in SSD-based RAID arrays. However, erasure coding on processor-based implementations such as a dedicated RAID controller relies on Galois Field arithmetic to perform matrix-vector multiplication, increasing computational complexity and leading to a huge number of memory accesses. We propose Re-RAID which uses ReRAM as the main memory in both RAID and SSD controllers. In Re-RAID, erasure coding can be processed in ReRAM memory to achieve high throughput. To minimize the overhead for recovering a single failure, we propose a confluent Cauchy-Vandermonde matrix as the generator matrix, which allows ReRAM memory on SSDs to perform the reconstruction task for a single failure. Experimental results show that our Re-RAID has a significant performance improvement in encoding and decoding compared with conventional processor-based implementation.

Keywords: Big data storage system, ReRAM, NAND flash memory, SSD, graph traversal, RAID, FTL, erasure codes, lifetime

PUBLICATIONS

1. **Lei Han**, Zhaoyan Shen, Duo Liu, Zili Shao, H. Howie Huang, Tao Li, “A Novel ReRAM-based Processing-in-Memory Architecture for Graph Traversal”, in *ACM Transactions on Storage (TOS)*, 2018.

2. **Lei Han**, Bin Xiao, Xuwei Dong, Zhaoyan Shen, and Zili Shao, “DS-Cache: A Refined Directory Entry Lookup Cache with Prefix-Awareness for Mobile Devices”, accepted in *2019 Design, Automation & Test in Europe Conference & Exhibition* (DATE ’19), Florence, Italy, March 25-29, 2019.

3. Fang Wang, Zhaoyan Shen, **Lei Han**, and Zili Shao, “ReRAM-based Processing-in-Memory Architecture for Blockchain Platforms,” in *Proceedings of the 24th Asia and South Pacific Design Automation Conference* (ASP-DAC ’19), Tokyo, Japan, Jan.21-24, 2019.

4. **Lei Han**, Zhaoyan Shen, Zili Shao, and Tao Li, “Optimizing RAID/SSD Controllers with Lifetime Extension for Flash-based SSD Array”, in *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (LCTES ’18), Philadelphia, Pennsylvania, United States, June 18-22, 2018.

5. **Lei Han**, Zhaoyan Shen, Zili Shao, H. Howie Huang, and Tao Li, “A novel ReRAM-based processing-in-memory architecture for graph computing,” in *2017 IEEE 6th Non-Volatile Memory Systems and Applications Symposium* (NVMSA ’17), Hsinchu, Taiwan, Aug.16-18, 2017.

ACKNOWLEDGEMENTS

First and foremost, I feel so grateful to have Prof. Zili Shao and Prof. Bin Xiao as my supervisors in the Hong Kong Polytechnic University (Polyu). I have learned a lot from their insightful guidance, meticulous attitude and professional supervision. It is my great pleasure to be a student of Prof. Shao, he has been giving me many elaborate suggestions and guidance even though he is now with the Chinese University of Hong Kong. Especially such a sentence "be a valuable member to society" has been affected me. I want to thank him for supporting me over the years. I also want to thank Prof. Bin Xiao for his encouragement and advice. Actually, without those fruitful discussions, I can hardly achieve the research outcomes which constitute the main content of this thesis.

I must acknowledge Prof. Howie Huang at the George Washington University and Prof. Tao Li at the University of Florida. I learned a lot from the discussions and interactions with them in academic research. Their truly scientist intuitions and genial personalities enrich my intellectual maturity. Besides, I want to thank Prof. Yi Lin at the Northwestern Polytechnical University, for his guidance and encouragement. I also express my gratitude to Prof. Shuai Li from the Polyu for his comments and suggestions on my research.

I would like to thank my research group in Polyu, including Dr. Yi Wang, Dr. Duo Liu, Dr. Renhai Chen, Dr. Zhaoyan Shen, Chenlin Ma, Yuanjing Shi, Fang Wang, Dr. Shang Gao, Dr. Zhe Peng, Zecheng Li, Songlin Hou and Lihao Liu. Thanks for their considerate assistance on my research and daily life during my Ph.D. study. Another particular thanks goes to my friends in Polyu, including Lei Xue, Zhijian He, Xingye Lu, Qiang Li, Yu Lei, Quanyu Dai, Hui Li, Liang Zhang, Xinbo Yu, Runjie Tan, Wengen Li, Ruosong Yang, Ningning Hou, Qiang Zhang, Edison Chan, Sitong Mao, Yumeng Guo, Jin Xiao, Wangmeng Xiang, Chuang Hu, Yinyan Zhang, Zhonghuang Yang, Jiaxing Shen, Jianrui Cai, Wenjian

Xu, Bo Tang, Shuhang Gu, Bo Lu, Tian Lan, Yin Xiao, Jacob, Daniel, Shamsa, Prof. Cecilia W.P. Li-Tsang and many others. The discussions and exchanges with them made my Ph.D. study a nice journey.

I want to thank Prof. Man Lung Yiu from Hong Kong Polytechnic University for kindly being the Chairman of the Board of Examiners (BoE). I also thank Prof. Tei-wei Kuo from National Taiwan University, and Prof. Xiaowen Chu from Hong Kong Baptist University, for kindly taking time out from their busy schedule to serve as my external examiners.

I recognize that this thesis would not have been possible without the financial assistance from the Polyu. I appreciate Prof. Shao and Prof. Xiao, and the Department of Computing for offering me the travel grants to attend several international conferences. I also acknowledge the student halls of residence of Polyu for providing me a cosy living space for my stay.

Finally, my special thanks goes to my family, including my parents, my grandmother and also in memory of my grandfather. Thanks for their endless love, support, and encouragement through my entire life. They are always on my side, and they always let me pursue my dream for so long and so far away from home. Also, I am really grateful for my girlfriend's endless love, patience, and understanding.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

The uses of big data have become ubiquitous in many areas of our daily life. For example, there were 500 millions Tweets sent [6] and 300 millions photos uploaded to Facebook [32] for storage each day in 2018. Although big data is increasingly becoming more understandable to computers with sustainable technologies, it is still extraordinarily complex and large for current computing infrastructures, leading to high data processing costs and high storage resource consumption. In the case of large-scale graph processing, simply increasing the number of processor cores is ineffective in improving performance [1]. Moreover, high-volume and high-variety information has increased significantly, requiring terabyte-level (TB) and petabyte-level (PB) storage systems for ongoing storage. SSD-based Redundant Arrays of Independent Disks (RAID) can provide mass and high-speed storage, and they rely on redundancy to protect data against data loss. Compared with hard disk drives (HDDs), SSDs benefit from lower access latency and energy consumption, and smaller rack space. However, SSDs suffer aging problems [121]. Especially in the RAID systems, parity updates incur extra writes and garbage collections (GC), which further degrades the lifetime of SSDs arrays. Reported from [47], 2%-7% SSDs develop at least one bad chip in the deployment of a data center. Frequently replacing failed drives makes big data application services unstable. In addition, among various methods for generating redundancy, several storage systems have been employing erasure coding to protect data due to the lower storage overhead. However, erasure coding on traditional processor-based implementations is still computationally expensive, further exacerbating the inefficiency of big data processing. Thus, it becomes extremely important to optimize big data storage systems in computing performance and operational costs.

In this thesis, we focus on optimizing big data storage systems with non-volatile memories in computing performance and operational cost. Specially, we employ two kinds of emerging non-volatile memories, metal-oxide resistive random access memory (ReRAM) [4] and NAND flash memory, to improve the performance of large-scale graph processing and erasure coding, and to improve the lifetime of SSD-based arrays and flash chips. ReRAM stores information according to the creation and destruction of the conductive filaments in the metal oxide layer. ReRAM enjoys low access latency (programming latency is less than 30ns), low energy consumption (0.1-3 pJ per bit) and long endurance (up to 10^{12} P/E cycles), so it can be a good candidate for main memory. Compared with other non-volatile memories such as STT-RAM and phase-change memory (PCM) which either have fast access or have high retention and endurance features, only the ReRAM crossbar structure has the computational capability of performing matrix-vector multiplication and sum operations, which inherently fits the process-in-memory (PIM) concept. Moreover, NAND flash memory has been widely adopted as the storage medium in many storage devices due to its fast access speed, such as in SSDs and embedded boards. However, it is still challenging to well utilize these non-volatile memories. First, processing data with ReRAM in a PIM way needs to be carefully examined. In the case of processing large-scale graphs, the graph mapping scheme and the execution mechanism on ReRAM memory should be properly designed to reduce data movement overhead and richly utilize memory bandwidth. Second, existing approaches that reducing the writes traffic on flash memory and redistributing parity on SSDs gain few benefits to extend the lifetime of flash memory. With the huge-variety of big data, some special data features, such as error-tolerance, are well worth exploring.

In the first part of this thesis, we investigate utilizing ReRAM to improve both computational and I/O performance for large-scale graph processing. We study the algorithm of breadth-first search (BFS). We propose a novel ReRAM-based processing-in-memory architecture called RPBFS, in which graph data can be persistently stored and processed in place. In RPBFS, the ReRAM-based memory banks are separated into graph banks and master banks. We design an efficient graph mapping scheme, in which a graph bank stores the adjacency list of a graph partition. We also propose an efficient graph traversal algorithm that works through collaboration with a master bank to traverse a graph. In RPBFS, the data movement for a graph traversal is wrapped within graph banks, and the movement is only related to the synchronization of vertex bitmaps. Moreover, we propose an analytical performance model to analyze the graph traversal efficiency with RPBFS, which can help us identify bottlenecks and provide optimization opportunities for our design.

In the second part, we optimize the storage efficiency of big data system with NAND flash memory and ReRAM, achieving lower operational cost. The ability of storage systems to maintain service in terms of time, such as the lifetime of storage devices, is particularly relevant to the operational cost. Moreover, many storage systems have been employing erasure codes to protect data, with coding performance one of the key factors into the system efficiency.

(1). We are the first to apply approximate storage, via the interplay of RAID and SSD controllers, to optimize the lifetime of SSDs arrays. We explore the benefits of relaxing the integrity constraints of flash blocks. Faulty flash blocks are reused to store approximate data that can tolerate some errors. With the goal of extending lifetime of SSD-based RAID arrays, we propose a cross-layer lifetime optimization framework, called FreeRAID (Flash-resurrection RAID). FreeRAID tightly couples the components in both RAID and SSD controllers. FreeRAID combines two techniques. First, with the knowledge of physical blocks in SSDs, the RAID controller in FreeRAID efficiently allocates normal and faulty blocks to serve data with different error-tolerances, and makes different types of data error-isolated. Also, FreeRAID and the existing optimized RAID schemes can coalesce to further reduce write traffic on SSDs. These optimized schemes include parity logging, parity caching and elastic striping. Second, FreeRAID can dynamically switch FTL strategies on an SSD to maintain access performance and storage efficiency. To determine whether data validity has been accomplished on faulty blocks, two error rate assessment approaches are proposed by considering two dominant errors sources of flash memory.

(2). For flash memory in embedded storage systems, we propose Rebirth-FTL, a pure software management in flash translation layer (FTL) for lifetime optimization. With an increasing amount of approximate data such as images and videos, Rebirth-FTL reuses faulty blocks that contain uncorrectable errors to store these data. Rebirth-FTL efficiently and effectively manages two addressable spaces, approximate space and normal space, with efficient address mapping, coordinated garbage collection and differential wear leveling. To make the flash devices approximate information-aware, we demonstrate how to pass the approximate information from applications to flash devices through the whole Linux operating system (OS) in a top-down way. We analyze the benefits of a flash memory with Rebirth-FTL using a lifetime model. We implement and deploy Rebirth-FTL on an embedded development board and a simulator, and we demonstrate its effectiveness on them.

(3). The traditional processor-based RAID controller relies on Galois Field arithmetic to perform matrix-vector multiplication for erasure coding, which is computationally expensive on processors. We propose a novel ReRAM-optimized RAID system for accelerating erasure coding, called Re-RAID. Re-RAID uses ReRAM as the main memory in both RAID and SSD controllers, and it performs erasure coding on ReRAM crossbars. To minimize the overhead for recovering a single failure, we propose a confluent Cauchy-Vandermonde matrix as the generator matrix. Then, the SSDs can leverage their ReRAM memories to recover a single failure, which can greatly alleviate the computing workloads of a processor-based RAID controller. For multiple failures, processors and ReRAM memories in the RAID controller work in close collaboration. The processors construct a decoding matrix, and map the matrix to ReRAM memories, and then the ReRAM memories perform matrix-vector multiplication to recover lost data.

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

1.1 Related Work

In this section, we describe state-of-the-art work related to optimization approaches for big data systems in computing and storage.

In the previous work, there has been work done in three main domains: (I) Processing-in-memory accelerator, (II) Approximate storage, and (III) Software-managed flash and RAID arrays. We briefly describe these techniques, and compare them with representative techniques in respective chapters.

1.1.1 Processing-in-memory Accelerator

Processing-in-memory technique is an effective way to alleviate the bandwidth bottlenecks by integrating computational logic within or near memory. Recent studies have proposed in-memory accelerators for specialized applications. Mirzadeh et al. [80] push logic toward memory by leveraging 3D-stacked DRAM designs. The logic layer integrated with several DRAM dies can execute data-intensive operations. Jeddeloh et al. [50] propose a three-dimensional DRAM architecture in which the DRAM is moved to the logic layer with high-performance transistors. Timing, refresh and thermal management for DRAM can be optimized locally. Akin et al. [3] propose a near-memory accelerator integrated within 3D stacked DRAM, and Zhang et al. [143] move memory-intensive computations closer to memory in GPU. For large-scale graph processing, Ahn et al. [1] propose a programmable PIM accelerator by integrating many logic cores into 3D-stacked memory to increase memory bandwidth. The proposed architecture can process four graph algorithms, such as Single-Source Shortest Path (SSSP).

Due to the inherent computational capability of ReRAM, some recent studies also explore ReRAM to accelerate several kinds of applications in a PIM way. PRIME [19] is a PIM architecture to accelerate neural network applications. In PRIME, ReRAM serves as main memory and serves as computation units. The ReRAM crossbar is utilized to perform matrix-vector multiplication for neural network applications. ISSCC [113] and Pipelayer [119] are pipelined architectures with memristor crossbars for processing neural networks computations, and different parallelism granularities on them are explored to accelerate computation. ReRAM also has been studied for processing graph algorithms. Pinatubo [65] is a processing-in non-volatile memory architecture for bulk bitwise operations. The read circuitry redesigned for multi-row bitwise operations, is efficient for bitmap-based BFS processing. GraphR [120] follows the principle of near-data processing, and utilizes ReRAM crossbars to serve both storage and computational functions. The compressed graph data are persistently stored in ReRAM memory, and then the graph data can be converted to sparse matrix representation and mapped to other ReRAM memory for processing. However, the conversion incurs high execution and transfer costs. This thesis proposes a novel ReRAM-based PIM for graph processing without any conversion, and the data movement is wrapped within memory by an efficient mapping scheme and an efficient traversal scheme.

1.1.2 Approximate Storage

Approximate storage exploits the error-tolerance of applications to reduce I/O latency and energy consumption of storage substrates. Bit-by-bit precision is costly for some applications; a small number of bit flips is acceptable to them, such as with videos and images. Recent work have demonstrated that approximate storage has led to multi-aspect improvements in solid state memories. Sampson et al. [109] propose reducing the number of write steps on PCM, to achieve higher write performance and energy savings. Similarly, Cui et al. [26] propose reducing the maximal threshold voltage for writes on 3D NAND flash memory, allowing write performance to be improved and the program disturbance in physical blocks to be alleviated. Ranjan et al. [100] leverage the error-tolerance of data to improve the energy-efficiency of spintronic memories. The quality on memory is configurable to meet the accuracy requirements of applications. Sampson et al. [108] allow a programmer to declare precise data or approximate data, and then store approximate data to approximate storage including cheaper memory, cache, and registers, achieving significant energy saving at very little accuracy loss.

Recent researches have proposed applying approximate storage for specific applications. Jevdjic et al. [51] compute bit-level reliability requirements for encoded video by tracking coding dependencies, and they implement different levels of error correction for streams reliability needs. Guo et al. [38] propose a selective error correction technique to implement high-density image storage. Via a case study of JPEG images, they can significantly increase the storage density of PCM with negligible quality loss. In [39], Guo et al. use good error protection technique for the important parts of images and videos, while using minimal effort to protect less important parts. The proposed unequal error protection technique can attain higher quality with lower computational complexity. Palomino et al. [90] employ varying degrees of approximations at both the algorithmic and data levels to reduce on-chip temperature when processing a video. The regions of a video can be classified by an adaptive content-driven approximate technique, and then the regions with different approximate modes are processed with suitable approximate computing.

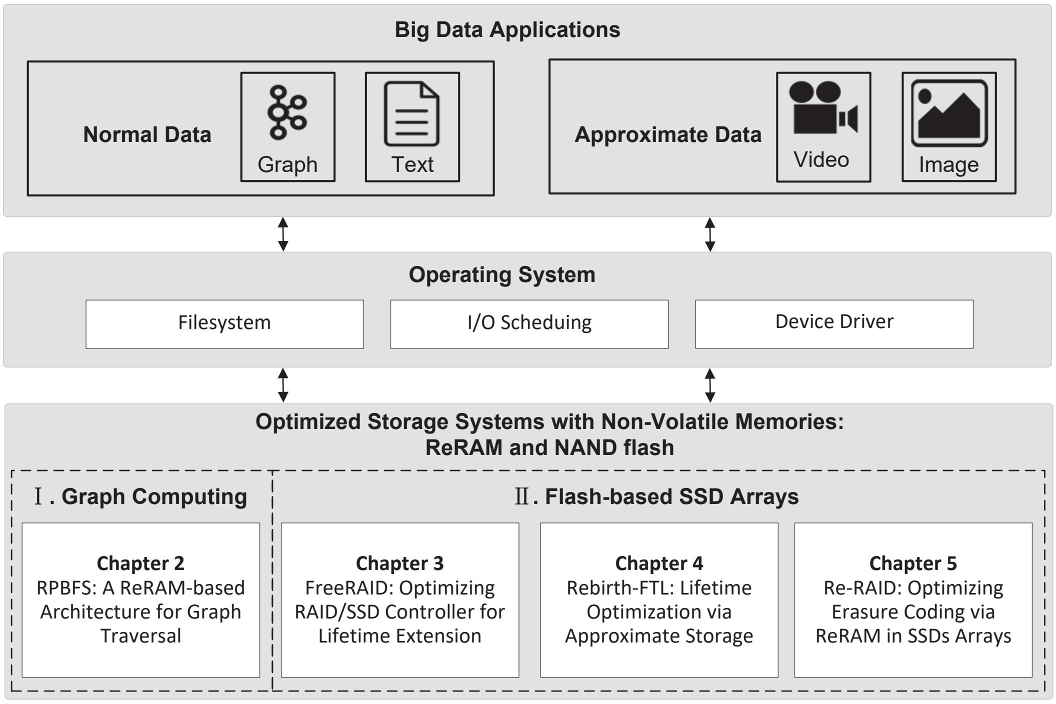
Our work shares a common principle with prior work of relaxing integrity constraints of storage substrates, but we aim to extend the lifetime of SSD-based RAID arrays and flash memories. This thesis focuses on software management at the RAID and FTL levels without the hardware changes (such as modifying the threshold voltage [109] [26]) or dedicated error correction codes (such as encoding videos with H. 264 [51]).

1.1.3 Software-managed RAID Arrays and Flash

An SSD-based RAID array provides a virtual logical disk by combining the space of the SSDs. Striping and parity are two commonly-used RAID techniques in the RAID controller. With striping, logically sequential data are divided into data chunks. Parity is a redundancy-based protection scheme, by which parity data is generated based on a group of data chunks. Write updates incur additional writes for parity updates. To efficiently reduce write traffic, recent research has proposed various optimized schemes on RAID. Parity logging [122] utilizes a journaling scheme to reduce small writes cost by augmenting a log device. Parity caching scheme [21] delays parity updates by caching all incoming requests in a buffer, so the number of reads and writes for generating parity can be reduced. Elastic parity logging [66] encodes new incoming data chunks to form new stripes, and the parity for a partial stripe is appended to a log device. For SSD-based RAID arrays, Li et al. [67] propose an analytical model to quantify the reliability of arrays, which can help decide the appropriate parity distribution. Pan et al. [91] propose a grouping-based elastic striping scheme to reduce both write traffic and response time on SSDs. Koo et al. [59] propose a dual RAID scheme which is a combination of RAID-5 and RAID-6 for maintaining high reliability and access performance. Besides that, the lifetime of SSD-based RAID arrays remains a major concern. Moon et al. [81] analyze the relationship between parity scheme and lifetime, and they find that write amplification is a major factor in the lifetime of RAID arrays. Yongseok et al. [87] use a log-structured cache to eliminate read-modify-write operations, and propose the use of destaging to enhance the lifetime of SSD arrays. Different from prior work which mainly focuses on reducing writes, our work explores the special features of big data, such as error-tolerance, to significantly extend the lifetime of SSD-based RAID arrays.

RAID schemes are categorized into several levels based on parity, and the methods for generating parity in a RAID system are varied. For example, with one to two parities generated for each stripe, RAID-5 and RAID-6 can tolerate one and two failed drive at any time, respectively. Many parity implementations exist. Fu et al. [36] propose a new Maximum Distance Separable (MDS) code with new parity chains and new parity distributions for RAID-6 to optimize degraded reads and partial stripe writes. With the same goal, D-Code [35] uses new kinds of horizontal parities to optimize I/O performance. Several codes are proposed to optimize parity computational complexity, including Tier-code [63], EVENODD [14], RD-P [24] and X-code [142]. Trifonov et al. [128] propose low-complex Reed-Solomon (RS) codes to improve encoding and decoding performance. Guruswami et al. [41] propose repair schemes for high-rate RS codes to optimize repair bandwidth in a cloud RAID system. Zhang et al. [144] propose an efficient Cauchy Reed-Solomon coding approach, called Caco.

The flash translation layer in the SSD controller manages flash memory. Several prior research studies specifically optimize FTL schemes in address mapping, garbage collection and wear leveling. Qin et al. [98] propose MNFTL to reduce the number of valid page copies for achieving low system response time. Liu et al. [70] propose RNFTL to improve the endurance and space utilization of blocks. DFTL [40] is an on-demand page-level FTL with one-level cache, with both the page-level mapping table and data blocks are stored in the flash memory. A block associative sector translation [58] scheme allocates a log block for only one



Big Data Applications
Normal Data
Graph
Text
Approximate Data
Video
Image
Operating System
Filesystem
I/O Scheduling
Device Driver
Optimized Storage Systems with Non-Volatile Memories:
ReRAM and NAND flash
Ⅰ. Graph Computing
Ⅱ. Flash-based SSD Arrays
Chapter 2
RPBFS: A ReRAM-based Architecture for Graph Traversal
Chapter 3
FreeRAID: Optimizing RAID/SSD Controller for Lifetime Extension
Chapter 4
Rebirth-FTL: Lifetime Optimization via Approximate Storage
Chapter 5
Re-RAID: Optimizing Erasure Coding via ReRAM in SSDs Arrays

Figure 1.1: Unified Research Framework.

data block for the efficiency of address translation. Jimenez et al. [52] relieve the weakest pages to implement block lifetime extension with a wear unleveling technique. Chang et al. [17] propose a typical static wear-leveling strategy called SWL to save the management overhead of FTLs. In contrast to these prior works, our work leverages the special feature of data to optimize the lifetime of flash memory without any hardware changes.

1.2 The Unified Research Framework

We present the unified research framework of this thesis in this section. Figure 1.1 illustrates the sketch of our framework.

In this thesis, we optimize big data systems with non-volatile memories from graph computing to flash-based SSD arrays, as shown in Figure 1.1. The data generated from big data applications are classified into two types: normal data and approximate data. Normal data are precise data, such as graph and text data. Approximate data can tolerate some errors, such as video and image which can tolerate some died pixels. Our work contains two parts.

In the first part for computing efficiency, we study the graph processing which is an important part of big data analysis. We investigate utilizing ReRAM to accelerate graph traversal. In the second part for storage efficiency, we propose integrating the RAID and SSD management to optimize the lifetime of flash memory by exploring the error-tolerance of approximate data. In addition, to improve the performance of erasure coding on SSD-based RAID arrays, we further propose leveraging ReRAM to achieve that.

In the first part, in Chapter 2, we propose a novel ReRAM-based processing-in-memory architecture for graph traversal, in which graph data can be persistently stored and processed in place. In the second part, in Chapter 3, we apply approximate storage via the interplay of RAID and SSD controllers to optimize the lifetime of SSDs arrays. The interplay tightly couples the components in both RAID and SSD controllers to efficiently manage the cross-layer space. In Chapter 4, we further explore approximate storage to optimize the lifetime of flash memory in FTL. We also demonstrate how to pass the approximate information from applications to flash devices in Linux OS. In Chapter 5, we use the ReRAM as an alternative main memory in both RAID and SSD controllers. The ReRAM-based main memory can perform erasure coding, which can greatly alleviate the computing workloads of processors in the controllers.

1.3 Contributions

The contributions of this thesis are summarized as follows.

- To minimize data movement overhead, we investigate utilizing ReRAM to improve the performance of large-scale graph processing. We propose a novel ReRAM-based processing-in-memory architecture called RPBFS in which graph data can be persistently stored and processed in place. An efficient graph mapping scheme is proposed to map a graph on multiple ReRAM memory banks. We also design an efficient graph traversal algorithm in RPBFS. In addition, we propose an analytical performance model to analyze the benefits of a graph traversal in RPBFS.

- We explore exploitable blocks in SSDs to serve approximate data in SSD-based RAID arrays. We propose FreeRAID, which leverages the interplay between RAID and SSD controllers to extend the lifetime of SSDs arrays. FreeRAID integrates two key techniques: dual-space management which can efficiently allocate space for normal and approximate data, and adaptive-FTL which can dynamically switch FTL schemes of an SSD to improve its storage efficiency.

- For flash memory in embedded storage systems, we propose Rebirth-FTL, a pure software management in flash translation layer for lifetime optimization. Rebirth-FTL efficiently and effectively manages two spaces, approximate space and normal space, with approximation-aware address mapping, coordinated garbage collection and differential wear leveling. We also develop a scheme to pass approximate information from userland to kernel space in Linux. A lifetime model is also presented for lifetime analysis.

- We propose Re-RAID which uses ReRAM as the main memory in both RAID and SSD controllers. The erasure coding in Re-RAID can be processed in ReRAM memory. To minimize the overhead for recovering a single failure, we propose a confluent Cauchy-Vandermonde matrix as the generator matrix, which allows ReRAM memory on SSDs to perform the reconstruction task for a single failure.

- We implement prototypes with the proposed techniques, and demonstrate the effectiveness of the proposed schemes by conducting a set of experiments.

1.4 Thesis Organization

The rest of this thesis is organized as follows.

- In Chapter 2, the first part of this thesis, we focus on optimizing the computational performance for big data. We investigate utilizing ReRAM to improve the performance of large-scale graph processing.

- In Chapter 3, the start of the second part, we optimize the storage efficiency of big data systems with NAND flash memory and ReRAM to achieve lower operational cost. We apply approximate storage via the interplay of RAID and SSD controllers to improve the lifetime of SSD-based RAID arrays.

- In Chapter 4, to extend the lifetime of flash memory in embedded storage systems, we propose Rebirth-FTL which reuses faulty blocks that contain uncorrectable errors to store approximate data for lifetime optimization in FTL.

- In Chapter 5, to alleviate the computing workloads of the RAID controllers, we propose Re-RAID which uses ReRAM as the main memory in both RAID and SSD controllers, and erasure coding can be processed in ReRAM memory.

- In Chapter 6, we present our conclusions and possible future directions for research arising from this work.

CHAPTER 2

A NOVEL RERAM-BASED PROCESSING-IN-MEMORY ARCHITECTURE FOR GRAPH TRAVERSAL

2.1 Introduction

The uses of graph-based computation for analyzing and understanding social networks, complex engineering systems, and metabolic networks are ubiquitous. With the tremendous increase in the amount of information, relationships between objects are often linked to form large-scale graphs, such as the friend relationship in social networks. In the above areas of applications, the common graph-theoretic algorithms on the large-scale graphs have been becoming increasingly important. A well-known example of algorithms is the breadth-first search that allows the peer-to-peer network BitTorrent to search all neighbors [22]. Current graph processing schemes mainly concentrate on CPU-based and GPU-based platforms. These traditional platforms separate graph processing into memory processing and processor processing, and it is still challenging to achieve good performance due to the many random and irregular memory accesses. This mechanism leads to a huge amount of data movement between memories and processors, so that the processors always wait for memories and execute instructions slowly [12] [1]. Moreover, the off-chip main memory system is a major consumer of energy due to the high capacitive load and power of buses and memory [115]. Although continuous efforts are being made to improve the multi-core parallelism and to optimize memory access performance, the data transfer between memory and processor in traditional platforms still represents one of the most significant bottlenecks for both performance and energy when performing graph-based algorithms. The possibility of reducing the data movement overhead in memory is therefore well worth exploring.

Processing-in-memory technique is an effective way to alleviate the bandwidth bottlenecks by integrating the computation logic within or near memory, so the memory-intensive computations can fully utilize the available memory bandwidth. Recent studies have proposed in-memory accelerators for specialized applications to reduce data movements [19] [125] [113] [119] [73] [143] [10] [149]. For example, Ahn et al. integrate PIM technology into 3D-stacked memory to increase memory bandwidth for graph processing [1]. Ozdal et al. propose an accelerator architecture to reduce the irregular access patterns and asymmetric convergence [88]. Although they are architectural accelerators for graph analysis, they cannot make graphs persistently stored in memory so as to involve slow secondary storage, and they ignore the effect of graph mapping and distribution on performance.

An emerging non-volatile memory, metal-oxide resistive random access memory, has the capability to perform arithmetic operations inside data storage [136], which inherently fits the concept of PIM. ReRAM enjoys lower access latency, lower energy consumption, and superior endurance than other non-volatile memories [139] [138] [134] [69]. Its most novel aspect is that the ReRAM crossbar structure is efficient at performing matrix-vector multiplication, which has been studied for neural network computation [19] [113] [119]. Furthermore, ReRAM crossbar has the potential to perform iterative graph processing algorithms. First, graph processing algorithms are not computation-intensive but memory-intensive [89] [72], and most of them (e.g., page-rank, graph traversal) can tolerate the imprecision arising from analog-to-digital conversions. Moreover, considering the fact that the size of cells keeps shrinking, multi-level cell (MLC) technology enables one ReRAM cell to store more information [5] [141] [98], which is suitable as a storage device to store large-scale graphs. Therefore, with the efficient capability in both storage and computation, ReRAM crossbar exhibits the potential to accelerate graph processing.

In reconsidering the connection between the computation and storage aspects of ReRAM crossbars, it is still challenging to utilize ReRAM crossbars to perform graph processing. First, a graph with several million vertices and edges needs to be mapped to multiple ReRAM crossbars, so the graph representation needs to be carefully organized. There are two major approaches to representing a graph: adjacent matrix and adjacency list. It is impractical to store the whole adjacent matrix in ReRAM, since the matrices of the real world graphs are highly sparse. For adjacency list, there are a number of formats, and each with different storage cost, computational characteristics and organizations. Second, the execution mechanism of ReRAM crossbars should be designed properly. Since most graph algorithms are iterative, it is important to maintain correctness in processing and the consistency of the intermediate data among all of the subgraphs located on the ReRAM crossbars. Some works have integrated ReRAM for large-scale graph processing [120] [45]. GraphR follows the principle of near-data processing, and it utilizes ReRAM crossbars for storage and computation [120]. The compressed graph data is persistently stored in ReRAM memory; however, it needs to be converted to sparse matrix representation for graph processing in ReRAM crossbars. The conversion incurs execution cost and transfer cost. Therefore, the data movement between ReRAM crossbars remains large, even the data do not need to pass through the memory hierarchy as in traditional platforms.

In this work we focus on accelerating graph traversal, and propose a novel ReRAM-based PIM architecture for BFS (RPBFS). RPBFS is a distinct ReRAM-based PIM architecture from recent PIM work. In this architecture, the ReRAM-based memory banks are separated into graph banks and master banks. The compressed adjacency lists are persistently mapped and scattered over multiple graph banks by an efficient mapping scheme. The master bank is selected for a graph to perform graph traversal through collaboration with graph banks. To reduce the data movement overhead, we design an efficient traversal scheme that can constrain the graph expansions inside the memory bank, and can parallelly perform the expansions on multiple memory banks. The movement of data in the RPBFS is only related to the synchronization of vertex bitmaps, which is much smaller than that of graph data. We also further explore the effect of graph distributions through an analytical performance model.

We conduct a series of experiments to evaluate the RPBFS across a wide variety of graphs compared with the state-of-the-art CPU-based and GPU-based parallel solutions [11] [71]. Our architecture yields speedups of up to 33.8× on the graph traversal performance, and achieves energy reductions of up to 142.8× over conventional systems. The results also verify the improvement in the performance of different graph partitioning schemes as the optimizing opportunity identified by an analytical model.

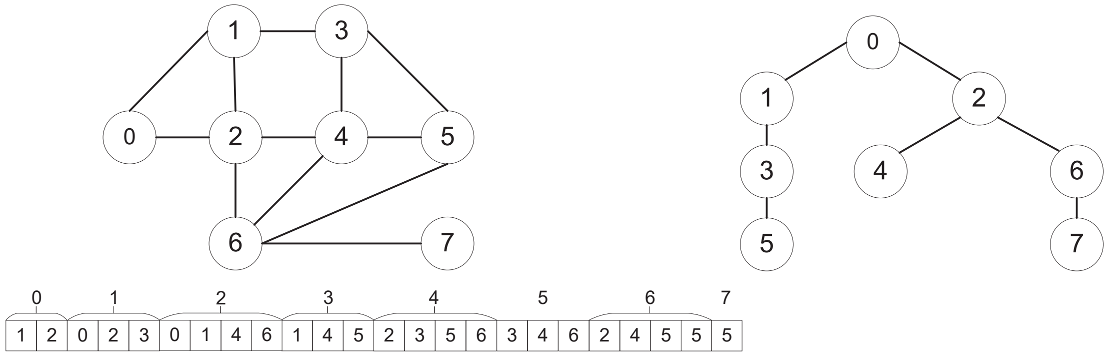
This work explores graph traversal based on ReRAM crossbars. We believe that our proposed architecture can also benefit other graph algorithms, such as single source shortest path. To summarize, this chapter makes the following contributions:

- We design a novel ReRAM-based PIM architecture with a set of peripheral circuits. An efficient graph mapping scheme is proposed to map a graph on multiple ReRAM memory banks.
- We design efficient graph traversal algorithms for graph banks and master banks, respectively. Data movement is minimized and bank-level parallelism is explored to effectively accelerate graph traversal.
- We propose an analytical performance model for our ReRAM-based PIM implementation. Finally, we evaluate our proposed scheme using a variety of real world graphs and compare it with other state-of-the-art solutions.

The rest of this chapter is organized as follows. The next section gives basic background to this study and gives a motivation example. Section 2.3 describes the architecture of RPBFS, as well as the graph mapping scheme. In Section 2.4, we briefly introduce how graph traversal proceeds on the RPBFS architecture. An analytical performance model is presented in Section 2.5, and we apply it to explore the optimization opportunity. Section 2.6 evaluates the performance of RPBFS in several key metrics. Finally, Section 2.7 discusses related work, and Section 2.8 concludes this chapter.

2.2 Background and Motivation

In this section, we first present the background on graph representation, BFS traversal and ReRAM basics. Then we use a motivation example to illustrate the traversal efficiency on a ReRAM crossbar.



(a) An example graph with its adjacency list

(b) One of its valid traversal trees from *Vertex* ‘0’

Figure 2.1: An example graph G and one of its valid traversal trees.

2.2.1 Graph Representation

Assuming a graph G with vertex set V and edge set E, there are a number of popular graph representations. Adjacency matrix and adjacency list are the most commonly used representations of graph, and the choice of the graph representation is situation specific.

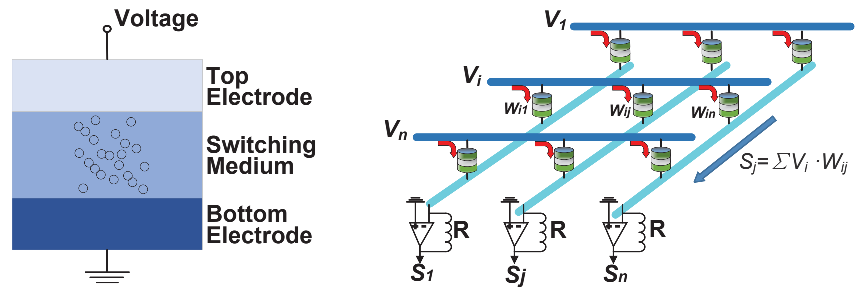
An adjacency matrix consists of rows that represent the source vertices, and columns that represent the destination vertices. The number of rows and columns equals the number of vertex ($|V|$). The values in a matrix are generally filled with zeros and ones. If two vertices are connected then the value in the source row and the destination column is represented by one. For the efficiency of storage, the compressed sparse row (CSR) format is one of the highest compression ratio representations. The CSR format puts the subsequent non-zeros of the matrix rows in contiguous memory locations, and uses an index array to point the list starts for describing the neighbors of each vertex. The size of the CSR representation of a graph equals the total size of edges ($|E|$) and the index array ($|V|$). Figure 2.1(a) shows the graph G of eight vertices and twelve edges with its adjacency list.

We argue that storing large-scale graphs in a matrix way is inefficient. First, it incurs enormous storage overhead, since the matrices of the real world graphs are highly s-parse. Second, some graph algorithms do not rely on matrix representation, such as BFS which can perform expansions by accessing adjacency lists. Even though some algorithms like PageRank can be performed based on a $|V|$-by-$|V|$ connectivity matrix, the high sparse matrix-vector multiplication operations can be converted by performing with compressed data formats, such as CSR and ELLPACK [74]. Thus we focus on the CSR format as graph representation in this work.

2.2.2 Breadth-first Search

Breadth-first search starts the traversal from a given source vertex and systematically explores a graph to discover every vertex. In accordance with the frontiers queue that is initialized with the source vertex, BFS explores their adjacent vertices and marks them as visited with the shortest depth. One of the valid traversal trees of graph G is shown in Figure 2.1(b). In addition, from Figure 2.1(a), it is easy to observe that some neighbors are contiguous vertices. For example, contiguous neighbors *Vertex 1* and *Vertex 2* are both neighboring vertices of *Vertex 0*. Suppose that *Vertex 0* is a frontier in the current level of a breadth-first search, after exploring its neighbors, *Vertex 1* and *Vertex 2* will be the frontiers for next level expansions. We investigate the contiguous neighbors of six real world graphs summarized in Table 2.5 in Section 2.6. We calculate the proportion of vertices that are of at least an average degree and have two contiguous neighbors. From Table 2.5, we can see that a number of vertices have contiguous neighbors on a suite of real world graphs. The contiguous neighbors may explore their neighbors at the same level in a graph traversal.

A top-down BFS, which aims at identifying the unvisited adjacent vertices of frontiers, is a traditional traversal algorithm. In a later stage, the direction from top-down can be switched to bottom-up. A bottom-up algorithm uses the unvisited vertices as frontiers to identify the visited vertices, and performs traversals more efficiently when the current frontiers are large [11] [71]. However, it still leads to random and irregular accesses of the graph data. Thus, reducing data movements is critical to improve the traversal performance.



(a) The structure of a ReRAM cell (b) A crossbar structure with 3*3 ReRAM cells.

Figure 2.2: ReRAM basics.

2.2.3 ReRAM Basics

ReRAM is an emerging non-volatile memory that stores data with resistance. A metal-oxide ReRAM cell consists of a top metal electrode, a metal-oxide resistive switch, and a bottom electrode [136], as shown in Figure 2.2 (a). By applying an external voltage across the ReRAM, the properties of conductive filament inside it change, leading to different resistances. A ReRAM array can be interconnected as a dense crossbar architecture without transistors, which is better suited for main memory due to the small size of the area of a ReRAM cell [82] [140]. Figure 2.2 (b) shows an area-efficient ReRAM crossbar array. If the input voltages V_1, V_2, ..., V_n are applied on the wordlines, and the values $W_{i,j}$ of cells are programmed, the current S_j at the end of jth bitline will represent the sum result of the dot product operations, $\sum V_i \cdot W_{i,j}$. In our design, if the voltage of the selected wordline is set to 1, while others are set to 0, the results at the output port in the bitlines will be the exact conductances of the cells, which can be used to identify neighbors.

2.2.4 Motivation Example

To illustrate the motivation of using ReRAM crossbars for graph traversal, we discuss it in two aspects. 1) The ReRAM crossbars can accelerate graph traversal. The neighbors related to a vertex are placed sequentially on ReRAM cells similar to the CSR format. Besides retaining the storage efficiency, by activating one wordline, all the cells on the wordline can be attained, which contains the neighbors of a requested vertex. 2) The properties of

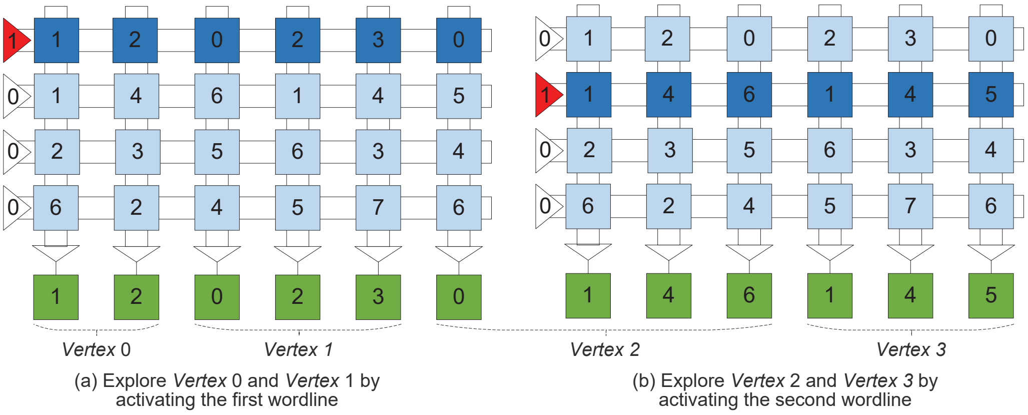

1
1
2
0
2
3
0
0
1
4
6
1
4
5
0
2
3
5
6
3
4
0
6
2
4
5
7
6
1
2
0
2
3
0
Vertex 0
Vertex 1
(a) Explore Vertex 0 and Vertex 1 by activating the first wordline
0
1
2
0
2
3
0
1
1
4
6
1
4
5
0
2
3
5
6
3
4
0
6
2
4
5
7
6
1
4
6
1
4
5
Vertex 2
Vertex 3
(b) Explore Vertex 2 and Vertex 3 by activating the second wordline

Figure 2.3: Activating one wordline can lead to the traversal of multiple vertices.

ReRAM, including its non-volatility, energy-saving feature, and fast access speed, make it a good candidate for PIM.

We use an example to illustrate the traversal efficiency on a ReRAM crossbar. The adjacency lists of the graph G are mapped to the ReRAM cells one after another on a 4*6 crossbar, as shown in Figure 2.3. Suppose that each cell can store one vertex, and that the source of the traversal is *Vertex 0*. In Figure 2.3 (a), the neighbors of the source can be attained by activating the first wordline and setting the input voltage of the rest to 0. Then, the output ports would not only contain all of the neighbors of *Vertex 0* (the first two bitlines), but also all of the neighbors of *Vertex 1* (the third to the fifth bitline), and one neighbor of *Vertex 2* (the last bitline). The next iteration is to explore the neighbors of *Vertex 1* and *Vertex 2*. Benefiting from the first activating wordline operation,it is only necessary to attain the remaining neighbors of *Vertex 2* , which can be done by activating the second wordline, as shown in Figure 2.3 (b). After this operation, the adjacency list of *Vertex 3* is also attained for the next expansion task. Following this access pattern, the neighbors of all of the vertices can be traversed without any extra access overhead. From Table 2.5 in Section 2.6, we observe that some of neighbors are contiguous. Therefore, in a BFS traversal, activating wordline operations on the ReRAM crossbars means that multiple neighbors of a vertex can be attained, and it has a probability to attain the neighbors of its contiguous vertices.

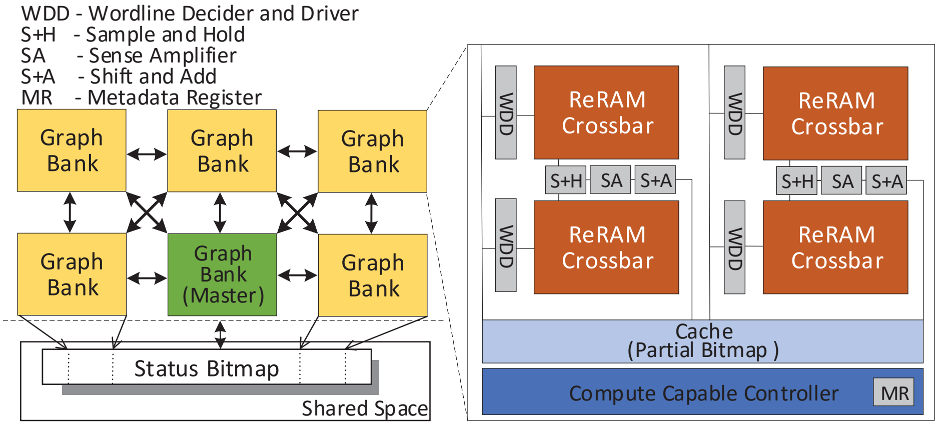


Figure 2.4: RPBFS architecture.

2.3 ReRAM-based PIM Architecture for Graph Traversal

We propose a novel ReRAM-based processing-in-memory architecture for breadth-first search, called RPBFS, which can efficiently accelerate graph traversal by minimizing data movement overhead. Figure 2.4 depicts an overview of RPBFS. RPBFS architecture partitions ReRAM-based memory banks into two types: master bank and graph bank. Due to the limited size of ReRAM crossbars, a graph with several millions vertices and edges could hardly be stored in a single memory bank, so the multiple memory banks are involved. A graph bank stores the adjacency list of a graph partition. A master bank is arbitrarily selected from the memory banks to schedule expansion tasks for each individual graph. The master bank stores the corresponding metadata of the graph banks, including the vertex range, and the s-tarting row number of ReRAM crossbars for the adjacency lists and location pointers of each graph partition. A memory bank can either be a master bank or a graph bank, and multiple partitions from different graphs can be mapped on a same memory bank.

This section gives details of the RPBFS architecture, and then discusses how the adjacency list can be mapped to a ReRAM crossbar. Finally we discuss how to map a graph on multiple memory banks.

2.3.1 Microarchitecture

A ReRAM-based main memory chip is composed of a number of memory banks, as shown in Figure 2.4. All of the memory banks are interconnected in a mesh network way with on-chip communication. The shared memory is connected with all of the memory banks, which enables high internal bandwidth. To implement computation and storage, a number of digital components should be orchestrated with ReRAM crossbars. In a memory bank, an integrated controller with computing capability is used to decode customized operations, and to provide control signals for all the peripheral circuits. WDD is the wordline decoder and writing driver, and is used to access data. For graphs, it can map the adjacency lists to ReRAM crossbars, as well as set the input voltage for traversing graphs. Sample and hold (S+H) receives the bitline current and holds the sample analog values before feeding it to a shared sense amplifier (SA). The SA implements the similar function of analog-to-digital converter (ADC), which provides high precision control [64]. Multiple ReRAM crossbars share one SA component in order to reduce area overhead and energy consumption. Due to the limited precision (up to 7-bit per cell) of ReRAM cells [5], shift-and-adds (S+A) is provided to support higher precision for large-scale graphs. The aggregated results are then stored in the cache for processing.

2.3.2 Mapping Graph to A ReRAM Crossbar

The adjacency list with the CSR format of a graph partition is mapped to a ReRAM crossbar, one cell after another. The adjacent vertices of one vertex may involve multiple rows if it is a hub vertex whose out-degree is large [71]. To indicate the starts of the corresponding vertices on a ReRAM crossbar, a pointer is needed to indicate the row index and column index of the ReRAM crossbar, which is similar to the indices in the adjacency list array. We use the location pointer [*row index, column index*] to identify the adjacency lists mapped in the ReRAM crossbars. Both adjacency lists of a graph partition and location pointers are stored in the same ReRAM crossbar to maintain correspondence. After the mapping of the neighbors for one vertex is complete, the location pointers for this vertex are also stored in

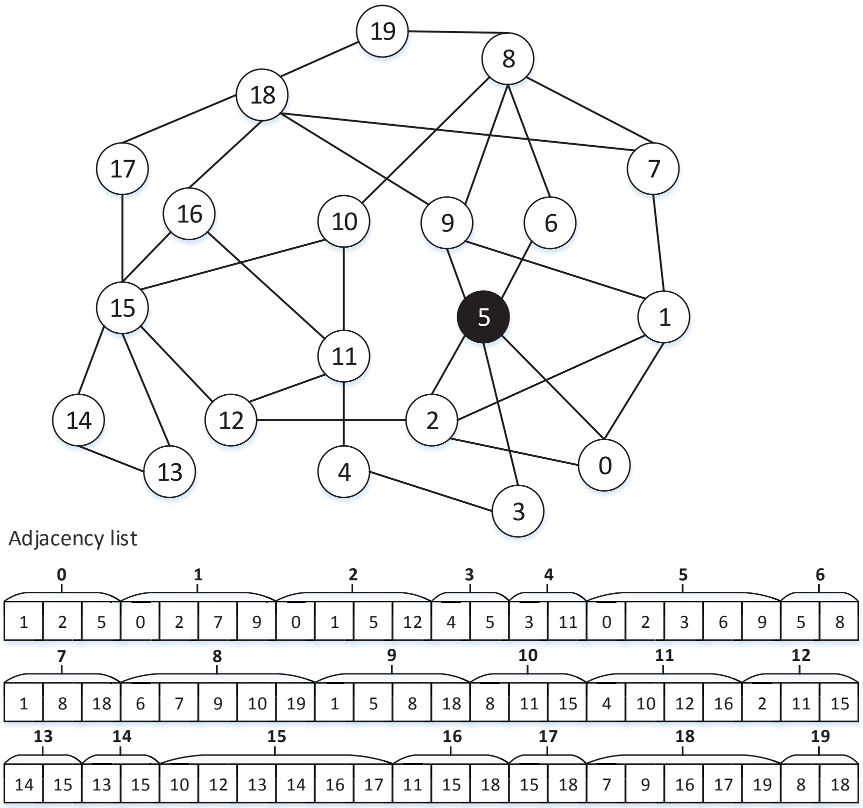


Figure 2.5: An example graph H with its adjacency list.

the same ReRAM crossbar, and then the available space for both the adjacency lists and the location pointers is re-computed. When there is not enough space on the current crossbar, the mapping moves to the next crossbar.

Figure 2.5 shows an example of graph H with 20 vertices and 66 edges, and its adjacency lists. Suppose that there are 6×6 ReRAM cells in one crossbar. Each vertex occupies one cell, and location pointers are stored every two cells for a row and a column index, respectively. The graph bank A demonstrates the mapping from *Vertex 0* to *Vertex 5* in Figure 2.6. When performing the mapping of *Vertex 5* that has five neighbors (indicated by red color), the third row in this ReRAM crossbar does not have enough cells to store all of the adjacent vertices of *Vertex 5*, so its neighbors *Vertex 6* and *Vertex 9* are stored in the fourth row. In this crossbar, the last two rows store the location pointers for this graph partition. The location pointer of *Vertex 5* can be attained by calculating the offset from the starting location pointer: $[(Vertex\ number \times 2)/row\_size + starting\_row, (Vertex\ number \times$

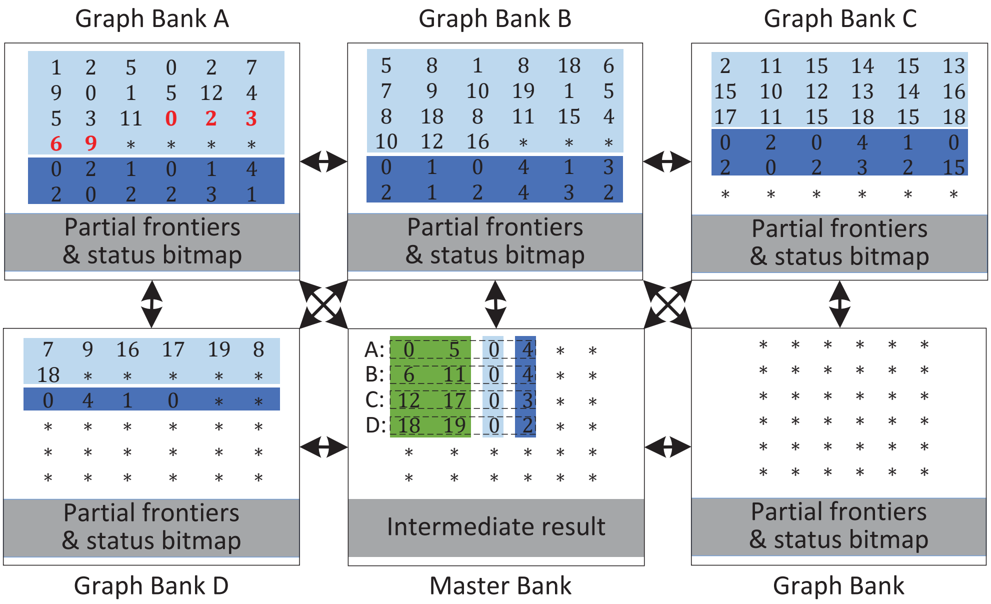


Figure 2.6: The layout of graph H involved with multiple banks.

$2)\%column\_size]$, where row_size and $column\_size$ refer to the dimensions of cells in the ReRAM crossbar, and $starting\_row$ denotes the starting row number for location pointers. In order to completely obtain the adjacent vertices of *Vertex 5*, we also need to get the location pointers of its prior vertex. The location pointers of *Vertex 4* are $[2,\ 2]$, so the cells after location $[2,\ 2]$ to $[3,\ 1]$ are adjacent vertices of *Vertex 5*.

2.3.3 Graph Layout in Multiple Crossbar Arrays

Multiple ReRAM memory banks are involved for large-scale graphs with several millions vertices and edges due to the scale limitation of a ReRAM crossbar (maximum 1024*1024 cells). The adjacency lists of a graph is persistently mapped to multiple ReRAM-based memory banks, and one memory bank is selected to be the master bank for recording metadata and scheduling expansion tasks. After finishing the adjacency list mapping in one graph bank, the master bank records the metadata information of the graph bank, including the vertex range information and the starting row number.

This is well illustrated with the example shown in Figure 2.5 and Figure 2.6. The adjacency lists of graph H are mapped to four memory banks. For generality and simplicity, in this example, we map the original adjacency lists with a greedy scheme to as many ReRAM cells as possible. We focus on how the multiple banks coordinate for storage and computation. Due to the symmetrical structure, here we arbitrarily select a memory bank as the master bank. The graph banks A, B and C store the adjacency lists of six vertices, respectively. The adjacency lists are mapped on the first four rows in graph banks A and B, while they are mapped on the first three rows in graph bank C. Correspondingly, the location pointers are stored from the fifth row in graph bank A and B, and from the fourth row in graph bank C. The graph bank D only stores the adjacency lists of the last two vertices of graph H, and the location pointers are stored in the third row. Considering the flexibility of this data organization, the master bank uses a 4-tuple to record the metadata of involved graph banks: (s_v, e_v, r_adj, r_ptr). s_v and e_v represent the vertex range from the starting vertex to the ending vertex, and r_adj and r_ptr denote the starting row number of the adjacency lists and location pointers. In this example, the first row data $(0, 5, 0, 4)$ in the master bank indicates that the vertex range of graph bank A is from *Vertex 0* to *Vertex 5* (marked in green), and "0" and "4" refer to the starting row number of the adjacency list and the location pointer respectively. In RPBFS architecture, a graph bank contains several ReRAM crossbars, the master bank records the tuples for each ReRAM crossbars of a graph bank in the same row. This graph mapping scheme for RPBFS incurs few extra storage overhead compared to the original CSR format. The metadata on the master bank is only related to the number of involved graph banks. The storage overhead consumes less than 0.1% of space for a graph with several millions vertices and edges.

The RPBFS supports customized graph distributions. A graph can either be distributed on more graph banks to improve the bank-level processing parallelism in the same partitioning method, or can be distributed with different partitioning methods, such as partitioning to balance the number of vertices, outgoing edges, and so on.

2.4 Breadth-first Search on ReRAM-based Main Memory

In this section, we describe the BFS traversal algorithm employed for our RPBFS architecture with the accompanying pseudo-code. Implementing a BFS traversal there involves three stages from data storage to memory execution: graph mapping, graph initialization, and graph traversal.

2.4.1 Graph Mapping

A graph is persistently mapped on multiple ReRAM-based memory banks, as discussed in Section 2.3. If there are some updates to a graph, both the adjacency lists on the graph banks and the metadata information on the master bank should be updated. In this work, we only study on the static graph which consists of a fixed sequence of vertices and edges.

2.4.2 Graph Initialization

In the initialization stage, the status bitmap and level array for each vertex are initialized with the source vertex. The status bitmap is a bitwise array where a single bit is used to indicate whether one vertex is visited or not, and the level array maintains the shortest distances of all the vertices from the source. Algorithm 2.4.1 and Algorithm 2.4.2 give the pseudo-code of the initialization stage for the master and graph banks, respectively. Before starting a graph traversal, the master bank sends the corresponding information (line 1), including the vertex range information and the starting row number, to the graph banks involved. Then, it creates a full status bitmap (FSB) and a level array ($Level$) in the shared memory, and updates them with the source vertex s (lines 2-5). The last step is to initialize the number of frontiers ($frontier\_num$) for determining whether or not the traversal has been completed. The number of frontiers is initialized to 1.

The initialization stage in a graph bank is presented in Algorithm 2.4.2. Each graph bank saves its own vertex range information and the starting row numbers related to all of its ReRAM crossbars to the registers for accelerating an expansion (line 1). According

```
Algorithm 2.4.1 BFS in a master bank.
Input: Source vertex $s$.
Output: Level array $Level$.
BFS initialization:
1: Send the corresponding vertex range and the starting row number to each graph bank;
2: Full status bitmap in Shared Memory $FSB \longleftarrow \varnothing$;
3: Level array in Shared memory $Level \longleftarrow \varnothing$;
4: $FSB[s] \longleftarrow 1$;
5: $Level[s] \longleftarrow 1$;
6: The sum of frontiers for the next level expansion $frontier\_sum \longleftarrow 1$.
BFS traversal:
1: while $frontier\_sum \neq 0$ do
2:     Send $start\_cmd$ to all graph banks;
3:     $frontier\_sum \longleftarrow 0$;
4:     Waiting for $frontier\_num$ from each graph bank;
5:         for each $frontier\_num$ do
6:             $frontier\_sum \longleftarrow frontier\_sum + frontier\_num$;
7:         end for
8: end while
```

to the vertex range information, each graph bank generates its own empty partial bitmaps for exploring its own graph partition (lines 2-4). The level information (c_level) is also initialized at this stage (line 5).

2.4.3 Graph Traversal

In this stage, the master bank collaborates with the graph banks to perform BFS traversal. It implements the synchronization barriers by sending a command signal to notify the status of the traversal tasks, as shown in Algorithm 2.4.1. In the master bank, if the number of frontiers for the next level of expansion is not equal to zero, the master bank sends a

Algorithm 2.4.2 BFS in a graph bank.

BFS Initialization:

```
Save the vertex range and the starting row number in the register;
Partial prior status bitmap in cache $PPSB \longleftarrow \varnothing$;
Partial status bitmap in cache $PSB \longleftarrow \varnothing$;
Partial frontier bitmap in cache $PFB \longleftarrow \varnothing$;
Current level information $c\_level \longleftarrow 2$.
```

BFS Traversal:

```
Waiting for $start\_cmd$ from master bank;
$PSB \longleftarrow FSB$;
$PFB \longleftarrow PSB \otimes PPSB$;
The number of frontiers in $PFB$ $frontier\_num \longleftarrow 0$;
for each $u$ is non-zero in $PFB$ do
    $frontier\_num \longleftarrow frontier\_num + 1$ ;
    Get location pointer of $u$ in crossbar arrays;
    Get adjacent vertices $\{A\}$ in crossbar arrays;
    for each $v$ in $A$ do
        if $FSB[v] \neq 1$
            $FSB[v] \longleftarrow 1$;
            $Level[v] \longleftarrow c\_level$;
    end for
end for
$PPSB \longleftarrow PSB$;
$c\_level \longleftarrow c\_level$ +1;
$PFB \longleftarrow \varnothing$;
Send $frontier\_num$, $finish\_cmd$ to the master bank.
```

$start\_cmd$ to inform all of the involved graph banks to perform a graph traversal. After resetting the $frontier\_sum$, the master bank waits until it receives the number of partial frontiers $frontier\_num$ from all of the graph banks (line 4). Summing up those numbers, the master bank determines whether or not the graph traversal has been completed (line 6).

Graph banks carry out the execution of expansions at this stage, as shown in Algorithm 2.4.2. In each iteration, once graph banks receive a $start\_cmd$, they access shared memory without memory collision to fill the partial status bitmap PSB with FSB (line 2). The size and offset of the PSB are determined by the vertex range information that has been assigned from the master bank at the initialization stage. After that, graph banks perform the (PSB $\otimes$ $PPSB$) operation to generate frontiers in the current level (line 3). For example, there are six vertices stored on a graph bank. Assume that the prior partial status bitmap $PPSB$ is "001101" in the preceding level, and the latest partial status bitmap PSB is "101111". Then the frontiers in the current level PFB is "100010" after executing ($PSB \otimes PPSB$), which means that the first and fifth vertex are valid frontiers. For each non-zero in PFB, the graph bank issues an expansion task that involves attaining location pointers and attaining adjacent vertices (lines 7-8). The graph banks containing the valid frontiers then activate the corresponding wordlines of the ReRAM crossbars to explore the neighbors of the frontiers. For each newly visited neighbor, the next step is to update status bitmap FSB and the level array $Level$ in the shared memory (lines 9-12). Since BFS inspects an adjacent vertex with the shortest level, the graph bank will examine whether a neighbor of the frontiers is being visited for the first time. If so, the graph bank marks it as a visited vertex by updating the status bitmap, as well as recording its level information. After finishing all of the expansion tasks of the valid frontiers, the graph bank replaces the $PPFB$ with the PSB, and adds one to the level information (lines 15-16). The final step is to send the number of partial frontiers $frontier\_num$ to the master bank. After completing the whole graph traversal, the level array is the output, which can either be transferred to other memory or flushed to local ReRAM memory for further use.

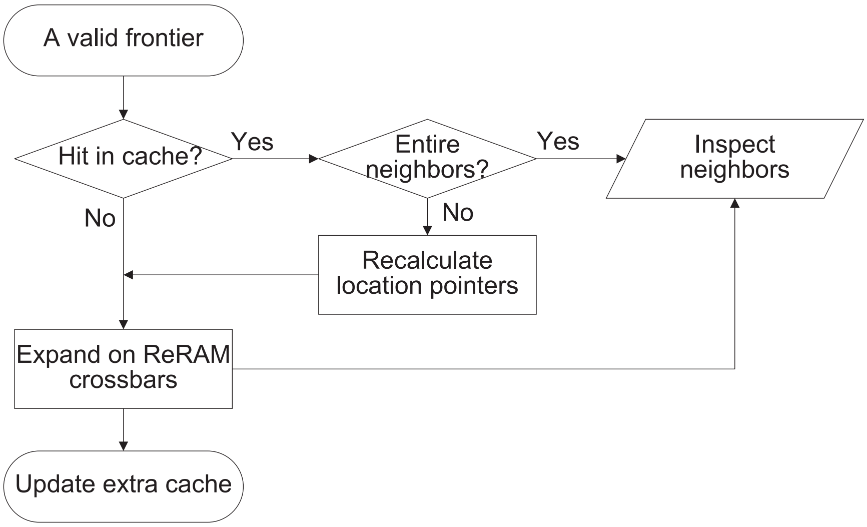


Figure 2.7: The workflow of the extra vertex cache.

2.4.4 Extra Vertex Cache

Besides attaining the adjacent vertices of a specified vertex, the partial or entire adjacent vertices of its contiguous frontiers can also be discovered by activating wordlines, as discussed in Section 2.2.4. Figure 2.7 presents the workflow of the extra vertex cache. RPBFS caches the extra adjacent vertices with the corresponding location offset and sum. The graph bank will first check whether the current frontier has been cached when starting a new expansion task. If it hits, the controller recalculates the location pointers of the current frontier according to the cached adjacent vertices. If only a portion of the neighbors is cached, then the graph bank will attain the rest of the neighbors by activating wordlines in ReRAM crossbars. Since graph banks explore the frontiers in order, the contiguous frontiers can benefit from extra vertex cache. We use FIFO as the cache replacement policy to manage the extra vertex cache.

2.4.5 Software-hardware Interface

RPBFS provides application programming interfaces (APIs) for developers so that they can easily access and perform graph traversal on the RPBFS. Table 2.1 gives a detailed description of the functions for three stages from data storage to memory execution.

Table 2.1: Software-hardware interface of RPBFS.

Software-Hardware Interface	Function description
$Map\_graph(G(V,E), memory\_address, bank\_address)$	map a graph to graph banks
$Put(memory\_address, bank\_address, size, vertex)$	transfer data to the destination address
$Get(memory\_address, bank\_address, size, vertex)$	attain data from the destination address
$Config\_graph(G(V,E), source, bank\_address)$	initialize a graph with the source vertex
$Run\_graph(G(V,E), bank\_address)$	start a graph traversal from the master
$Output\_graph(G(V,E), memory\_address, bank\_address)$	further process the output

In the graph mapping stage, the function Map_graph is used to map a graph G to the memory banks. The assigned memory bank works as the master bank for this graph. The adjacency lists of a graph are provided to the master bank, and then routed to the proper graph banks. In order to persistently map the graph to the memory banks, the Put and Get calls are implemented for copying and attaining the adjacency lists to and from a graph bank, respectively. If the destination graph bank has no space to store the adjacency lists, the master then picks another graph bank in which to store it.

In the configuration stage, the $Config\_graph$ function is provided to initialize the status bitmap and level array with the source for an assigned graph.

Finally, the function Run_graph is provided to perform graph traversal on RPBFS. The processing result function $Output\_graph$ is invoked to further process the output.

2.4.6 Limitations

Our proposed RPBFS has several limitations. First, RPBFS only considers the mapping and distribution for unweighted graphs. It requires additional storage footprints to keep the weight information associated with edges for weighted graphs. Second, RPBFS relies on a master bank to implement the level-synchronized breadth first search. The synchronization cost among all the graph banks varies depending on the graph distributions. We only consider a graph with several millions vertices and edges in this work, which needs dozens of graph banks with 4 crossbars (each has 1024*1024 cells). Third, RPBFS requires the elimination of shared memory collision at a fine-grained level. And the intermediate data updating operations for a graph traversal are executed on the shared memory, the wear-out of ReRAM will not be a concern. Finally, RPBFS only implements the breadth-first search algorithm; other graph traversal algorithms such as depth-first search can also be achieved on the proposed architecture but need new dedicated algorithm implementations.

2.5 Performance Analysis

In this section, we formulate the performance analysis of a graph traversal on the RPBFS architecture we developed in previous sections. Since BFS is a memory-dominant algorithm, we utilize the latency of the memory access operation to determine the total execution time on the RPBFS. The graph traversal performance is determined by several parameters, including graph distribution and hardware configuration. Some design choices are provided by our RPBFS; thus, we propose this analytical performance model to analyze the performance. The analysis can provide us with some insights to identify opportunities for improvement. Table 2.2 and Table 2.3 list the major notations used in this analysis.

2.5.1 Performance Model

We focus on the key factors of the RPBFS, so we assume that there is no conflicts involving parallel accesses to the different addresses in the shared memory, to avoid complexity [146]. Graph banks in one iteration begin by attaining their own frontiers from the shared memory, they then concurrently execute the tasks involved in expanding the valid frontiers, and finally finish the tasks of inspecting and updating the status array to the shared memory. Therefore, we generally analyze the performance of one graph bank to model the RPBFS performance. There are three steps to completing the expansion tasks in one traversal iteration in one

Table 2.2: Analysis terminology

Term	Definition
$G(V, E)$	Graph G with V vertices and E edges
L	Maximum number of levels for which BFS operates
F_{kl}	Number of the frontiers in the kth graph bank in level l
A_i	Number of the adjacent vertex of ith vertex
S_{il}	Number of the adjacent vertex of ith vertex in level l. $S_{il} : 0, A_i$
V_k	Number of vertices whose adjacency list is stored in the kth graph bank
E_k	Number of edges stored in the kth graph bank
S	Number of ReRAM cells of a wordline in a crossbar
P	Number of ReRAM cells to represent a vertex
B	Shared memory bandwidth (bit/s)

graph bank. First, a graph bank attains frontiers from the status bitmap. Once there are valid frontiers in the current level, the expansion tasks are issued. The neighbors of the frontiers are explored on the ReRAM crossbars. At the final state, the status and the level array are inspected and updated in the shared memory.

According to the Table 2.2 and Table 2.3, we can easily get Equation 2.1. Each vertex can be a frontier only once in the whole traversal process, so the total number of frontiers is the number of vertices in a graph. Similarly, since the neighbors of a vertex (A_i) will be traversed only once, if the vertex is not a frontier in the current level, then we assume that it has no adjacent vertex at this level. If it is a frontier, all of its adjacent vertex will be visited. Therefore, the number of edges in a graph bank is the sum of the neighbors of the frontiers in the whole traversal process.

$$V_k = \sum_{l=1}^{L} F_{kl}, E_k = \sum_{i=1}^{V_k} \sum_{l=1}^{L} S_{il} \tag{2.1}$$

Table 2.3: Latency terminology

Term	Definition
t_{reram}	Latency of activating one ReRAM wordline and passing peripheral circuits
t_{cache}	Latency of accessing a vertex in the cache of graph banks
t_{pro_cache}	Latency of processing one bit in the cache of graph banks
t_{pro}	Latency of processing one bit in the shared memory
t_{item}	Latency of processing one item of an array in the shared memory
t_{syn}	Average synchronization latency of graph banks in each level

Attaining frontiers: Attaining frontiers can be divided into two parts in one traversal iteration. First, the graph banks access the shared memory without collision to get the corresponding status bitmap which matches the vertex range. The kth graph bank has V_k vertex, so it costs V_k/B to attain the status bitmap from the shared memory. After that, graph banks perform XOR operations to attain frontiers with the current status bitmap and prior status bitmap. Since processing XOR operations is linear with the number of vertex in a graph partition, we can model the runtime for attaining frontiers in level l to be:

$$T_{attain} = V_k/B + V_k t_{pro\_cache} \tag{2.2}$$

Exploring adjacent vertices: The expansions of the frontiers are capsuled in a ReRAM graph bank, and the latency of expansions is bounded by the number of frontiers. According to Algorithm 2.4.2, we estimate the time spent on ReRAM crossbars in two parts: attaining location pointers and attaining adjacent vertices.

Attaining the location pointers is needed before exploring the neighbors of a frontier in the RPBFS. Due to the extra vertex cache, some of the location pointers are attained from the ReRAM crossbars, while the rest are attained from the cache. Therefore, we can estimate the total latency for attaining the location pointers in one iteration as:

$$T_{location} = \alpha F_{kl} t_{cache} + \beta F_{kl} t_{reram} \tag{2.3}$$

Here, α and β represent the proportion of attaining location pointers from the ReRAM crossbars and the cache in a level.

Similarly, a graph bank attains the adjacency lists of the frontiers either from the ReRAM crossbars or from the extra vertex cache. The cost of passing the ReRAM crossbars depends on the number of times that wordlines are activated, which is strongly associated with the number of adjacency list of the frontiers in the current level. Suppose that a vertex needs P ReRAM cells to store. In level l, the total number of ReRAM cells involved will be P times of the number of adjacency list of the frontiers, which can be expressed with $\sum_{i=0}^{V_k} S_{il} \times P$. The straightforward way of calculating the number of times that the wordlines are activated is $\sum_{i=0}^{V_k} S_{il} \times P \div S$. However, the adjacency list of the ith frontier may be distributed more than $S_{il} \times P \div S$ wordlines. For example, two neighbors of a vertex may be placed on two rows in the ReRAM crossbar. Thus, we time a coefficient θ (1, 2) before the $\sum_{i=0}^{V_k} S_{il} \times P \div S$. Furthermore, the adjacency list can also be attained from the cache, and the time cost is linear with the number of adjacency lists. Therefore, we can estimate the runtime for attaining the adjacency lists in level l to be :

$$T_{adjacency} = \frac{\gamma \theta P \sum_{i=1}^{V_k} S_{il} t_{reram}}{S} + \delta \sum_{i=1}^{V_k} S_{il} t_{cache} \tag{2.4}$$

The γ and δ represent the proportion at which the adjacent vertices are attained from the ReRAM crossbars and cache in a graph bank, respectively.

Updating status bitmap and array: When an inspection is issued for a newly visited neighbor, both the status bitmap and level array will be updated in the shared memory. Since the checking status bitmap is antecedent, the number of checking operations in level l equals to the number of the neighbors of the frontiers. If a neighbor has been visited in previous levels, then the graph bank skip the update operation. The status bitmap and level array are updated only for newly visited neighbors, so the runtime for an inspection and update in level l can be modeled as follows:

$$T_{update} = \sum_{i=1}^{V_k} S_{il} t_{pro} + \eta(\sum_{i=1}^{V_k} S_{il} t_{pro} + \sum_{i=1}^{V_k} S_{il} t_{item}) \tag{2.5}$$

Here, η ($0<\eta\leq1$) represents the proportion of newly visited neighbors in the current level.

Based on above analysis, we can estimate the total execution time for traversing a graph in the RPBFS architecture. Since all of the graph banks involved perform traversal in a parallel way, the total execution time of a graph traversal is the sum of the maximum latency of all of the graph banks in each level. For an arbitrary graph bank, we use t_{syn} to represent the average synchronization time among all of the graph banks in one iteration. The total latency to traverse a graph in the RPBFS architecture would then be:

$$\begin{aligned} T_{total} &= \sum_{l=1}^{L} (T_{attain} + T_{location} + T_{adjacency} + T_{update} + t_{syn}) \\ &= V_k(\frac{L}{B} + Lt_{pro\_cache} + \alpha t_{cache} + \beta t_{reram}) \\ &+ E_k(\frac{\gamma\theta P t_{reram}}{S} + \delta t_{cache} + t_{pro} + \eta t_{pro} + \eta t_{item}) + Lt_{syn} \end{aligned} \tag{2.6}$$

2.5.2 Performance Analysis

The above model can help us to identify bottlenecks and provide opportunities for improving the RPBFS. Typically, the graph bank with more vertices and edges has less synchronization latency compared to the other banks, and the level L can be determined with the source vertex. According to Equation 2.6, the latency of traversing a graph is mainly determined by V_k and E_k in the RPBFS architecture, which is linear with the number of vertices and edges of a graph partition. We can see that the traversal has a complexity of $O(V_k + E_k)$, which matches our expectation that the partial graph traversal is wrapped in a graph bank. In contrast, conventional graph traversal solutions have a complexity of $O(V+E)$, or $O(V)$ for sparse graphs. Our proposed framework reduces the time complexity. Moreover, following the above model, it is easy to see that the improvements in performance can be achieved through a better distribution of graphs that are evenly partitioned. A straightforward way of partitioning graphs is to balance both the number of vertices and the number of edges among the graph banks.

2.6 Experimental Evaluation

In this section, we evaluate our RPBFS design. We first introduce the experiment methodology. Then, we present the performance and energy results.

2.6.1 Methodology

We compare our RPBFS with the state-of-the-art CPU-based parallel implementation and GPU-based solution Enterprise [11] [71]. The RPBFS is modeled by heavily modified N-VSim to simulate the peripheral circuit [30], as well as to implement the traversal scheme attached by a controller, which provides the control signals to all of the peripheral circuits. We modify the simulator as a trace-based system to evaluate its performance. CPU-based implementation is a multithreading and direction-optimizing solution with OpenMP runtime library [27]. Enterprise implements efficient threads scheduling and unique memory hierarchy on GPU platform. Both CPU-based and GPU-based solutions have the fitting memory and computation capability for the test workloads.

The configurations of the RPBFS architecture and detailed configurations of other platforms, from which the related timing parameters are derived [19] [113] [82], are illustrated in Table 2.4. There are four ReRAM crossbars per graph bank, each crossbar contains 1024 * 1024 ReRAM cells, and we assume that all of the cells have the same properties without non-uniform drops in voltage [145]. The ReRAM cell is conservatively assumed to be a 4-bit MLC, and we use eight cells to represent a vertex and four cells to represent a location pointer for large-scale graphs. The high-R state (HRS) and low-R state (LRS) are set to 2500KΩ and 50KΩ respectively, and the read and write voltage are set to 0.7V and 2V

Table 2.4: The configurations of RPBFS architecture and hardware.

Controller	16 registers; one Core at 1.2GHz
Cache	512KB
Shared Memory	16MB
Internal Bus	50GB/s

ReRAM-based memory	16 Banks/chip; 4 Crossbars/Bank;
	1024*1024 Cells/Crossbar;
	tRCD-tCL-tRP-tWR 18-9.8-0.5-30 (ns)

CPU Cores	Inter Core2 Q9550 with 2.83GHz
CPU L1 Cache	32KB SRAM
CPU L2 Cache	6144KB SRAM
Main Memory	4GB with two channels
Graphics Card	GTX TITAN X with 3072 CUDA cores

respectively. The cell currents of an LRS and HRS ReRAM are 40uA and 2uA, respectively. The energy cost of reading and writing are 1.59pJ and 5.53nJ. To estimate the energy and area cost of other digital components in the RPBFS, we use the data from [113].

All of the graphs for our evaluation are represented using a compressed sparse row format. We perform a sorting operation on the vertex. This pre-processing operation does not change the graph topology. The graph data is loaded into the ReRAM memory bank, DRAM memory, and GPU memory ahead, respectively, which excludes the I/O time from secondary storage device. The time switch from top-down to bottom-up on traditional platforms is triggered by the ratio of hub vertices, and an integer array is maintained in the memory to track the status of each vertex. The timing starts when the source vertex is given, and ends when the search is completed. We use traversed edges per second (TEPS) as the metric for the traversal performance. We also compare the energy consumption of the RPBFS with other platforms for performing graph traversal. The power consumption of GPU is measured

Table 2.5: Graph specification

Graph Name (Abbr.)	# Vertices (M)	# Edges (M)	Proportion of Contiguous vertices	Directed
Enron (ER)	0.04	0.37	23.8%	N
Slashdot (SD)	0.08	0.95	12.9%	Y
Wikipedia (WT)	2.39	5.02	7.2%	Y
Web of Berkeley&Stanford (WB)	0.69	7.60	36.5%	Y
Amazon (AM)	0.40	3.20	70.6%	Y
RoadNet (RA)	1.96	5.53	21.1%	N

using the NVPROF [85], and the power consumption of CPU and DRAM is estimated using the Intel Power Gadget for Linux [78].

We use six graphs with different kinds of connections as the workloads [62], as shown in Table 2.5. The Amazon product co-purchasing network (AM) graph has 400 thousand vertices and 3.20 million edges, while the Wikipedia talk network (WT) has 2.39 million vertices and 5.02 million edges. The web graph of Berkeley and Stanford (WB) has more than 7.60 million edges, but only 685 thousand vertices. Two related small graphs are also tested: the email communication network from Enron (ER) which only has 36 thousand vertices and 367 thousand edges, and the Slashdot social network (SD) which has 82 thousand vertices plus 948 thousand edges. The other undirected graph road network of California (RA) has 1.96 million vertices and more than 5.53 million edges.

2.6.2 Evaluation Results

We first show the speedup in the performance of our RPBFS algorithm compared with other direction-optimizing solutions on traditional platforms. The direction-optimizing solutions have been proved that they perform better than the top-down algorithm [11]. We then show

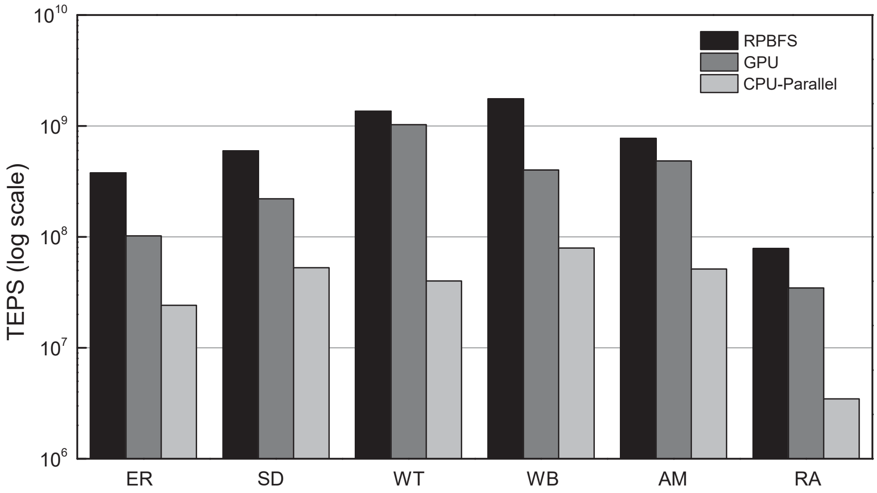


Figure 2.8: Performance of RPBFS and direction-optimizing GPU-based and CPU-based solutions.

the scalability of RPBFS by measuring the performance of the different scales of ReRAM crossbars. We map the graphs to the memory banks with the scales of 256*1024, 384*1024, 512*1024, 768*1024 and 1024*1204 ReRAM cells, respectively. After that, we compare the improvement in performance by employing different data partitioning schemes. Finally, we evaluate the energy that was consumed.

• **Traversal Performance**.

Figure 2.8 compares the traversal performance of our RPBFS solution, the GPU-based framework Enterprise, and CPU-based 64-thread parallelism implementations, respectively. The RPBFS performs up to 16.0× better than the Enterprise, and up to 33.8× better than the CPU-based solution.This is because our RPBFS wraps the adjacency list access within the memory banks, so as to reduce the data movement overhead. The data movement among the memory banks is related only to the synchronization of vertex bitmaps, and is far less than the that in other solutions. Tracking the status of each vertex also incurs the data movement overhead in both three platforms. As shown in Figure 2.8, RPBFS achieves a 33.8× and 22.2× speedup over CPU-based solution in graph WT and RA, because the status array for those graphs with a large number of vertex needs to be maintained and inspected by passing through CPU memory hierarchy. RPBFS achieves a smaller speedup in WT, AM, and RA by 1.32×, 1.59× and 2.27× than the Enterprise, this is because the Enterprise optimizes the

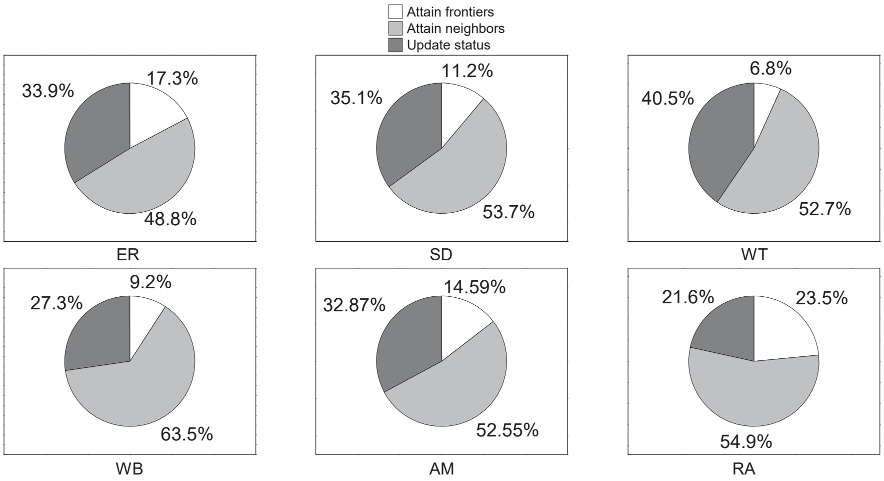


Figure 2.9: The breakdown of the execution time of BFS in RPBFS architecture.

Table 2.6: Performance improvement by employing extra vertex cache

Graph (Abbr.)	ER	SD	WT	WB	AM	RA
Percentage improvement	1.62%	0.47%	0.31%	1.93%	2.98%	2.31%

scheme of scanning the whole status array which resides in GPU global memory.

Figure 2.9 presents the breakdown of the execution time of BFS traversal on the RPBFS. The execution time for each graph is divided into three parts: attaining frontiers, attaining neighbors, and inspecting status. We can see that attaining neighbors occupies the majority of the execution time in all of the workloads, averagely it takes 54.4%. Moreover, it costs 23.5% of the total time to attain frontiers in the workload RA. This is because RA has the highest diameter so that it costs the graph banks a portion of time to access the shared memory for attaining the frontiers of each level. In the WT, it costs 40.5% of the total time to update status, since a small portion of vertices own the majority of edges, so that the cost of inspecting and updating is somewhat higher than other workloads.

Employing extra vertex cache can further accelerate graph traversal. This is because extra vertex cache reduces the number of redundant activating wordline operations. Table 2.6 shows the performance improvement of RPBFS when extra vertex cache can retain up to

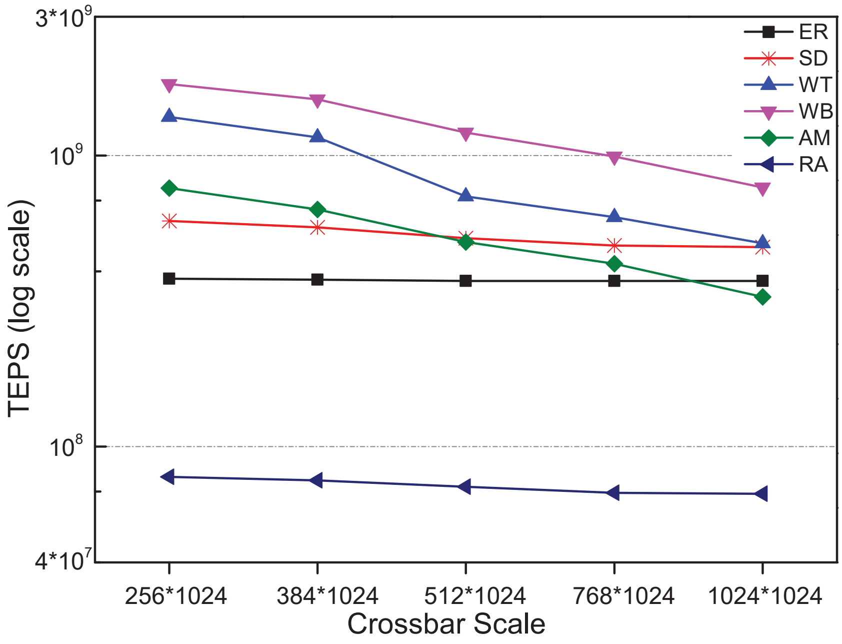


Figure 2.10: Performance with the scalability of RPBFS.

fifteen extra adjacent lists on the ReRAM crossbars with 1024*1024 scale. We can see that employing extra vertex increases the traversal performance by up to 2.98% compared to the RPBFS without cache mainly due to the contiguous neighbors of vertices. The directed graph WT has the lowest proportion of contiguous vertices, plus the simple cache replacement policy, it shows only slight performance improvement.

• **Scalability Performance**.

Figure 2.10 evaluates the scalability of RPBFS. When a graphs are mapped to the ReRAM crossbar with the scale decreasing from 1024*1204 to 256*1024 cells, more graph banks are involved. For the relatively large graphs WT and WB, the RPBFS with small scale crossbars outperforms those with a bigger scale up to 2.7× and 2.3×, respectively. The reason for this is that the diameter in these two graphs is small, so that the frontiers are sufficient in each level. Thus, more graph banks are involved in concurrently performing expansion tasks with the small scale crossbars. The same situation applies to AM due to the sufficient frontiers. By contrast, for another graph RA, where the proportion of vertex numbers and edge numbers accounts for around 35% and not enough benefits can be obtained from the small scale crossbars, the improvement in performance is only 1.5×. This is because most of vertices have few neighbors, so there are a limited number of frontiers in a level. For the workloads ER and SD, which have only several thousands of vertices and edges, the improvement in performance with small scale crossbars is almost imperceptible due to the low level of parallelism and the insufficient number of frontiers. It should be noticed that, involving more graph banks can improve bank-level parallelism, but the parallelism of the processing graph traversal still depends on the distribution of frontiers in each level. The optimal situation is one where the frontiers in each level can be evenly distributed on all the ReRAM crossbars, as the parallelism of the ReRAM crossbars can be fully utilized. However, it is not practical to distribute the optimal frontiers, since a graph traversal can start any vertex, and the topology of a graph is unknown.

• **Effect of Graph Distributions**.

According to the RPBFS performance model described in Section 2.5, the total execution time is strongly associated with the number of vertices and edges in an individual graph bank. Therefore, employing better data partitioning schemes can improve computing parallelism and minimize the synchronization time among all of the graph banks [55] [107] [18]. Since the adjacency lists are mapping to a graph bank with the greedy algorithm, the number of edges in each bank is almost equal. The optimized partitioning method is to balance the number of vertices, which means evenly spreading the vertices and edges to the graph banks. To show the effect of the graph partitioning schemes on the traversal performance, we reorder the vertex where the out-degree is in an ascending order, so as to generate the worst case (RPBFS+WC) and the optimized case (RPBFS+OC). In the worst case, the adjacency list of vertices will be mapped to the graph banks from the lowest number of out-degree vertices, so that the first graph bank owns the maximum number of vertices, while the last graph bank has the minimum number of vertices as a graph partition. By contrast, both vertices and the corresponding adjacency lists are almost evenly distributed on the graph banks in the optimized case, which is implemented by mapping the adjacency lists with the out-degree in ascending order from both ends. We evaluate the expansion time of three different graph distributions without the execution time of the partitioning. Figure 2.11 shows the results of the performance of three graph distributions. We can see that employing optimized graph

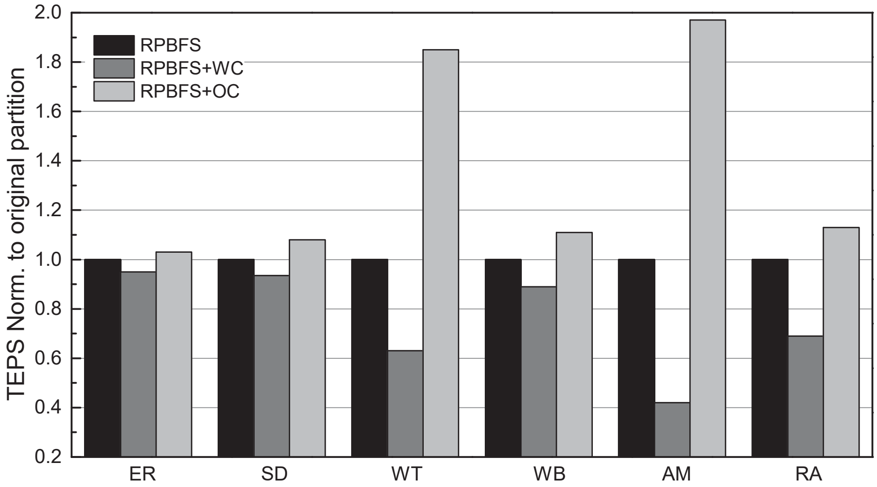


Figure 2.11: Performance improvement with different graph partitioning schemes.

distribution can further improve the performance of RPBFS on WT and AM, which have sufficient frontiers in each level, with the improvement being up to 1.85× and 1.97× that of the original scheme. The performance gap of AM between the optimized and the worst scheme aggravates to 4.7×, because the worst partition limits graph bank parallelism, and the traversal depth incurs more synchronization time, including the cost of scanning frontiers. The optimized scheme reduces the synchronization time and improves the graph bank parallelism to some extent, since all of the graph banks nearly have balanced vertices and frontiers. For the graphs ER and SD, due to the simple topology, better graph partitioning shows only a slight improvement in performance. We also validate our performance model by comparing values improvement ratio analyzed through our model with that of the experiments. The maximum number of vertices of graph RA partition mapped on a graph bank in the worst case is around 131 thousand, which is almost 1.9× than the average number of vertices in the optimized case, around 70 thousand. This situation also appears on the number of edges. From the result of RA in Figure 2.11, we can see that our model accurately estimates the performance within 8% difference. The results show that our model is an accurate analyzer of traversal performance for a number of graphs.

• **Energy and Area**.

The energy saving results of the RPBFS are presented in Figure 2.12. We see that RPBFS

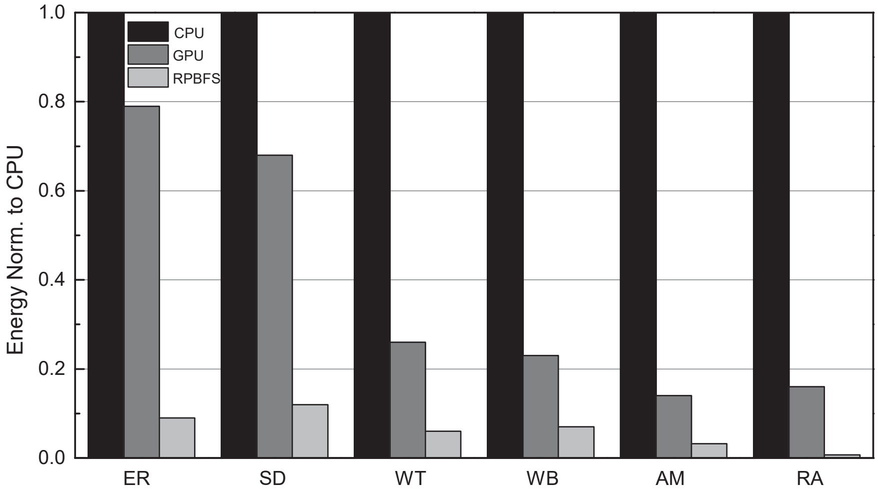


Figure 2.12: Energy saving results ($vs.$ CPU and GPU).

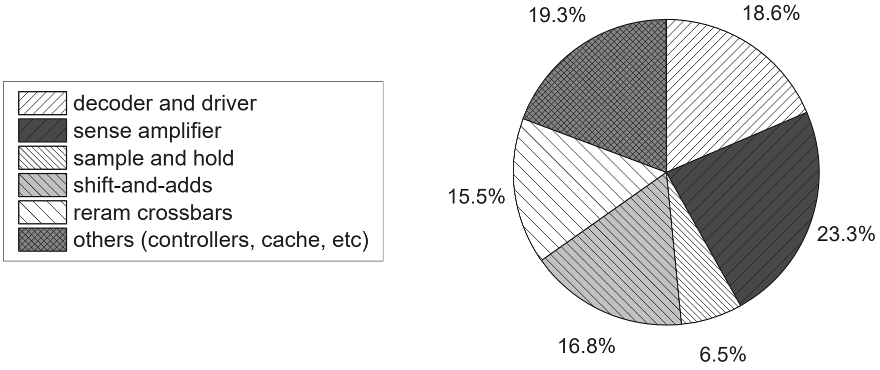


Figure 2.13: Area breakdown of RPBFS.

is more energy-efficient than other solutions due to less data movement overhead and the property of the ReRAM crossbars. The RPBFS consumes up to 142.8× and 11.4× less energy than the CPU and GPU solutions, respectively. For the small graphs ER and SD, the energy savings of RPBFS over the CPU solution are 11.1× and 8.3×, while GPU has energy savings of only up to 1.47× over CPU. For the large graph RA, the bigger diameter incurs a vast number of accesses from random neighbors. Thanks to the localization of processing expansions in the RPBFS architecture, the transfer cost and computation cost of RPBFS are reduced.

Figure 2.13 shows the area breakdown of a ReRAM memory bank. The ReRAM crossbars make up 15.5%, while the sense amplifier and the shift-and-adds which are the shared components make up 40.1% of the whole area.

2.7 Other Related Work

In this section, related work involving graph accelerators and architectural designs for data-intensive workloads is discussed.

Graph Accelerators. There are several existing studies on implementing graph accelerators. Disk-based GPUs are common platforms for accelerating graph traversals [147] [48] [71] [133] [79]. Hong et al. [48] proposed new methods for parallel breadth-first search implementations by efficiently utilizing memory bandwidth. Enterprise is a GPU-based solution that leverages the streamlined scheduling of GPU threads and workload balancing to improve the parallelism of the processors [71]. Merrill et al. [79] presented a BFS implementation on the GPUs with a memory-access-efficient data representation. Moreover, Zhang et al. [146] implemented a graph processing system on a FPGA-HMC platform based on the co-design and co-optimization of software/hardware. They optimized the bitmap scheme to reduce the memory access, and they proposed a performance model to generalize the total execution time of BFS. Graphgen is an FPGA-based framework that uses a vertex-centric model [84], while X-stream is an edge-centric approach to stream partitions for graph processing [104]. Although these studies have provided a considerable computation units, and have optimized the memory hierarchy to some extent, they still suffer from relatively long random accesses and short computations.

Some architectural works examined large-scale graph processing on processing-in-memory [88] [1] [120]. Tesseract maximized the available memory bandwidth by integrating PIM technology into 3D-stacked memory [1], while Ozdal et al. proposed an accelerator architecture to reduce irregular access patterns and asymmetric convergence [88]. GraphR utilized ReRAM crossbars to store graphs, and realize the massive parallelism to accelerate processing [120]. RPBFS is a distinct ReRAM-based PIM architecture from GraphR [120]. Instead of partitioning ReRAM memory into different regions, RPBFS integrates both storage and computation capability on ReRAM memory. Thus, RPBFS reduces the transfer cost. Moreover, RPBFS implements a level-synchronized graph traversal with compressed graph representations, while GraphR realizes it as a special case of single-source shortest path in a matrix way which needs to perform an extra graph conversion [25]. In addition, RPBFS aims at a specific graph application, and implements it on a dedicated architecture and the corresponding algorithm. GraphR is a general graph processing accelerator, thus it needs to consider difference access and mapping patterns.

Processing-in-memory. PIM integrates the computation units inside memory to reduce the data movement overhead. The concept of PIM has been proposed for many years. Because of the intolerable cost of the integration, the industry concentrated on off-chip memory. PIM is resurgent by putting logic layer into 3D-stacking memories where data resides, which can alleviate the bandwidth bottleneck between the logic and the memory, and reduce the energy cost [10] [147] [112] [148] [143] [3] [95] [80]. Zhu et al. [148] designed a 3D-stacked memory within the logic to accelerate graph algorithms by performing matrix multiplication. Similarly, Mirzadeh et al. [80] adopted a 3D-stacked logic-in-memory architecture for computation. PRIME is a PIM architecture to accelerate neural network applications [19], and it utilized the inherent matrix-vector multiplication capability of the ReRAM crossbars. ISSCC [113] and Pipelayer [119] are pipelined architectures with memristor crossbars to process computations. Ann et al. [2] proposed PIM-enabled instructions to implement processing in memory; this approach is compatible with existing cache coherence and virtual memory mechanisms.

2.8 Summary

In this chapter, we have proposed a novel ReRAM-based processing-in-memory architecture for breadth-first search, called RPBFS. RPBFS can accelerate large-scale graph traversals by reducing data movement overhead. Graph data movement are wrapped within graph banks. Benefiting from the data-mapping scheme in the ReRAM crossbars and the efficient graph traversal algorithms among memory banks, our architecture can effectively achieve improvement in traversal performance compared with other solutions, and significant energy saving under various workloads.

CHAPTER 3

OPTIMIZING RAID/SSD CONTROLLERS WITH LIFETIME EXTENSION FOR FLASH-BASED SSD ARRAYS

3.1 Introduction

SSD-based RAID arrays have been gaining ever-increasing prevalence in enterprise, such as XtremIO from EMC [31] and VSP from Hitachi Data System [126]. Compared with traditional HDD-based RAID arrays, SSD-based RAID arrays feature lower access latency, rack space and energy consumption. However, SSDs suffer aging problems. Flash blocks inside SSDs can only be experienced by a limited number of Program/Erase cycles before they are discarded. An accumulation of discarded blocks sacrifices the lifetime of an SSD. For example, 30% - 80% of SSDs in a data center develop bad blocks during their lifespan [47]. Parity updates incur extra writes in the RAID system, which aggravates the aging issue. Although existing techniques, such as delay writes and parity redistribution, can ease this problem, effective techniques are still urgently needed to extend the lifetime of SSDs in the RAID system. The interplay between RAID and SSD controllers which are the center controls of storage systems opens a new door for this issue through the exploitation of the special features of data.

The benefits of exploring data inaccuracies have been extended to non-volatile memories with increasing demand for persistently storing data. Small amounts of bit flips are acceptable for some applications. For instance, images or videos become prominent consumers of storage, and they can tolerate an error rate of around 10% [105]. Instead of changing data to imprecise in approximate computing, such as omitting mantissa bits in floating point operations, relaxing integrity constraints exhibits huge potential to significantly extend

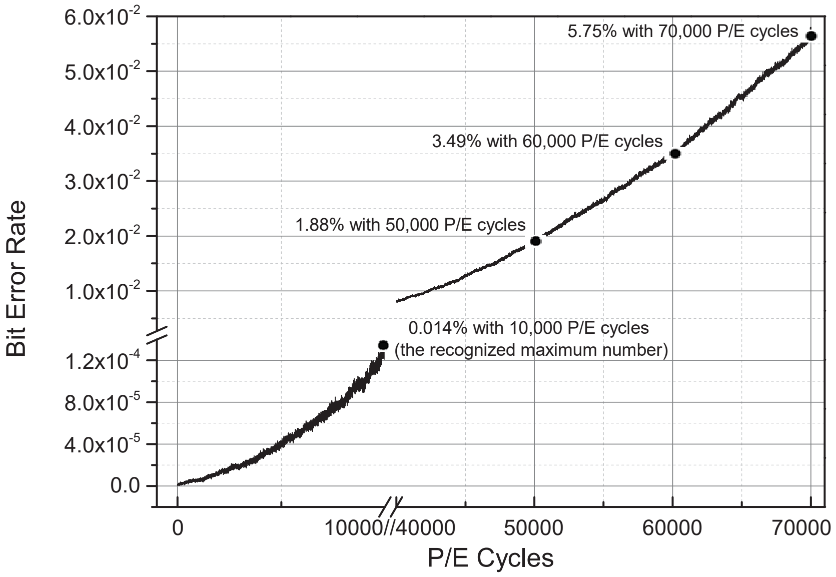


Figure 3.1: Higher error tolerance can extend the endurance of a NAND flash block.

the lifetime of flash memory. Figure 3.1 shows the experimental results we have conducted based on an MLC NAND flash chip [110]. It can be observed that the maximum error rates (obtained from the page with the highest error rate among all pages in a block) are 0.014% with 10,000 and 5.75% with 70,000 P/E cycles, respectively. The results demonstrate that the maximum endurance of a block can be extended by more than six times with more error-tolerance.

Controllers in an SSD-based RAID system are the key parts to improving system performance. RAID controller managing data and parity chunks is built on top of a number of SSDs. SSD controller mainly handles functions of address mapping, garbage collection and wear leveling, and the FTL in an SSD controller manages flash memory [98] [70]. Although SSD controllers have been well studied for improving the lifetime of flash memory [75], this is not the case with RAID controllers, which are not completely involved in extending the lifetime of SSDs. First, a RAID controller is unaware of the age conditions of SSDs, resulting in incompatible traffic, which in turn aggravates the aging rate of some SSDs. Second, a semantic gap exists between the RAID controller and the SSD controller. The semantic gap eliminates the interactions between the two controllers. For example, the parity distribution strategy only works in a RAID controller, and the garbage collection scheme is only controlled by an SSD controller. Third, simple storage management in both RAID and SSD controller ignores the special features of data. Many applications have a mix of error-free and error-relaxation data, and the tolerance of error-relaxation data varies. However, the stripping technique in RAID breaks up error-free/error-relaxation data groups, making it inefficient to explore the benefits of relaxing integrity constraints of flash memory.

In this chapter, we optimize the lifetime of SSD arrays via the interplay between RAID and SSD controllers. We propose a cross-layer lifetime optimization framework, **F**lash-**re**surr**e**ction **RAID** (FreeRAID). Our design rationale is to add a new phase, i.e., the exploitable phase, in the lifecycle of a block besides normal and bad phases. Exploitable blocks can be used to store error-relaxation data, even though uncorrectable errors will be persistent. With the goal of extending the lifetime of an SSD-based RAID array, FreeRAID combines the following two techniques. First, with the knowledge of physical blocks in SSDs, the RAID controller in FreeRAID efficiently allocates normal and exploitable blocks to serve data with different error-tolerances, and makes different types of data error-isolated. In addition, FreeRAID and the existing optimized RAID schemes can coalesce. Second, FreeRAID employs Adaptive-FTL to maintain performance and storage efficiency by dynamically switching FTL strategies. The lifespan of an SSD in FreeRAID is divided into three stages by the proportion of different types of blocks, and the space allocation and garbage collection schemes in FTL can be dynamically switched in accordance with the stage that an SSD stays in.

We conduct a series of experiments to evaluate FreeRAID compared with conventional RAID solutions and FTLs. Evaluations across a wide variety of workloads show that FreeRAID can significantly extend the lifetime of SSDs arrays and maintain I/O performance. To summarize, this chapter makes the following contributions:

- We explore exploitable blocks in SSDs to serve error-relaxation data for lifetime extension.
- We propose FreeRAID, which leverages the interplay between RAID and SSD controllers to extend the lifetime of SSD-based RAID arrays.

- We evaluate FreeRAID with various workloads on a simulator, and the results show that we can significantly increase the lifetime of SSD-based arrays by up to 2.17× compared with conventional RAID solutions and FTLs.

The rest of this chapter is organized as follows. The next section gives the basic background of this study. In Section 3.3, we describe exploitable blocks. We also describe the architecture of FreeRAID, and its two key techniques. The performance of FreeRAID in several key metrics is evaluated in Section 3.4. Finally, Section 3.5 discusses other related work, and Section 3.6 concludes this chapter.

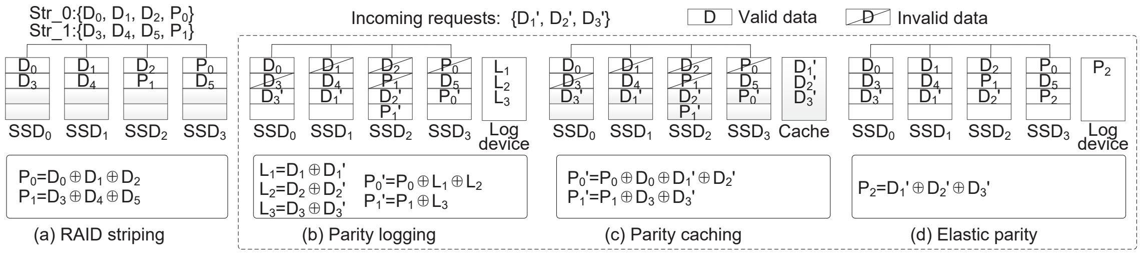
3.2 Background

This section briefly presents background on SSD-based RAID arrays, optimized RAID schemes and approximate storage.

3.2.1 SSD-Based RAID Arrays

An SSD is composed of an array of flash chips, which includes multiple blocks. A block is further divided into multiple pages [97]. Each block can endure a finite number of P/E cycles. The error detection/correction codes (EDC/ECC) in an SSD controller can detect and correct some of errors. Any uncorrectable failures even one bit will exhaust the whole block which is then marked as a bad block. An SSD is replaced when the number of bad blocks in it reaches a certain number [67].

An SSD-based RAID array consists of a RAID controller and multiple SSDs, and provides a virtual logical disk by combining the space of the SSDs. Striping and parity are two commonly-used RAID techniques. With striping, logically sequential data are divided into chunks. These chunks are stored on different physical devices and can be accessed concurrently. Parity is a redundancy-based protection scheme, by which parity data is generated based on a group of data chunks for data recovery. According to the redundancy, RAID schemes are categorized into several levels, such as RAID-5 and RAID-6. We will focus on



(a) RAID striping (b) Parity logging (c) Parity caching (d) Elastic parity

Figure 3.2: Three optimized RAID schemes.

RAID-5 in which one parity is generated for each stripe so that the system can tolerate one failed drive at any time. Figure 3.2(a) shows an example of a RAID-5 scheme. The data chunks $\{D_0, D_1, D_2\}$, and $\{D_3, D_4, D_5\}$ construct two stripes Str_0 and Str_1, with the corresponding parity chunks $\{P_0\}$ and $\{P_1\}$ generated to protect these stripes. The chunks in each stripe are placed on different physical devices.

3.2.2 Existing Optimized RAID Schemes

A primary issue arises when deploying RAID on SSD-based arrays: the write updates. In particular, parity chunks within the same stripe need to be updated for every data update. These extra updates incur more garbage collection operations and delay access performance on SSDs. In order to mitigate the update overhead, various optimized RAID schemes have been studied. These schemes have been grouped into three categories: parity logging, parity caching and elastic striping [122] [21] [66]. Figure 3.2 illustrates how three optimized RAID schemes work. Suppose that an incoming stream of requests $\{D_1', D_2', D_3'\}$ will update two stripes Str_0 and Str_1. With the parity logging scheme, as shown in Figure 3.2(b), data chunks will be out-of-place updated at the flash level, and the log chunks generated by the old and new data will be appended to a log device. Parity caching delays parity updates by caching all incoming requests in a buffer. In Figure 3.2(c), new parity P_0' is generated when $\{D_0, D_1\}$ are updated together. To further reduce parity traffic, elastic parity logging is introduced by encoding new data chunks to form log chunks, and the parity for a partial stripe is appended to a log device. As shown in Figure 3.2(d), three new requests are gathered to construct a new stripe Str_2.

3.2.3 Approximate Storage

Approximate storage exploits the error-tolerance of applications to reduce I/O latency and energy consumption. Approximate storage is enough for these applications which can tolerate a small portion of persistent errors. Attempts have been made in prior studies to utilize data inaccuracies to improve writing performance and relieve capacity constraints [109] [8] [26]. Sampson et al. [109] and Jinhua et al. [26] propose reducing the number of write steps on non-volatile memories to achieve high write performance and energy saving. Moreover, Jinhua et al. [26] propose extending the lifetime of SSD by reducing maximal threshold voltage, but a reduced voltage can only extend dozens of erase operations of a block. Azevedo et al. [8] extend the lifetime of PCM blocks by pairing spare blocks in disabled pages. Our proposed scheme is a distinct solution for lifetime optimization from recent work. First, we focus on the lifetime extension of SSD-based RAID systems which have parity updates. Second, we propose reusing faulty blocks to store error-relaxation data, and we can significantly extend the endurance of blocks.

3.3 FreeRAID

We present FreeRAID which leverages the interplay of RAID and SSD controllers to enhance the lifetime of SSD-based arrays. The basic rationale behind the FreeRAID system is that a new phase is added, i.e., the exploitable phase, in the life cycle of a block besides normal and bad phases. The exploitable blocks can be used to store error-relaxation data, even though errors will be persistent. To efficiently utilize exploitable blocks, FreeRAID integrates two key techniques:

- *Dual-space management* to efficiently allocate space for ordinary and error-relaxation data, and make them error-isolated.
- *Adaptive-FTL* to dynamically switch FTL schemes to optimize access performance in line with the age status of SSDs.

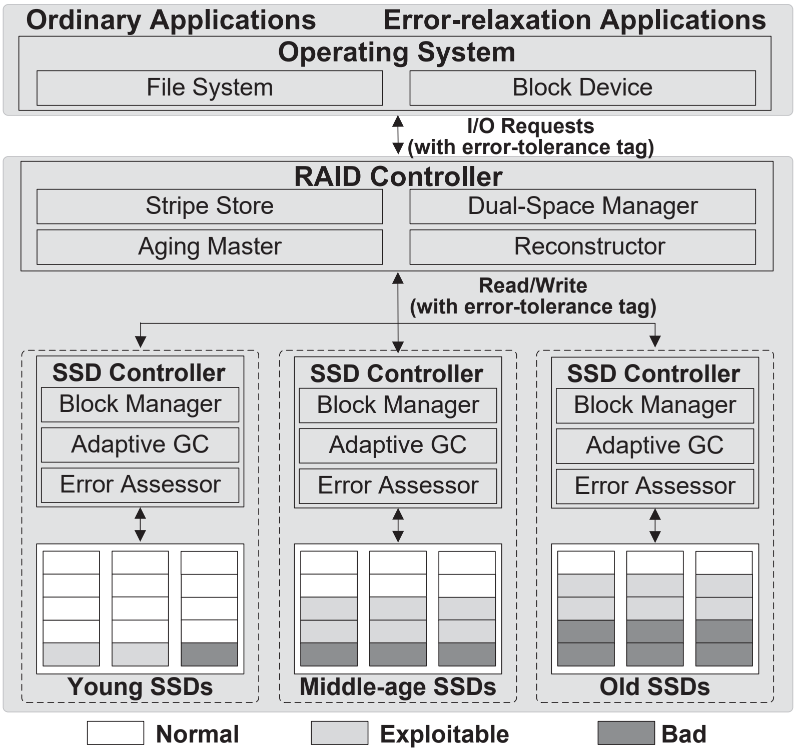


Figure 3.3: Overview of FreeRAID.

3.3.1 Overview

Figure 3.3 shows the general architecture of FreeRAID. FreeRAID tightly couples the components in both RAID and SSD controllers. In particular, FreeRAID implements the Dual-space management to efficiently allocate space for ordinary and error-relaxation data. Via the interplay between RAID and SSD controller, SSDs expose the information about their physical blocks to the RAID controller, and the RAID controller delivers data along with its error type to SSD controllers. Error-free data is gathered to construct ordinary stripes, while error-relaxation data is gathered to construct error-relaxation stripes. Dual-space management makes different kinds of stripes error-isolated for data reliability. Moreover, FreeRAID can collaborate with existing optimized RAID schemes, further extending the lifetime of SSDs by reducing write traffic. In addition, to improve storage efficiency and weaken the effect of garbage collection on performance, Adaptive-FTL is integrated into FreeRAID. FreeRAID divides the lifespan of an SSD into three stages, and Adaptive-FTL switches FTL strategies in accordance with the stage an SSD remains at.

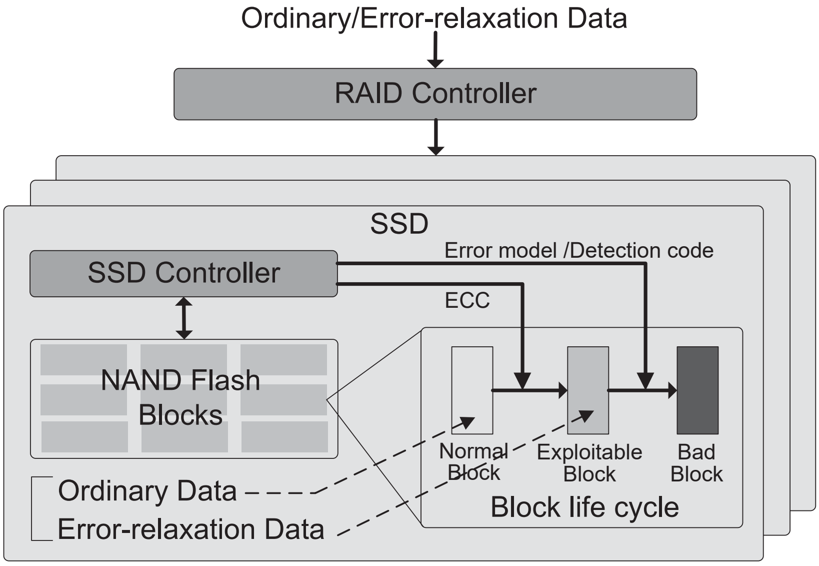


Figure 3.4: The transitions of blocks in a lifecycle and their error rate.

3.3.2 New Type: Exploitable Blocks

SSDs are assembled using NAND flash chips which contain a number of blocks. There are usually two types of blocks in an SSD: normal blocks and bad blocks. Normal blocks protected by ECC are intact, and they can degrade and wear out over time. Thus, some of normal blocks will be marked as bad blocks if they contain uncorrectable errors, and then they will be dismissed and replaced by reserved normal blocks. Instead of abandoning faulty blocks which contain a handful of uncorrectable errors, we reuse them as exploitable blocks to store error-relaxation data, as shown in Figure 3.4. Exploitable blocks still belong to the user addressable block area, and they are preferentially assigned for error-relaxation data. For reliability, we adopt an error model or employ a powerful error detection code to evaluate the validity of data on exploitable blocks. When the uncorrectable errors in an exploitable block exceeds a threshold, this exploitable block finally turns out to be a bad one.

All blocks follow a lifecycle from the normal phase to the bad phase. It is easy to identify a normal block only if a block is error-free under the protection of ECC. As for exploitable blocks and bad blocks, both have exhausted their error-correction resources, we use error rate to distinguish them. To satisfy different requirements of various error-relaxation applications, exploitable blocks are classified into three levels by error rate: low,

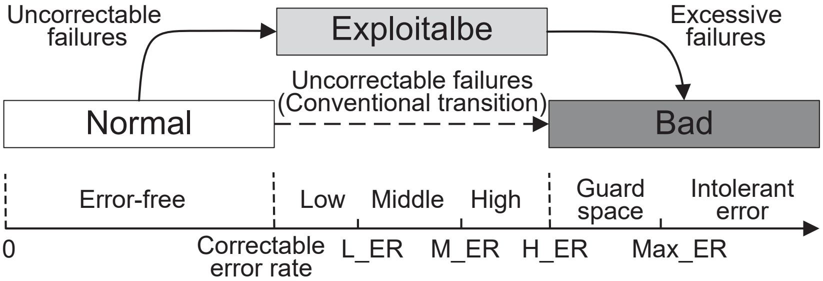


Figure 3.5: The transitions of blocks in a lifecycle and their error rate.

middle, and high, respectively. As shown in Figure 3.5, exploitable blocks in the low level are faulty blocks with an error rate that is no more than a pre-set error threshold Low_ER. The error rate of exploitable blocks in the middle level is larger than Low_ER but no more than the error threshold $Middle\_ER$. Similarly, the error rate of exploitable blocks in the high level stays between the threshold of $Middle\_ER$ and $High\_ER$. For reliability, we conservatively reserve a guard space between the thresholds $High\_ER$ and Max_ER. All error thresholds can be configured to accommodate the requirements of various applications.

The classification of exploitable blocks can be determined by a “read-after-write” scheme. The SSD controller writes the pre-defined data on an exploitable block, and then reads data from it, and finally erases it for use. By counting the number of corrupted bits, the exact error rate of this exploitable block can be attained. The SSD controller then inserts it into a free block list according to its error rate level. Note that it is too costly in terms of I/O bandwidth and computational resources to perform an error rate assessment at each erase operation. Thus, we argue that it is acceptable to assess the error rate of an exploitable block on a periodic basis, such as every 50 erase operations.

3.3.3 Dual-Space Management

Block transitions in SSDs occur, and application requirements vary. FreeRAID maintains two data pools for data allocation by obtaining the information about the physical blocks of SSDs: ordinary pool and error-relaxation pool. FreeRAID employs a stripe store and a dual-space manager to manage the data allocation in the RAID controller. The error-relaxation pool is specifically targeted at serving error-relaxation data, which is almost a free lunch compared with conventional allocation schemes.

• **Dual-Space Manager**.

The ordinary pool in FreeRAID is a scarce resource since it can serve all kinds of applications, while the error-relaxation pool is specifically targeted at serving error-relaxation applications. The dual-space manager can regulate the space of two data pools by monitoring the physical block information of SSDs, which requires close cooperation between RAID and SSD controllers. If the RAID controller needs more normal blocks for the ordinary pool, it will inform an SSD to release more normal blocks, and then the SSD controller will use reserved blocks to replace the faulty blocks. In addition, the dual-space manager can develop the age differential among SSDs to reduce the possibility of correlated failures. It adopts an uneven parity distribution to control the aging rates of SSDs. In the case of an SSD having a large number of faulty blocks, which makes an SSD array less reliable, the dual-space manager will assign more parity write to it for a quick disk replacement.

• **Stripe Store**.

An SSD-based RAID array is composed of $(N + 1)$ SSDs numbered from 0 to N. Multiple chunks form a stripe which is a collection of n data chunks and k parity chunks ($n + k <= N + 1$), and will then be distributed on different SSDs. Once a data chunk fails, the RAID controller requests other chunks in the same stripe to recover it. Taking data validity into consideration, ordinary data and error-relaxation data should be separated to construct stripes.

The stripe store collaborates with the Dual-space manager to implement error isolation for stripes. A stripe in FreeRAID is divided into multiple sub-stripes. Each sub-stripe contains data segments and parity segments. FreeRAID distinguishes between ordinary and error-relaxation requests by the error-tolerance knowledge of applications, and then groups requests into corresponding sub-stripes. The error-relaxation requests stored on exploitable blocks must be isolated with ordinary requests, unless they are treated as ordinary requests. We further illustrate data allocation and error isolation via an example, as shown in Figure

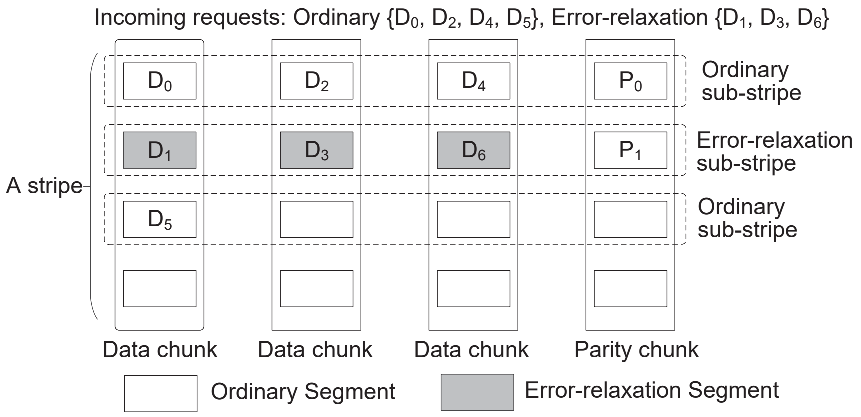


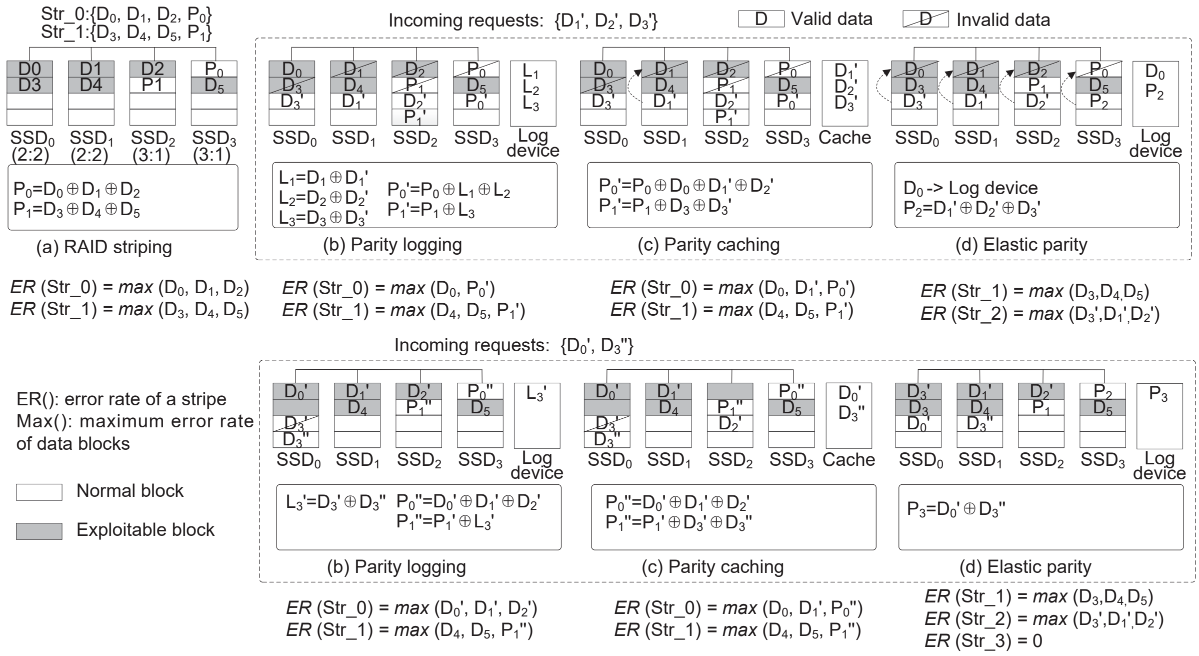
Figure 3.6: An example of data allocation and error isolation.

3.6. Ordinary requests $\{D_0, D_2, D_4\}$ are gathered to construct an error-free sub-stripe, while the requests $\{D_1, D_3, D_6\}$ are gathered to construct an error-relaxation sub-stripe, and its error rate is the maximum error rate among the data segments $\{D_1\}$, $\{D_3\}$ and $\{D_6\}$. Since the parity segment has heavy updates, thus the parity segment in each sub-stripe must be an ordinary segment. From this figure, we can see that the ordinary segment $\{D_5\}$ may become an acnode segment if future requests are error-relaxation ones. The stripe store can gather multiple acnode segments of cross-stripes to construct a sub-stripe. The stripe store maintains a segment mapping table to record the mapping information of sub-stripes, and records the type of each segment. A segment is the unit of allocation, and the size of a segment is aligned with multiple physical flash pages, thus the write performance can converge with sequential write performance in the SSD.

• **Combination with Existing Optimized RAID Schemes**.

FreeRAID and the existing optimized RAID schemes can coalesce. Ordinary stripes are processed in the same way as in existing optimized RAID schemes, while in case of error-relaxation stripes it is necessary to consider their error rates for the reliability of data when there are write updates.

Figure 3.7 illustrates how FreeRAID works in collaboration with existing optimized RAID schemes. Suppose that six error-relation requests are gathered to construct two stripes Str_0 and Str_1, and they are stored on exploitable blocks. Since the parity segments are



Str_0:{D0, D1, D2, P0}
Str_1:{D3, D4, D5, P1}
Incoming requests: {D1', D2', D3'}
D Valid data
D Invalid data
SSD0 (2:2)
SSD1 (2:2)
SSD2 (3:1)
SSD3 (3:1)
P0=D0⊕D1⊕D2
P1=D3⊕D4⊕D5
(a) RAID striping
L1=D1⊕D1'
L2=D2⊕D2'
L3=D3⊕D3'
P0'=P0⊕L1⊕L2
P1'=P1⊕L3
Log device
(b) Parity logging
P0'=P0⊕D0⊕D1'⊕D2'
P1'=P1⊕D3⊕D3'
Cache
(c) Parity caching
D0 -> Log device
P2=D1'⊕D2'⊕D3'
(d) Elastic parity
ER (Str_0) = max (D0, D1, D2)
ER (Str_1) = max (D3, D4, D5)
ER (Str_0) = max (D0, P0')
ER (Str_1) = max (D4, D5, P1')
ER (Str_0) = max (D0, D1', P0')
ER (Str_1) = max (D4, D5, P1')
ER (Str_1) = max (D3,D4,D5)
ER (Str_2) = max (D3',D1',D2')
Incoming requests: {D0', D3''}
ER(): error rate of a stripe
Max(): maximum error rate of data blocks
Normal block
Exploitable block
L3'=D3'⊕D3''
P0''=D0'⊕D1'⊕D2'
P1''=P1'⊕L3'
(b) Parity logging
P0''=D0'⊕D1'⊕D2'
P1''=P1'⊕D3'⊕D3''
(c) Parity caching
P3=D0'⊕D3''
(d) Elastic parity
ER (Str_0) = max (D0', D1', D2')
ER (Str_1) = max (D4, D5, P1'')
ER (Str_0) = max (D0, D1', P0'')
ER (Str_1) = max (D4, D5, P1'')
ER (Str_1) = max (D3,D4,D5)
ER (Str_2) = max (D3',D1',D2')
ER (Str_3) = 0

Figure 3.7: Combination with existing RAID schemes.

calculated by data segments in the buffer, and they are stored on normal blocks, thus they are intact at this moment. The error rates of Str_0 and Str_1 are the maximum error rate of their data segments, as shown in Figure 3.7 (a).

Assume that three update requests arrive sequentially, Figure 3.7 (b) illustrates how the parity logging scheme handles write updates. The delta $\{L_1\}$ and $\{L_2\}$ for updating $\{D_1\}$ and $\{D_2\}$ are logged on the log device. Since errors are persistent in $\{D_1\}$ and $\{D_2\}$ when reading them from exploitable blocks, the delta $\{L_1\}$ and $\{L_2\}$ are inaccurate after the computations. Thus, after updating parity $\{P_0'\}$ with $\{L_1\}$ and $\{L_2\}$, $\{P_0'\}$ is inaccurate even though it is stored on a normal block, and its error rate is not more than the sum of the error rates of $\{D_1\}$ and $\{D_2\}$. In order to restrict the error rate of a stripe, the new incoming data $\{D_1'\}$ and $\{D_2'\}$ are placed on normal blocks. Similarly, the update $\{D_3'\}$ is placed on a normal block. In the next stage, new requests $\{D_0'\}$ and $\{D_3''\}$ come to update two stripes. Since the parity $\{P_0'\}$ in Str_0 has been associated with two error-relaxation segments, it needs to be re-calculated. The parity is updated with the new $\{D_0'\}$, and $\{D_1'\}$ and $\{D_2'\}$ read from normal blocks, thus, all of the segments are error-free. To improve the block efficiency, the data segments in Str_0 can be moved on exploitable blocks. For new updated data {D$_3$"} in Str_1, since it updates the old data read from a normal block, a delta can be computed on the log device in this traditional way.

Figure 3.7 (c) illustrates an example of the parity caching scheme in FreeRAID. The incoming requests {D$_1$', D$_2$', D$_3$'} are cached. New parity {P$_0$'} is calculated by requests {D$_0$, D$_1$', D$_2$'} with the minimum number of error-relaxation segments. Since {P$_0$'} only connects to one error-relaxation segment, the request {D$_1$'} can be assigned to an exploitable block. The update with {D$_3$'} in Str_1 is processed the same way as in the parity logging scheme. In the next stage, considering the reliability of stripe Str_0, the parity {P$_0$"} is updated by {D$_0$', D$_1$', D$_2$'}. Since it still only connects to one error-relaxation segment, the {D$_0$'} can be stored on an exploitable block.

Figure 3.7 (d) gives an example of how elastic parity works in FreeRAID. The incoming requests on different SSDs form a log stripe. We can see that the incoming requests {D$_1$', D$_2$', D$_3$'} are gathered to construct a new stripe Str_2. After that, {D$_0$} can be treated as an acnode segment, and it can be moved to the log device. Since {D$_0$} has persistent errors, it turns to be the normal data for the termination of its errors propagation. Subsequent requests {D$_0$', D$_3$"} are gathered to construct a partial stripe Str_3 following the elastic parity scheme, and the new generated parity is appended to the log device.

3.3.4 Adaptive-FTL

SSDs in FreeRAID manage normal and exploitable blocks for allocation, and expose their block information to the RAID controller. The inequality of the numbers in the two types of blocks among SSDs results in a mismatch between I/O performance and reliability, such as the effect of garbage collection on read and write performance. FreeRAID coordinates Adaptive-FTL to optimize I/O performance. We elaborate on the functioning of Adaptive-FTL in this sub-section.

• **Block Manager**.

Figure 3.8 shows the block management in Adaptive-FTL. The addressable space consists of normal blocks and exploitable blocks. With regard to exploitable blocks for different

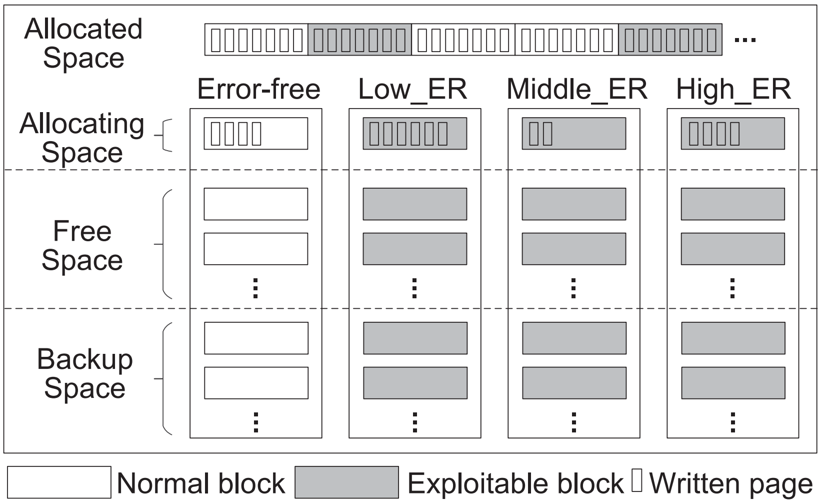


Figure 3.8: Block Allocation in FreeRAID.

requirements, the block manager divides them into three types according to their error rates. Besides that, according to the proportion of normal, exploitable and bad blocks on an SSD, we further divide the lifespan of an SSD into three stages: young, middle-aged, and old, as shown in Figure 3.3. When an SSD begins, we assume that all blocks are normal, and the initial stage is young. Normal blocks can degrade overtime, and exploitable blocks and bad blocks will appear. An SSD enters the middle-aged stage when there are a certain number of exploitable blocks. At this stage, error-relaxation data are preferably stored on exploitable blocks. At the old stage, an SSD contains more exploitable blocks and bad blocks than previous stages. FreeRAID tends to accelerate aging for reducing the possibility of correlated failures of SSDs. Proactively replacing superfluous exploitable blocks on an SSD can quickly consume the reserved good blocks, so as to accelerate its aging.

The block manager relies on a simplified page-level FTL, and severs requests in accordance with the error tolerance of requests. Read requests are directly served after translating a logical address to a physical address. For write requests, the block manger allocates free pages according to the stage that an SSD lives in. In youth, all writes are assigned to normal blocks. The block manager allocates free pages from normal blocks to serve all kinds of requests. At the middle-aged and old stages, the exploitable blocks are in service for error-relaxation requests. The block manager first identifies the error-tolerances of requests delivered from the RAID controller, and then selects an appropriate allocator of exploitable blocks to allocate pages. For example, a request with a high error-tolerance tag will be assigned to the High_ER allocator. If there are no free exploitable blocks for error-relaxation requests, the normal block allocator can assign a free page for requests. Moreover, the numbers of normal blocks and exploitable blocks hover with the demands of applications. If free normal blocks are not plentiful, the RAID controller can collaborate with the SSD controller to replace faulty blocks with reserved blocks from the backup space.

• **Adaptive Garbage Collection**.

Adaptive-FTL employs adaptive-GC to reduce the impact of the inequality of blocks among SSDs on performance. Adaptive-GC switches GC strategies in accordance with the stage that an SSD stays. At the stage of young age, adaptive-GC only reclaims normal blocks, and the GC process is triggered when free normal blocks are below a watermark called GC_normal. At the middle-aged stage, adaptive-GC executes different collection strategies for normal blocks and exploitable blocks, respectively. Normal blocks are a scarce resource at this stage, so adaptive-GC uses a radical way to reclaim them. A higher watermark for normal blocks called $GC\_radical$ is set. For exploitable blocks, copying valid data to other places also carries persistent errors, so adaptive-GC copies the living data from a Low_ER or Middle_ER victim block to an available Low_ER exploitable block so that the maximum error rate of the copied data is no more than {L_ER+M_ER}. The targeted block will then turn out to be a High_ER exploitable block, and the SSD controller will inform the RAID controller about the change in reliability issue for future updates. To reclaim H_ER exploitable blocks, valid pages will be copied to a normal block to terminate their error propagation. Correspondingly, the watermarks GC_low, GC_middle and GC_high are set for reclaiming different levels of exploitable blocks, and they are much lower than $GC\_radical$ to reduce the impact of GC on the quality of data. At the old stage, a victim High_ER exploitable blocks will be marked as bad, and then will be replaced by a reserved block to accelerate aging of an SSD.

Adaptive-FTL redesigns block management, address mapping, and garbage collection in SSD's controller to collaborate the RAID controller for exploring the special feature of data. In addition, Adaptive-FTL can be compatible with existing wear leveling schemes, such as static and dynamic wear leveling.

• **Error Rate Assessor**.

Normal blocks rely on ECC to determine whether the validity of intact data has been accomplished, while it is still challenging for exploitable blocks, which have exhausted error-correction resources, to judge the validity of data for error-relaxation applications. Many sources can cause flash memory errors, such as P/E cycling and retention time [37] [135] [124]. Flash memory P/E cycling incurs the shift and fluctuation of threshold voltage in a memory cell. The retention error is caused by charge leakage over time after a flash memory is programmed, and it has a positive correlation with the number of P/E cycles. Thus, by considering two dominant sources of errors, we propose two schemes to implement early error detection of exploitable blocks.

Error-Assess Model. We generalize an exponential growth equation derived from [135] [124] to predict the errors caused by P/E cycling and retention time on an exploitable block:

$$\begin{aligned} RBER(n,t) &= RBER_{cycles}(n) + RBER_{retention}(n,t) \\ &= Ae^{Bn} + C \\ &+ (a_0 + a_1 n + a_2 n^2 + a_3 n^3)t \\ &+ (b_0 + b_1 n + b_2 n^2 + b_3 n^3) \end{aligned}$$

where n is the number of P/E cycles, t is the retention time whose unit is a day, and A, B, C are correlation coefficients derived from the parameters for 3x nm MLC in [124]. $RBER_{cycling}$ denotes the bit error rate caused by P/E cycling, and it is dependent on the P/E cycles n. $RBER_{retention}$ denotes the bit error rate caused by retention time, and it is related to the both P/E cycles n and retention time t. In the model of retention errors, we use a polynomial curve fitting with a least-square calculation to generalize the correlation between the retention errors and P/E cycles from [135]. a$_i$ and b$_i$ ($0 \leq i \leq 3$) are correlation coefficients, and they are calculated as follows: -0.018, 0.016, -0.0018, 0.0001, 0.1752, -0.04, 0.0323, -0.001. By validating the results with the motivation experiment shown in Figure 3.1, we find that our model can accurately analyze the error rate of exploitable blocks.

Error Detection Code. To guarantee reinforced reliability for exploitable blocks, more powerful and stronger error detection codes can be employed in FreeRAID to obtain the exact error rates. For example, EG-LDPC [102], a class of LDPC codes constructed deterministically using the points and lines of Euclidean geometry over a Galois field, is a good candidate for detecting high errors with few decoding cycles. In a 255-bit EG-LDPC codeword that contains 175 information bits, there is only a small possibility that the errors affecting up to 12 bits (6.8%) cannot be detected. However, such codes with excellent detection capability incur a fantastic amount of storage overhead, for example, 80 parity bits are needed in a 255-bit codeword. We believe that a proper error detection code for FreeRAID exists, which satisfies the requirements of detection capability and has low storage overhead. The design of efficient error detection codes is beyond the scope of this chapter; thus, we use the error-assess model to judge errors on exploitable blocks.

FreeRAID employs the error rate assessor to guarantee the reliability of exploitable blocks. The error rate assessor serves two main functions. First, it judges the quality of error-relaxation data by using the error-assess model. For a read request on an exploitable block, the error rate assessor first calculates the error rate of the exploitable block with its best knowledge, and then compares the calculated error rate with the error-tolerance of the request. The data will be returned to applications if the quality is qualified. Otherwise, a block failure will be reported to the RAID controller which will use redundancy to recover the failure. Second, the error rate assessor can proactively predict the validity of data. Risky data will be reported to the RAID controller even if the data has not been accessed. Proactive recovery can reduce the number of degraded reads, and consequently, improve the responsiveness of applications.

3.4 Evaluation

In this section, we present the experimental methodology and the experimental results of our proposed FreeRAID.

3.4.1 Experiment Setup

We evaluate FreeRAID via a trace-driven simulator, and compare it with conventional RAID and FTL schemes. FreeRAID is modeled by heavily modified Flashsim [13] to implement RAID schemes and the components in both the RAID controller and SSD controller. We implement a RAID-5 scheme with five devices on the simulator, and add a log device for the existing optimized RAID schemes. We simulate each SSD with 16GB of raw capacity, in which 15% of flash blocks are over-provisioning space. Each block has 128 pages, and each page is of 4KB. A stripe contains four sub-stripes, and each chunk in a sub-stripe is 64KB.

We use the erase number as the lifetime of normal blocks, and use the error rate to identify exploitable blocks and bad blocks in the simulator. The error rate thresholds Low_ER, $Middle\_ER$, $High\_ER$ for three kinds of exploitable blocks are set to 0.0004, 0.0006 and 0.0010, respectively. These thresholds are set based on persistent-storage benchmarks presented in [109], by which the endurance of a flash block can be significantly extended. These thresholds reflecting the potentials of lifetime extensions can be configured to accommodate the requirements of applications. To develop reasonable error rate fluctuations, we initialize the error rates of exploitable blocks and set the increase of error rates after each erase operation following the Gaussian distributions with different parameters. The mean of initialized error rate is 0.00009 with 0.0025 variance; and the means of increase of error rates for three kinds of exploitable blocks are $(23 * 10^{-9})$ with 0.005, $(33 * 10^{-9})$ with 0.01, and $(50 * 10^{-9})$ with 0.015.

We consider four real-world I/O traces [34] [131], as shown in Table 3.1. Financial1 and Websearch capture the workload from OLTP applications running at two large financial institutions. The original Websearch is a read dominant workload. We invert all read requests

Table 3.1: Workload characteristics

Workloads	# of requests	Write Ratio	Average request size
Financial1	5334987	76.8%	3466B
In-Web	9896966	100.0%	15589B
Webmail	7795815	81.8%	4096B
Online	5700499	73.8%	4096B

to writes to generate a new trace called In-Web, choosing it because of its ample accesses and large address space. Webmail and Online traces are collected from a department mail server and a course management system, respectively. Note that there is no error-tolerance tag in these traces; we chose them because they are popular workloads and they have a large number of writes. In order to evaluate the performance of FreeRAID, we plug in error flags to random entries of these original traces. The mixed data containing both ordinary and error-relaxation data can reflect the overarching features of the applications.

3.4.2 Evaluation Results

• Lifetime Extension.

We use the number of writes before the first device failure happens to evaluate the lifetime of an SSD-based RAID array. Although each trace spans a very large address space, only a small proportion of address ranges is accessed. Therefore, in order to trigger more erase operations, we compact the whole addressable space of RAID to 4GB, and allocate extra an 5% of blocks as reserved space. The first failure happens when one of the SSDs has exhausted all reserved blocks. We generate two sets of age groups of SSDs to simulate different scenarios in a data center. The first array is an age-balanced group in which all the SSDs have the same wear-out rate. The remaining erase number of each normal block is set to 5. The other array has differential ages derived from [9]. One of the SSDs in the different-aged array is younger than the others, and it will handle more parity writes. All blocks in the

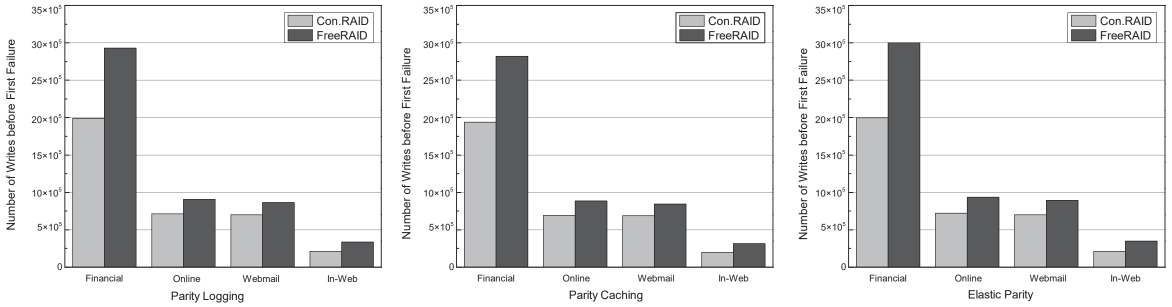


Figure 3.9: Total writes of FreeRAID and a conventional RAID with SSDs in balanced ages.

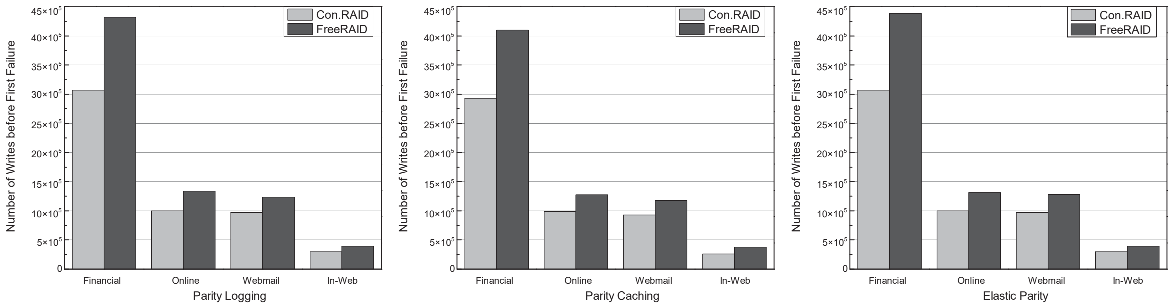


Figure 3.10: Total writes of FreeRAID and a conventional RAID with SSDs in differential ages.

younger SSD have 15 remaining erase number, while the remaining erase number of blocks in the other SSDs is set to 5. We compare FreeRAID with a conventional RAID (Con.RAID) in which blocks are marked as bad if their remaining erase number is zero. In FreeRAID, bad blocks are generated after the transitions of exploitable blocks. We set the maximum ratio of exploitable blocks in an SSD at 30%, meaning that the superfluous exploitable blocks will be marked as bad blocks, and then will be replaced. In this experiment on FreeRAID, we feed it workloads with half error-tolerated requests and half normal requests.

Figure 3.9 shows the total number of writes before the first failure happens in FreeRAID and a conventional RAID under three existing optimized schemes. When all SSDs share the same wear-out rate, FreeRAID achieves a lifetime extension of up to 65% across different traces. Compared with the conventional RAID, 65% and 50.7% extra writes can be realized in FreeRAID in In-Web and Financial workloads, respectively. FreeRAID gains 29.4% and 27.8% extra writes in Online and Webmail, respectively. The main reason for this increase is that FreeRAID distributes most of the error-relaxation data on exploitable blocks, which

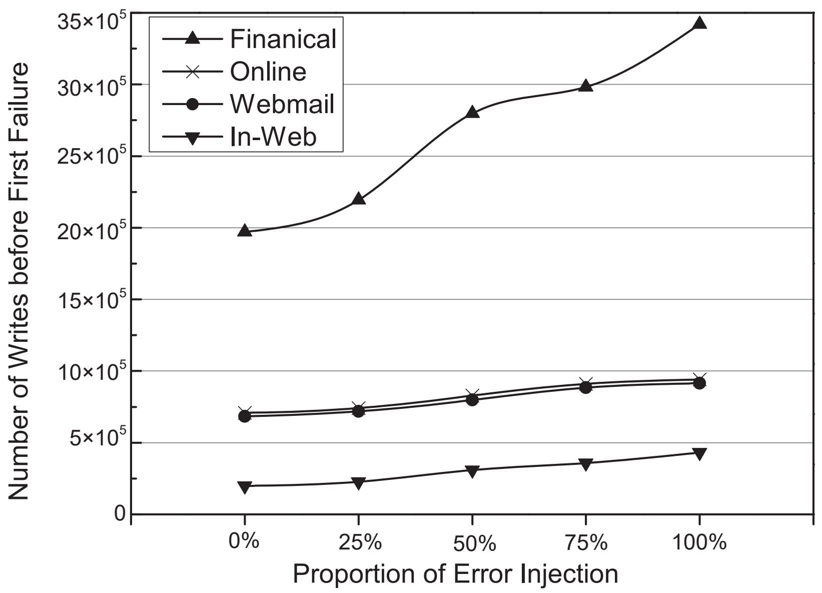


Figure 3.11: Lifetime extensions under different ratios of error-relaxation workloads.

significantly reduces the worn-out rate of normal blocks. Among the existing optimized schemes, the elastic parity scheme can respond with more writes since it reduces more write traffic and GC operations on SSDs than other schemes.

Figure 3.10 shows the lifetime extension of SSDs with the differential ages. FreeRAID increases the total writes by up to 46.7% compared to the conventional RAID scheme in three existing optimized schemes. This implies that FreeRAID can significantly improve the lifetime of an SSD-based RAID array beyond parity redistribution.

Figure 3.11 shows the total number of writes before the first failure happens at various error injection levels in FreeRAID without any optimized RAID schemes. We can see that FreeRAID significantly extends the lifetime of an array under four workloads containing more ratios of error-relaxation data. In the In-Web, FreeRAID extends the lifetime by up to 2.17×. Averagely, FreeRAID achieves 1.31× and 1.63× lifetime extensions with 50% and 100% ratios compared to the traditional RAID (all data is error-free), thereby demonstrating the effectiveness of our proposed techniques.

• **Average Response Time**.

To evaluate the effectiveness of Adaptive-FTL in FreeRAID, we compare it with the state-

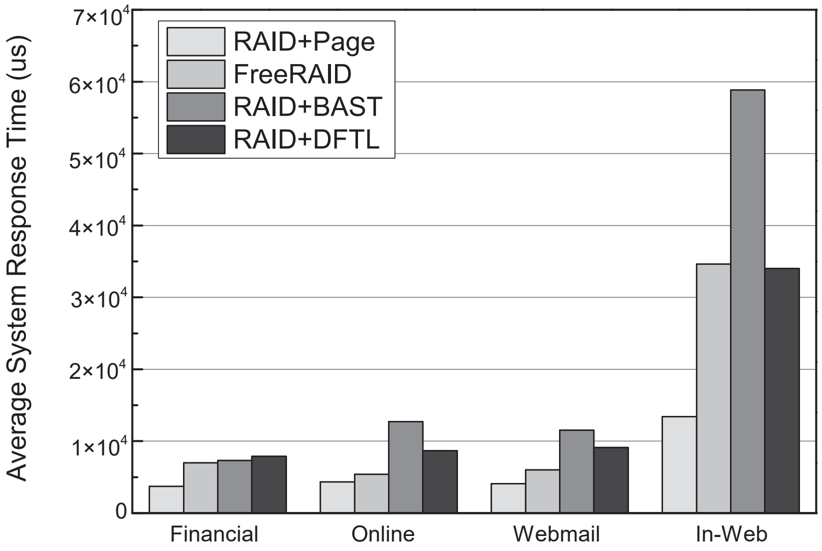


Figure 3.12: Comparisons of average system response time.

of-the-art FTLs in a standard RAID-5 scheme. DFTL is an on-demand page-level FTL with one-level cache, both page-level mapping table and data blocks are stored in the flash memory [40]. Block associative sector translation (BAST) scheme allocates a log block for only one data block [58]. Page FTL is an ideal page-level FTL without any constraints. The major performance metric to be evaluated is the average request response time. Since DFTL, BAST, and Page FTL only work on normal blocks, we feed the original workloads to them. For FreeRAID, we use the error-injected workloads to evaluate the effectiveness of Adaptive-FTL. We set the initial proportion of exploitable blocks of five SSDs to 0, 10%, 15%, 25%, and 30% respectively. Figure 3.12 presents the experimental results of average write response time with different FTLs. We can see that FreeRAID with Adaptive-FTL has an obviously lower response time than the conventional RAID with BAST and DFTL schemes, with a reduction by up to 57.6%. The main reason for this result is that Adaptive-FTL in FreeRAID is based on a simplified page-level FTL which has no extra log reads and writes than BAST and DFTL. However, Adaptive-FTL brings some overhead compared with the Page FTL, the overhead mainly includes extra writes for moving error-relaxation data from normal blocks to exploitable blocks, and the error-assess process for exploitable blocks.

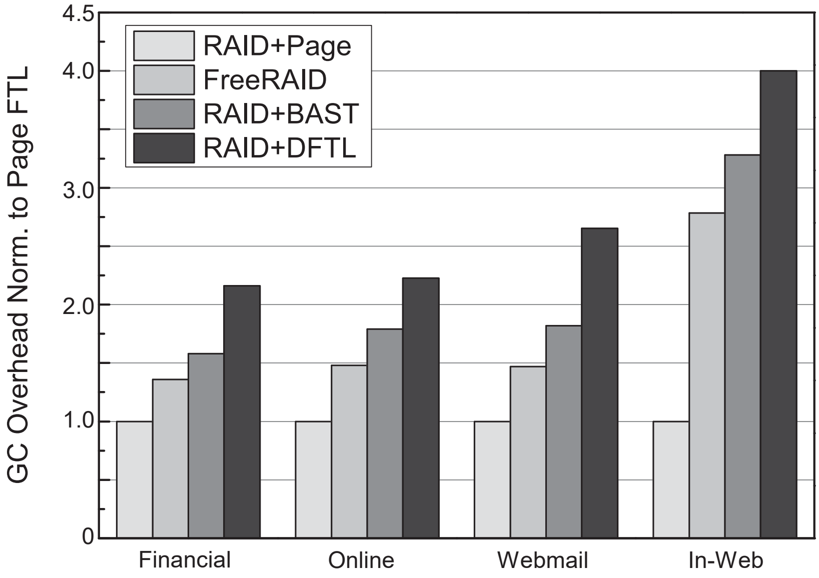


Figure 3.13: Comparisons of GC overhead.

• **GC Overhead**.

We measure the average number of GC requests across all SSDs in terms of the GC overhead. The victim blocks are picked by a default greedy algorithm on FTLs. We distribute the SSDs into different stages in FreeRAID, and initialize a portion of invalidated blocks so as to quickly trigger garbage collection. A GC trigger happens when the number of normal blocks drops below 25% of normal space in the young stage, and 35% in the middle and old stages. Garbage collection for three kinds of exploitable blocks is triggered when the numbers of free blocks drop below 15%, 10%, and 5% of the total blocks. Since DFTL, BAST, and Page FTL can only reclaim normal blocks, while Adaptive-FTL can work on both normal and exploitable blocks, thus we feed a portion of the workload to them to lower the effect of different parameters and differences in blocks. The GC overhead normalized to the Page FTL is shown in Figure 3.13. We can see that FreeRAID with Adaptive-FTL significantly reduces the number of GC requests over that in DFTL and BAST. It reduces 1.8× of GC requests compared with DFTL in Webmail workload. The reason for this result is that Adaptive-FTL efficiently performs garbage collection with different schemes in accordance with the stage an SSD stays, and it is a lightweight page-level implementation when compared with DFTL and BAST.

• **Overhead Discussions**.

Our proposed scheme incurs performance overhead which mainly comes from three parts: adaptive-GC, error-relaxation data migration, and mapping table lookups. First, adaptive-GC processes for normal and exploitable blocks are triggered by different watermarks in accordance with the stage an SSD stays at. Adaptive-GC for normal blocks in middle age is more radical than at a young age. The higher watermark generates more GC operations, resulting in a performance gap with the ideal Page FTL, as shown in Figure 3.13. Second, for normal block efficiency, migrating error-relaxation data from a normal block to an exploitable block incurs extra writes and reads. We can schedule the migration tasks to accommodate a trade-off between space efficiency and I/O performance, such as blocking the migrations. Third, the mapping table lookups and the flag checks in both RAID and SSD controllers incur minor performance overhead.

3.5 Other Related Work

SSD-based RAID is a reliable storage system with the redundancy-based protection technique. Several existing studies have focused on improving the lifetime of a RAID array. Jimenez et al. [52] relieve the weakest pages to implement block lifetime extension. They operate on healthy blocks in a proactive way, but this incurs a loss of capacity. Moon et al. [81] analyze the relationship between the parity protection and the lifetime of SSD arrays, and they find that the write amplification is a major factor in the lifetime of RAID. Yongseok et al. [87] use a log-structured cache to eliminate read-modify-write operations, they propose the use of destaging, instead of garbage collection to enhance the lifetime of SSDs. Both of these existing work focus on reducing write traffic, while we explore the special features of data to extend the lifetime of SSD-based RAID.

Exploiting the error resilience in modern applications has led to multi-aspect improvements in storage. Sampson et al. [109] and Azevedo et al. [8] explore data inaccuracies to optimize the performance and storage efficiency of PCM, and Jinhua et al. [26] optimize them on 3D flash-based SSDs. For specific applications, Jevdjic et al. [51] compute bit-level reliability requirements for encoded video by tracking coding dependencies, and implements different levels of error correction for streams' reliability needs, and Guo et al. [38] propose selective error correction technique to implement high-density image storage.

3.6 Summary

In this chapter, we proposed FreeRAID, which leverages the interplay of RAID and SSD to extend the lifetime of SSDs arrays. The exploitable blocks are explored to store error-relaxation data in FreeRAID, even though uncorrectable errors will be persistent. FreeRAID employs Dual-space management and Adaptive-FTL to improve block efficiency and maintain data reliability and I/O performance. Our experiments show that we can significantly increase the lifetime of SSD-based arrays compared to conventional RAID solutions and FTLs.

CHAPTER 4

REBIRTH-FTL: LIFETIME OPTIMIZATION VIA APPROXIMATE STORAGE FOR NAND FLASH

4.1 Introduction

NAND flash memory has been widely adopted as a storage medium in embedded storage devices (e.g. smartphones, tablets, etc.) and enterprise storage systems (e.g. SSDs, flash array, etc.). With feature-size reductions and multi-level cell technology, the density of flash memory is dramatically improved. However, the lifetime of memory cells inevitably deteriorates. For instance, with 2x-nm technology, MLC (Multi-Level Cell) and TLC (Triple-Level Cell) can only endure 3,000 and 1,000 Program/Erase (P/E) cycles, respectively [76]. This life-deterioration trend will continue as we move to smaller feature size (e.g. 7-nm) and QL-C (Quad-Level Cell) technology. Therefore, the declining lifetime of NAND flash memory exhibits challenges for memory management.

Today, we are storing more and more approximate data that can be more error-tolerant than regular data which needs bit-by-bit precise, such as image and video. Approximate data can accept a small number of bit flips. For instance, edge detection on an image can stand up to 16.9% error rate [116]. The error tolerance shows the potential to improve the lifetime of flash memory. Figure 3.1 in Section 3.1 shows the relationship between the P/E cycles and the corresponding bit error rates on an MLC NAND flash chip [110]. Based on this observation, in this chapter we further explore the error-tolerance of data to optimize the lifetime of flash memory in embedded storage system.

Recent work have demonstrated that approximate storage has led to multi-aspect improvements in solid state memories. Sampson et al. [109] and Jinhua et al. [26] propose reducing the number of write steps on non-volatile memories, so as to achieve higher write performance and energy saving. Azevedo et al. [8] explore the error-tolerance of approximate data to optimize storage efficiency in PCM. Guo et al. [38] propose the selective error correction codes for high-density image storage, and Jevdjic et al. [51] implement different levels of error corrections for encoding video. Distinguished from these prior work, our work focuses on flash memory management at the flash translation layer without the hardware changes (such as the incremental step pulse programming strategy to reduce writes [109] [26]), or the dedicated ECC techniques (for encoding images with the progressive transform codec [38] and videos with H. 264 [51]).

In reconsidering the connection between the FTL and approximate storage for lifetime extension, it is still challenging to apply approximate storage into embedded storage system without the involvement of hardware. First, a page-level FTL needs to manage two separated spaces, i.e., normal space and approximate space, and it needs to consider the error rate at block-level. The approximate space which consists of the faulty blocks can be used to store error-relaxation data, and it should be separated with normal space which serves error-free data. Second, two spaces vary dynamically since some flash blocks will turn into faulty ones with the increase of P/E cycles, so the garbage collection (GC) and wear leveling (WL) in the FTL should be orchestrated to maintain good I/O performance. Moreover, coping valid pages from a victim faulty block to another faulty block may double the error rate of approximate data in both GC and WL schemes, thus the GC and WL need to be carefully examined for the reliability of approximate data. Third, there exists an approximation-awareness barrier between applications and flash devices drivers in embedded operating systems. The OS is layered into several distinct subsystems, and the separations between them make it challenging to deliver the special features of data through the whole OS.

In this chapter, we present Rebirth-FTL, which reuses faulty blocks to improve the lifetime of flash memories. We propose a new kind of blocks, approximate blocks that contain a handful of uncorrectable bit errors to store error-relaxation data. Rebirth-FTL manages two separated spaces to serve applications. Normal space serves all kinds of data and guarantees the integrity of data, while approximate data is specifically targeted at serving error-relaxation requests. Three key components as pure software management are redesigned to accommodate two separated spaces in Rebirth-FTL. The approximation-aware address mapping scheme manages both normal and approximate spaces, and allocates blocks for the corresponding data. To maintain I/O performance and validity of data, we propose coordinated garbage collection which collaboratively reclaims free blocks with data migration between two spaces. Differential wear leveling, with the different strategies, is proposed to spread the wearing of two spaces evenly. We also propose a lifetime model for lifetime analysis to evaluate the benefits of reusing faulty blocks. In addition, to eliminate the approximation-awareness obstacle, we demonstrate how to pass approximate information from applications to flash devices in Linux.

We have prototyped Rebirth-FTL on an embedded development board and a simulator, and we have conducted a series of experiments to evaluate its performance. Evaluations across a wide variety of workloads show that Rebirth-FTL significantly outperforms conventional FTLs in lifetime (up to 3.46× improvement). To summarize, this chapter makes the following contributions:

- We propose Rebirth-FTL, which reuses faulty blocks for lifetime optimization, with three key components in FTL that are redesigned.
- We present how to pass approximate information from userland to kernel space with minimum OS modification and overhead.
- We analyze the lifetime benefits of Rebirth-FTL in a lifetime model.
- We demonstrate the effectiveness of our technique by conducting a set of experiments.

The rest of this chapter is organized as follows. The next section gives basic background to this study. Section 4.3 describes the overview of our proposed Rebirth-FTL, as well as its three key components. In Section 4.4, we briefly introduce how the approximate information of data is delivered through the whole Linux OS in a top-down way. An analytical lifetime model is presented in Section 4.5, and we apply it to explore the optimization opportunity. Section 4.6 evaluates the performance of Rebirth-FTL in several key metrics. Finally, Section 4.7 concludes this chapter.

4.2 Background

This section briefly presents the background on NAND flash memory, flash translation layer, and approximate storage.

4.2.1 NAND Flash Memory

A typical NAND flash memory is partitioned by blocks, and a block is further divided into multiple pages. A page is the basic unit of read and write operations, while a block is the basic unit of erase operations. Read, write and erase are three basic operations in NAND flash memory. The write operation writes data to an available page, and the data can be read from it. Due to the out-place update, the page can be re-written after an erase operation which is performed on a block basis. ECC is employed to accomplish the required reliability of blocks while reading and writing, and normally it can perform single or multiple bit-error corrections. A flash block can endure a finite number of P/E cycles. An increasing number of P/E cycles incurs the shift and fluctuation of the threshold voltage of a memory cell, and then uncorrectable errors occur. Any uncorrectable failures with even one bit error will exhaust an entire normal block which is then marked as a bad block. Bad blocks will be dismissed and replaced by reserved good ones. Typically, a single-level flash memory cell can tolerate 10,000 P/E cycles [16]. With multi-level cell technology, the density of flash memory is dramatically improved, while the endurance reduces. The MLC and TLC can be reprogrammed for 3,000 P/E cycles and 1,000 P/E cycles, respectively. The storage systems normally have strict requirements on reliability. This life-deterioration trend will continue to challenge current systems as we move to smaller feature size and QLC technology.

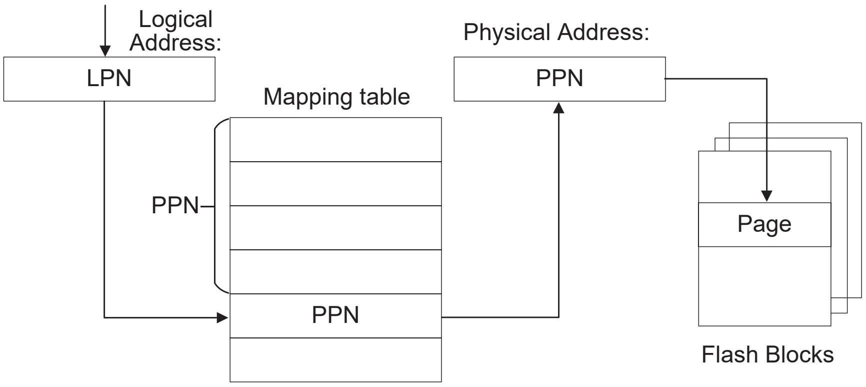


Figure 4.1: A typical page-level FTL scheme.

4.2.2 Flash Translation Layer

The FTL layer emulates a flash memory as a block device, so a file system can access flash memories transparently. FTL serves major management functions: for address translation, an address mapping table is utilized to translate addresses between logical and physical addresses; for garbage collection, the obsolete spaces are reclaimed by erasing blocks, the process involves reading, rewriting and erasing; for wear leveling, a technique to prolong the lifetime by distributing erasures and writes evenly across memories. Various FTL algorithms have been proposed, and they can be grouped into three types: page-level, block-level, and hybrid mapping schemes. The page-level mapping scheme allocates any physical page on the flash memory to a logical page, as shown in Figure 4.1. The disadvantage is that the mapping table consumes large amounts of space since it maintains an entry for each logic page. The block-level mapping scheme first consults the table to find a physical block, then goes to the corresponding page according to the offset. Although it reduces the size of mapping table, it may waste storage space due to its inflexibility. A trade-off between the page-level and block-level mapping schemes is a hybrid mapping scheme, which uses block-level mapping for data blocks, and page-level mapping for log blocks. However, performance reduction is unavoidable since it increases the algorithmic complexity of address mapping schemes. In this chapter, we focus on the page-level mapping scheme.

4.2.3 Approximate Storage

Approximate storage exploits the error tolerance of applications to reduce I/O latency and energy consumption. A small number of bit flips is acceptable for some applications such as image processing, and the overall quality of service is satisfied. For example, a high fidelity image read from the approximate storage substrates still can be applied for face recognition, which leverages the fact that human senses can tolerate imperfections in output. Even with a certain degree of quality reduction in an image, users can still recognize the character. Recent research has proposed applying approximate storage for images and videos [51] [39] [90] [38]; the principle is to employ unequal error protection techniques to implement lower computational complexity and higher storage efficiency.

4.3 Rebirth-FTL

In this section, we present Rebirth-FTL, a lifetime optimization scheme in the flash translation layer, which gives a second birth of a faulty block to store data. We first give an overview of our scheme, and then we present its three key components.

4.3.1 Overview

Figure 4.2 shows the general architecture of our NAND flash memory storage system. Rebirth-FTL, a pure software management in the flash translation layer, resides above the MTD subsystem, and manages two separated spaces: normal space and approximate space. Rebirth-FTL is aware of the approximate information from applications, and it can efficiently allocate the corresponding blocks for requests in accordance with their $approx\_flags$. Error-relaxation requests with valid $approx\_flags$ can tolerate a small amount of bit flips, thus Rebirth-FTL preferentially allocates approximate space to serve them. In order to achieve high block efficiency, Rebirth-FTL integrates approximation-aware address mapping, coordinated garbage collection, and differential wear leveling.

Normal blocks protected by ECC are error-free, and some will be developed to faulty

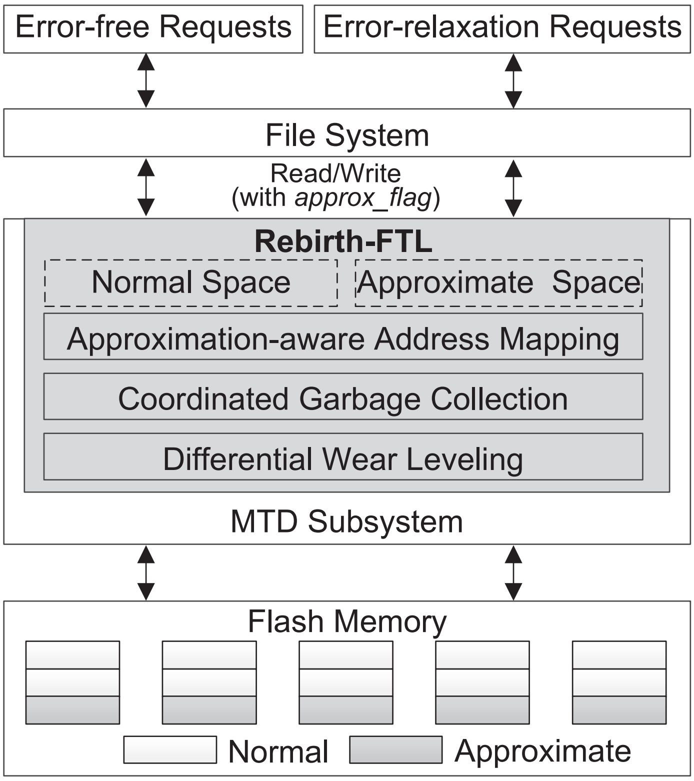


Figure 4.2: NAND flash system architecture with Rebirth-FTL.

ones if they have been experienced excessive P/E cycles. We reuse some faulty blocks (denoted as approximate blocks) to store error-relaxation data, which seems to promise a rebirth for these faulty blocks. Approximate blocks will turn out to be bad ones if they have more bit errors which exceed a pre-set threshold. The error rate of a faulty block can be determined by a "read-after-write" scheme [46]; thus, we use the error rate to distinguish approximate blocks and bad blocks.

4.3.2 Approximation-aware Address Mapping

Rebirth-FTL adopts an approximation-aware address mapping approach to allocate normal blocks and approximate blocks for requests. It allocates space at the page granularity, but it ensures that the pages for requests with same $approx\_flags$ can be grouped physically at the block granularity. Figure 4.3 shows the block management in Rebirth-FTL. Normal blocks can serve both error-free and error-relaxation requests; thus, two allocator heads are

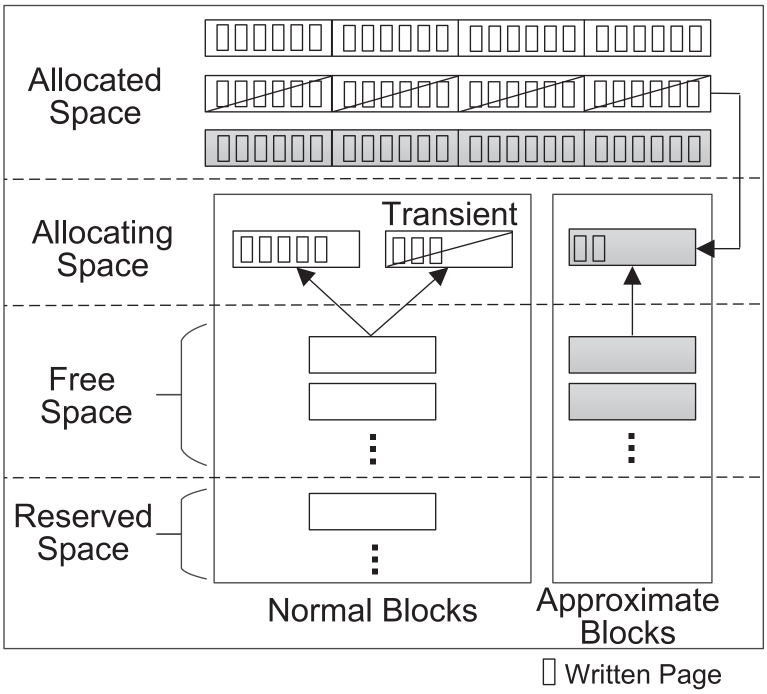
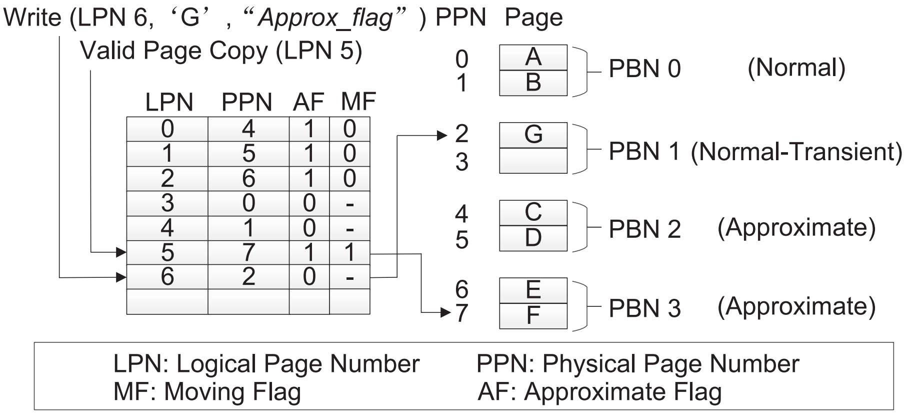


Figure 4.3: Block management in Rebirth-FTL.

maintained for normal blocks. If there is no more approximate block for error-relaxation requests, the data will be stored on the normal block (normal-transient blocks). Later, data on the allocated normal-transient blocks can be migrated to available approximate blocks for block efficiency. The approximate blocks specifically target at serving error-relaxation requests, and an allocator head is maintained to allocate free pages from an approximate block.

The approximation-aware address mapping approach relies on a page-level mapping table. The logical page of a request sent to the flash memory is mapped to a physical page in accordance with its flags. This is well illustrated with the example shown in Figure 4.4. Suppose there are two normal and two approximate blocks, and each block contains two pages. The error-free data 'A' and 'B' are assigned to the physical page 0 and 1 in the first normal block, and their approximate flags are set to 0. Accordingly, the error-relaxation data 'C', 'D', and 'E' are assigned to the pages on approximate blocks, and their approximate flags are set to 1. Moreover, we add a moving flag in the table to restrict the error rate propagation for the pages in approximate blocks, since the error rate of data migrated from



LPN	PPN	AF	MF
0	4	1	0
1	5	1	0
2	6	1	0
3	0	0	-
4	1	0	-
5	7	1	1
6	2	0	-

LPN: Logical Page Number PPN: Physical Page Number
MF: Moving Flag AF: Approximate Flag

Figure 4.4: Error-separated address mapping in Rebirth-FTL.

an approximate block to another block may increase. For example, suppose ‘F’ is being copied to physical page 7 from a victim approximate block due to a garbage collection, and its moving flag is set to 1. For the next garbage collection, ‘F’ will be delivered to a normal block to terminate the error rate propagation by checking its moving flag. When error-relaxation request ‘G’ comes, the physical page 2 in a normal-transient block will be assigned to it, and its approximate flag is set to 0 since there is no available page slot in other approximate blocks. Later the request ‘G’ can be migrated to an available approximate block.

4.3.3 Coordinated Garbage Collection

Coordinated garbage collection collaboratively reclaims normal and approximate blocks in different manners they see fit, as shown in Algorithm 4.3.1. First, the coordinated garbage collection processes for two kinds of blocks are triggered by different watermarks. Specifically, the coordinated GC is triggered when free blocks below the watermark GC_normal and GC_approx, respectively. Second, in contrast to the conventional FTLs in the selection scheme of a victim normal block, coordinated GC preferentially selects an allocated normal-transient block as a victim block (lines 3-6). If a normal-transient block is selected as the victim, the migration of valid pages from the victim to an available approximate block is executed, and the approximate flag of these pages will be updated in the mapping table.

Algorithm 4.3.1 Coordinated Garbage Collection.

Input: Invalid physical normal/approximate block $BKinvalid$.
Output: Perform the garbage collection operation.

Normal blocks:

```
if $BKinvalid$ exists then
    Perform GC on $BKinvalid$
else if Allocated normal-transient blocks exist
        && Available approximate blocks exist then
    Select a victim block $BKtransient$ containing the least valid pages.
    Migrate valid pages to an approximate block
    Perform GC on $BKtransient$ end
else if Allocated normal blocks exist then
    Select a victim block $BKnormal$ containing the least valid pages.
    Perform GC on $BKnormal$ end
end if
```

Approx blocks:

```
if $BKinvalid$ exists then
    Perform GC on $BKinvalid$
else if Allocated approximate blocks exist then
  Select a victim approximate block $BKvictim$
    for each valid page $PGvalid$ in $BKvictim$ do
      if $PGvalid$ has been moved then
        Copy $PGvalid$ to a free page in a normal block
      else
        Copy $PGvalid$ to a free page in an approximate block
      end if
    end for
  Perform GC on $BKvictim$ end
end if
```

Third, when the garbage collection for approximate blocks is triggered, reclaiming a victim approximate block needs to consider the error propagation of valid pages by checking their moving flags. If a valid page in the victim has a valid moving flag, it will be copied to a normal block for the termination of the error propagation (lines 5-7). This can guarantee the validity of data on approximate blocks.

4.3.4 Differential Wear Leveling

Differential wear leveling spreads the wears of normal and approximate blocks in a well controlled manner. Dynamic and static wear leveling are adopted for normal blocks, while only dynamic wear leveling is adopted for approximate blocks. This is because the static wear leveling which copies data from a least-worn approximate block to a most-worn approximate block may double the error rate of data due to persistent errors. Moreover, differential wear leveling regulates allocation strategies for normal blocks. When normal blocks are sufficient, and the number of approximate blocks is lower than a pre-set threshold N_{approx}, a most-worn normal block is picked as a normal-transient block. The normal-transient block will be erased more frequently than other normal blocks due to the migration, which can accelerate the transition from a normal block to an approximate one. With an increasing number of approximate blocks, the free normal blocks are pressed. If the number of approximate blocks exceeds the threshold N_{approx}, the differential wear leveling reverses the allocation strategy for normal blocks: a least-worn normal block is assigned as a normal-transient block.

4.4 Data Attributes Cut-through

The Linux OS is layered into a number of distinct subsystems, the separations between them make it challenging to pass the approximate information of data from userland to kernel space. This section introduces how the approximate information of data is delivered through the whole Linux OS in a top-down way. Figure 4.5 shows the data attributes cut-through in the Linux I/O stack. Applications normally work on their own files; thus, we label the

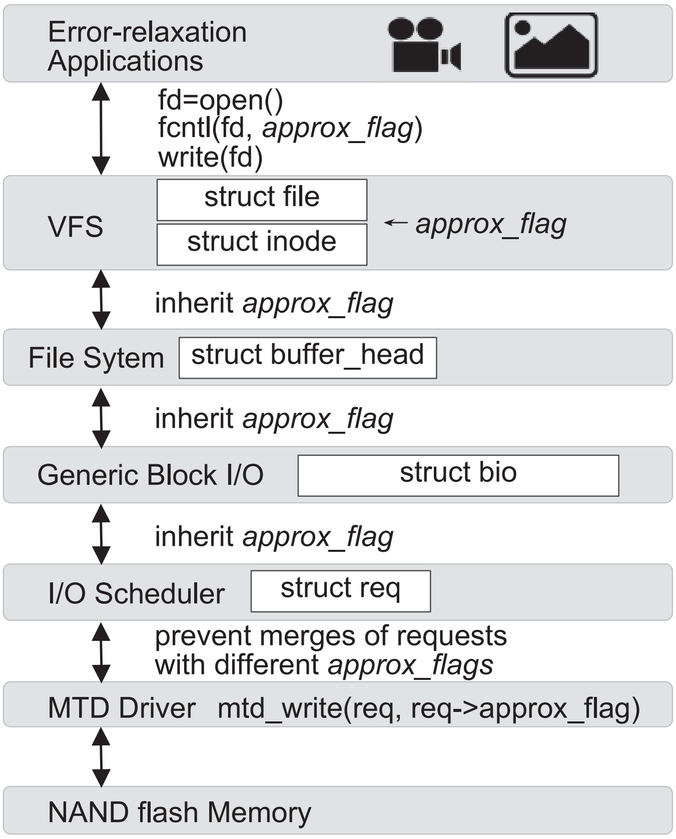


Figure 4.5: Data attributes cut-through in the Linux I/O stack.

approximate information of data through the file I/O operations. We add the $approx\_flag$ to the Linux kernel data structures "inode" and "file" after invoking "open(create)" and "fcntl" functions. The structure "inode" contains metadata about a file, and it represents a file in the virtual file system (VFS) which is an abstraction layer on top of individual file system. The structure "file" represents an open file, and it can pass any operation on the file, until the last close. With the user-space programs on a file, the write requests related to the file will inherit its approximate information. When an error-relaxation application invokes the "write" function to write data to a file, the structure "buffer_head" will inherit the $approx\_flag$ from the data structure "inode" in the file system layer. To cover different write mechanisms, we label the approximate information to requests at an assembly point before the requests enter the generic block I/O layer. Correspondingly, the data structures "bio" and "req" will successively inherit the flag in the generic block I/O layer. In the I/O scheduler layer, we prevent the merge of different types of requests by modifying the elevator algorithm. The error-relaxation data will not be merged with the error-free data even though they are adjacent to each other. In the MTD driver layer, we modify the driver to inform Rebirth-FTL to allocate the approximate or normal blocks for the corresponding requests according to their $approx\_flag$.

Our approach makes flash devices approximation-aware with minor OS modification and overhead. For example, disabling the merge of requests in the I/O scheduler has little effect on performance, due to the non-mechanical property of flash memories. Our approach keeps the original data access pattern, and provides compatibility across applications and the storage medium.

4.5 Lifetime Model

In this section, we formulate the lifetime analysis of a flash memory with Rebirth-FTL. We use the number of served write requests before the flash memory fails as the lifetime metric. The worn-out process of blocks is referred from [57]. Flash memory failure happens when the memory has exhausted all reserved good blocks. We focus on the benefits of reusing faulty blocks, thus we make the following assumptions. First, we assume that the lifetime of a flash block is only related to the P/E cycling. P/E cycling is the dominant error source, and other error sources have a positive correlation with it. Second, we assume that all the blocks evenly wear out.

According to the Table 4.1, we can easily obtain actual number of page writes occurring in the flash as being $(1+WAF) \cdot N_r$ for a request [49] [132]. Write amplification factor (WAF) is a numerical value that represents the amount of data written to the flash memory in relation to the amount of data that the OS has to write. Then we can obtain the number of erase operations incurred by a request:

$$E_{blk} = (1 + WAF) \cdot N_r / N_p \tag{4.1}$$

Conventional FTLs. Both normal blocks and reserved blocks can be used to store

Table 4.1: Analysis terminology

Term	Definition
WAF	Write Amplification Factor
N_r	Average page size requested from a write
N_p	Number of page in a block
N_n	Number of normal blocks in flash memory
N_r	Number of reserved blocks
E_n	Average allowed P/E cycles of a normal block
E_r	Average allowed P/E cycles of a reserved block
E_a	Average allowed P/E cycles of an approximate block
$P(\alpha)$	Proportion of error-relaxation data written on normal-transient blocks
$Q(\alpha)$	Migration probability induced by $P(\alpha)$
δ	Proportion of approximate blocks transformed from normal blocks

data in conventional FTLs. By summing up the allowed P/E cycles of all the blocks, we derive N_{ori}, the total number of allowed P/E cycles of the flash memory as follows:

$$N_{ori} = N_n \cdot E_n + N_r \cdot E_r \tag{4.2}$$

Further, we can derive the number of served write requests before a flash memory fails with a perfect wear leveling:

$$\begin{aligned} T_{ori} &= N_{ori}/E_{blk} \\ &= \frac{(N_n \cdot E_n + N_r \cdot E_r) \cdot N_p}{(1 + WAF) \cdot N_r} \end{aligned} \tag{4.3}$$

Rebirth-FTL. Rebirth-FTL can serve both error-free and error-relaxation data. To be fair, given a write requesting N_r pages, let $(1-\alpha) \cdot N_r$ denote the number of requested normal pages, then $\alpha \cdot N_r$ denotes the number of requested approximate pages. In the Rebirth-FTL, if approximate space is not enough to serve the requested approximate pages, part of the error-relaxation data will be stored on normal-transient blocks. Later, the data on normal-transient blocks can be migrated to the approximate blocks for block efficiency. Thus, the write traffic on normal blocks contains the requested normal pages and a part of the requested approximate pages, while the traffic on approximate blocks contains the rest of the requested approximate pages and the migration, and the migration traffic is a portion of the data stored on normal-transient blocks. Now, we can derive O_n and O_a, the actual number of normal and approximate page writes occurring in flash for a request, receptively.

$$O_n = (1 + WAF) \cdot ((1 - \alpha) \cdot N_r + \alpha \cdot N_r \cdot P(\alpha)) \tag{4.4}$$

$$O_a = (1 + WAF) \cdot (\alpha \cdot N_r \cdot (1 - P(\alpha))) + \alpha \cdot N_r \cdot P(\alpha) \cdot Q(\alpha)) \tag{4.5}$$

Hence, we can calculate the number of erase operations on normal (E_{nor}) and approximate (E_{appr}) blocks incurred by a request:

$$E_{nor} = O_n/N_p, E_{appr} = O_a/N_p \tag{4.6}$$

The total number of allowed erase operations in Rebirth-FTL contains two parts: N_{nor} for normal blocks and reserved blocks, and N_{appr} for approx blocks which are transited from a part of normal blocks.

$$\begin{aligned} N_{nor} &= N_n \cdot E_n + N_r \cdot E_r \\ N_{appr} &= \delta \cdot N_n \cdot E_a \end{aligned} \tag{4.7}$$

Therefore, we can derive the number of served write requests before the flash fails by achieving a minimum on normal and approximate blocks:

$$\begin{aligned} T_{rebirth} &= min\{N_{nor}/E_{nor}, N_{appr}/E_{appr}\} \\ &= min\{\frac{(N_n \cdot E_n + N_r \cdot E_r) \cdot N_p}{(1 + WAF) \cdot N_r \cdot (P(\alpha) \cdot \alpha + (1 - \alpha))}, \\ &\frac{N_n \cdot \delta \cdot N_p}{(1 + WAF) \cdot N_r \cdot \alpha \cdot (1 - P(\alpha) + P(\alpha) \cdot Q(\alpha))}\} \end{aligned} \tag{4.8}$$

Lifetime Analysis. The above model compares the lifetime of a flash memory with conventional FTLs and Rebirth-FTL. According to Equation 4.8, if we set α to 0, which means that all the requests are error-free, the lifetime of a flash memory with Rebirth-FTL is the same as that of conventional FTLs. For mixed requests, the lifetime improvement ratio for normal blocks with Rebirth-FTL is $1/(1 + (P(\alpha) - 1) \cdot \alpha)$. It is easy to see that improvements can be achieved by increasing the percentage of error-relaxation requests, and by reducing write traffic on normal-transient blocks. A straightforward way is to accommodate the supply relationship between the number of error-relaxation requests and the number of approximate blocks.

4.6 Experimental Evaluation

In this section, we evaluate our Rebirth-FTL design. We first introduce the experiment methodology, and then we present the performance results in several key metrics.

4.6.1 Experiment Setup

We use an embedded developing board to conduct our experiments. The board is equipped with a Cortex A8 processor, 512MB RAM, and 1G NAND flash memory. Each block has 128 pages, and each page is of 4KB. This board has Linux V3.0.8, and the flash memory is formatted with Ext2. Due to the hardware restrictions, we generate 32 approximate blocks which have up to a 2.4% bit error rate by continuously performing a number of erase operations without ECC. The approximate blocks with the specified error rate are determined by performing "read-after-write" in each erase operation. We mainly consider two types of workloads on the embedded board. First, the image and video are partially selected from [33] [54] as the error-relaxation applications, and the quality is measured by a metric: average peak-signal to noise ratio (PSNR). Second, the text files with random characters are error-free applications.

We use a simulator to evaluate the lifetime performance. The metric is the number

Table 4.2: Workload characteristics

Workloads	Function	# of requests	Write Ratio	Average write size
Mds	Media server	2848747	41.5%	7680B
Hm	Hardware monitoring	4602627	56.5%	8192B
Prn	Print server	16819297	46.0%	10240B
Proj	Project directories	65841031	15.2%	36352B
Prxy	Web Proxy	181157932	38.8%	11776B

of served write requests before a flash memory exhausts all of the reserved blocks. The Rebirth-FTL is modeled by heavily modified Flashsim [13]. We simulate a flash memory with 2GB raw capacity in which 20% of blocks are reserved space, and the memory initially only contains normal blocks. We use the erase number as the lifetime of a flash block. To accelerate aging, the remaining erase number for normal blocks is set to 1, and the numbers for reserved blocks and approximate blocks are set to 10,000 and 50,000 (the related error rate is 1.88% in Figure 3.1 in Section 3.1). Moreover, Rebirth-FTL allows up to 40% of normal blocks which will be transformed into approximate blocks. The threshold N_{approx} is set to the number of 20% of blocks. We conduct experiments using I/O traces from [7], as shown in Table 4.2. We randomly plug the $approx\_flag$ into the original traces according to the different ratios of the error-relaxation data.

4.6.2 Evaluation Results

We first show the comparisons of average write response time with different FTLs on the embedded board. Then we show the quality loss in varied proportions of error-relaxation requests on the embedded board. An image and a video with the $approx\_flag$ are accessed from the embedded board with Rebirth-FTL, which makes flash devices approximate-aware. After that, we demonstrate the improvement in lifetime by employing our Rebirth-FTL.

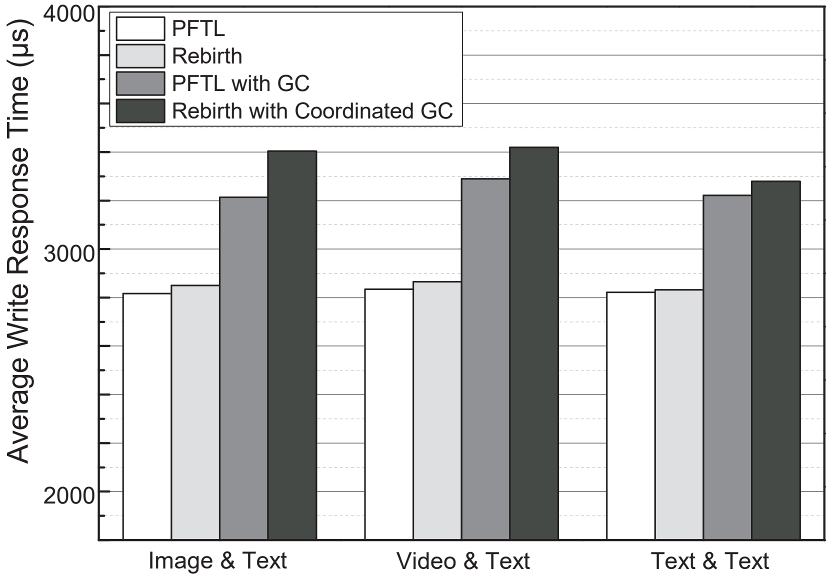
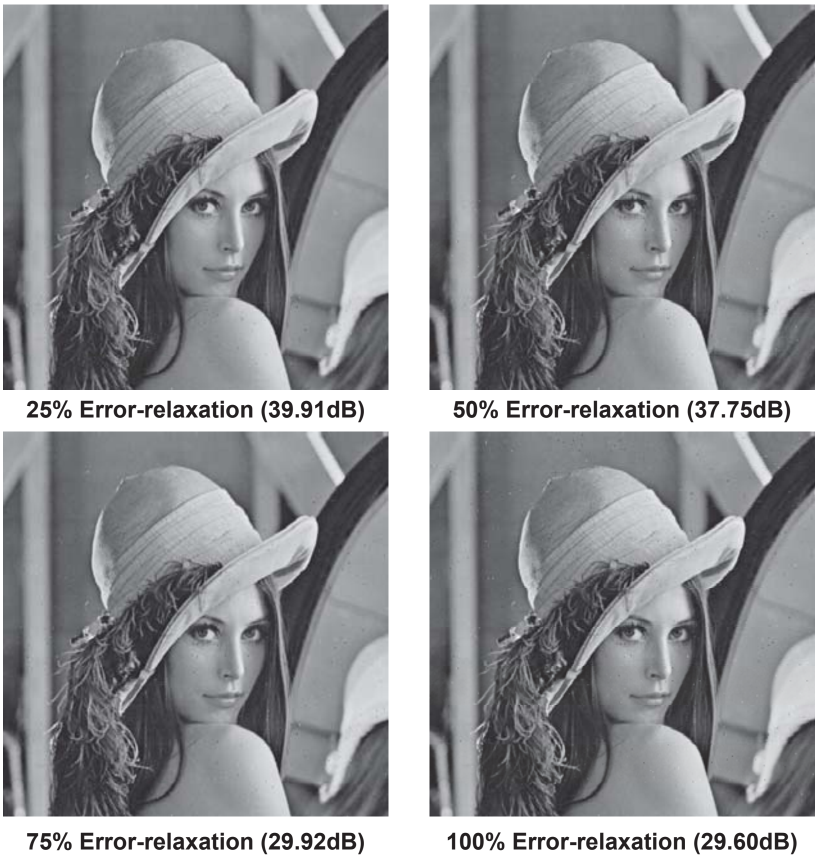


Figure 4.6: The average write request response time from Rebirth FTL and PFTL.

• **Response Time**.

Different from prior approximate storage techniques and FTL schemes, our Rebirth-FTL does not change any physical property, and it roots in a page-level mapping scheme. Considering the fairness, we compare our proposed Rebirth-FTL with an ideal page-level FTL (PFTL) on the embedded board. In order to fairly compare the response time with GC, we compact the whole addressable space to 16MB approximate space and 32MB normal space. Once the free space drops below 50%, the garbage collection process is triggered. Four obsolete blocks and normal-transient blocks are generated in advance for GC. The error-relaxation data (image and video) and error-free data (text) with the same size are repeatedly written in parallel to the flash memory, and the size of each write in sync mode is 4KB. Figure 4.6 presents the average response time of writes on normal and approximate blocks, while the writes for the file system metadata are excluded. Compared to the ideal PFTL scheme, Rebirth-FTL incurs up to 1.1% write overhead without GC in different applications. With coordinated GC, our Rebirth-FTL experiences a longer average response time than PFTL in the mixed workloads, with additional 5.9% and 3.9% latency. This is because Rebirth-FTL preferentially migrates valid pages from normal-transient blocks to approximate blocks for reclaiming obsolete blocks. Normally, most pages in the normal-transient blocks are valid,



25% Error-relaxation (39.91dB)

50% Error-relaxation (37.75dB)

75% Error-relaxation (29.92dB)

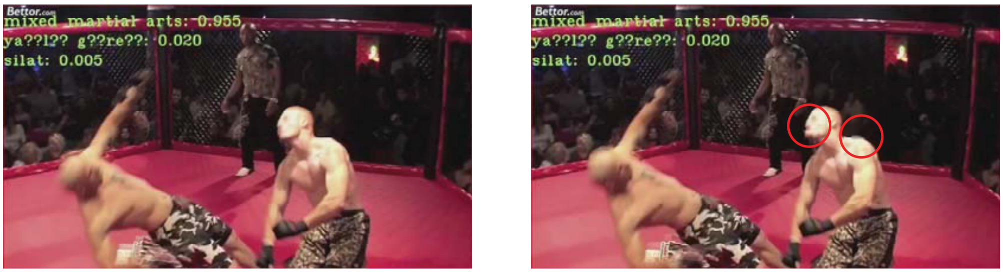
100% Error-relaxation (29.60dB)

Figure 4.7: Quality loss in PSNR with different percentage of error-relaxation requests.

which involves extra copies compared to those in PFTL. Due to the pre-set normal-transient blocks, our Rebirth-FTL preferentially reclaims these blocks and migrates valid pages. Thus, it incurs 1.8% additional latency in the text&text workloads compared with PFTL.

• **Impact on Quality**.

We first evaluate the quality of an image output on the embedded board with Rebirth-FTL. We store “Lenna” on the flash blocks in varied proportions of error-relaxation requests. The “Lenna” is a bitmap without any image compression technology. Figure 4.7 shows the quality loss of the image when it is read from the memory with Rebirth-FTL. It can be observed that the quality loss is 39.91dB at PSNR when only 25% of requests are delivered to the approximate blocks. The quality is reduced to 29.60dB when the image is totally stored on approximate blocks. Even so, the image is still acceptable for applications such as face recognition which normally consider a good image at PSNR of more than 28dB. The results



(a) Original Frame

(b) Frame output with Rebirth-FTL

Figure 4.8: One example of bad frames in the video.

Table 4.3: Quality Loss of an example of bad frames in PSNR.

Signal	PSNR
Luma	27.1dB
Chrominance	46.1dB
Chroma	37.7dB

verify that the lifetime of a flash memory can be significantly extended with the involvement of more approximate blocks, which also mirrors our proposed model.

To evaluate the quality of a video output with Rebirth-FTL, we use the “MSU Video Quality Measurement Tool” (VQMT) from [129]. A part of a video as the error-relaxation requests is stored on the embedded board, and then we read it out with Rebirth-FTL. Since the video is encoded by a compression technique, some frames are completely destroyed if the base part and the deltas both have errors. Figure 4.8 shows an example of the visible bad frames from the video. We can see that the frame only has two obvious parts of dead pixels (labeled by the red circle), and it is acceptable for users who are watching a boxing match. The video signal consists of three separate signals: Y for “Luma”, U for “Chrominance”, and V for “Chroma”. The VQMT tool evaluates the quality of bad frames at PSNR with those three signals, Table 4.3 shows the results of signals at PSNR. It can be observed that the quality losses are 27dB, 46.1dB, and 37.7dB at PSNR with three separate signals. Kan et al. [56] discuss that 27dB is the minimum Quality of Service (QoS) requirement for videos. Thus, some frames read from the embedded board with Rebirth-FTL meet the QOS

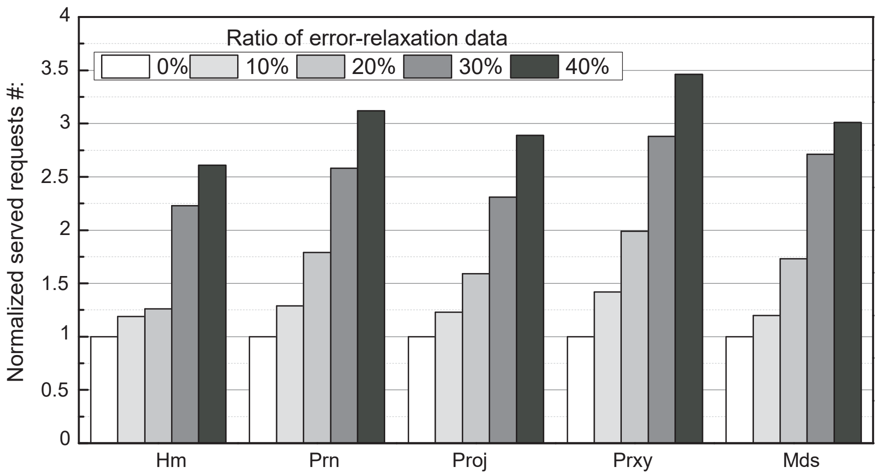


Figure 4.9: Normalized served requests #: comparing baseline and Rebirth-FTL.

requirements.

• **Lifetime**.

As it takes too long to reach the lifetime of NAND flash on the board, and the GC cost is insufficient within a limited space, for effective comparison, we use a simulator to evaluate lifetime performance when adopting Rebirth-FTL and the baseline PFTL. The normalized results with the different ratios of error-relaxation data are shown in Figure 4.9. It can be seen that Rebirth-FTL significantly increases the number of served requests before a flash memory fails over the baseline. On average, Rebirth-FTL achieves 1.26×, 1.67×, 2.54× and 3.01× improvement for the ratios of error-relaxation data equaling to 10%, 20%, 30% and 40% respectively. Rebirth-FTL maximizes the lifetime to 3.46× in workload $Prxy$. The results also verify our model: the lifetime extensions along with the ratios of error-relaxation data in Rebirth-FTL.

4.7 Summary

In this chapter, we have proposed Rebirth-FTL for lifetime optimization of flash memories by exploring the error-tolerance of data. Rebirth-FTL manages two spaces, and redesigns three key components in FTL: approximation-aware address mapping, coordinated garbage collection, and differential wear leveling. We also have developed a scheme that can pass approximate information from userland to kernel space in Linux. Moreover, a lifetime model has been presented for lifetime analysis. We have prototyped Rebirth-FTL on an embedded board and a simulator. The experimental results have shown that Rebirth-FTL can significantly improve the lifetime of flash memories, and maintain the service.

CHAPTER 5

OPTIMIZING CAUCHY REED-SOLOMON CODING VIA RERAM CROSSBARS IN SSD-BASED RAID SYSTEMS

5.1 Introduction

SSD-based RAID arrays, such as FlashArray from PureStorage [123] and XtremIO from EMC [31], have become ubiquitous,. They have exhibited superior performance to meet ever-increasing low-latency and high-throughput I/O requirements for big data applications. The fault-tolerance schemes employed by SSD-based RAID arrays provide data protection and maintain high-level service performance. For example, XtremIO introduces both row and diagonal parity calculations to protect data [31].

Methods for generating redundancy in a RAID system are varied, such as mirroring and erasure coding. More RAID systems have been employing erasure coding to protect data due to the lower storage overhead compared to other coding policies. In particular, erasure coding is a storage process through which a data object is separated into smaller fragments, and each of those fragments is encoded to generate the redundancy for fault-tolerance. Reed-Solomon (RS) codes are the most popular among erasure codes, which encode and decode with matrix multiplications [28]. RS codes have been extensively employed in cloud storage system such as Hadoop since they save significant storage space compared to 3-replication [117]. With the superior I/O performance of SSD-based RAID arrays, parity calculation performance becomes more and more important, especially for $degraded\ read$ which requests data from unavailable SSD disks. In the period between failure and recovery, the coding efficiency for reconstructions remains worthy exploring.

Reed-Solomon coding, building on the matrix multiplications for a RAID system, challenges current processor-based implementations such as a FPGA-based hardware controller [43]. First, the implementation of matrix multiplication involves Galois Field (GF) arithmetic [29], termed $GF(2^w)$ for w-bit words, which is computationally expensive on processors. Multiplication and division operations over $GF(2^w)$ rely on multiplication tables or discrete logarithms to perform. Cauchy Reed-Solomon (CRS) code [68] [94] coverts RS codes to a code with 1-bit words, and it replaces the expensive multiplications with additional XOR operations. However, it is still fussy, since processors need the special MMX and SSE instruction set extensions for speedup [118]. Second, the current processor-based implementations separate data coding into memory access and processor processing, which leads to a huge amount of data movement between memories and processors. Using the specific hardware accelerator may exacerbate the data movement overhead. In the case of GPU [68], utilizing GPUs to accelerate CRS coding requires reading bitmatrix from constant memory and read data from shared memory. Third, the reconstructions with RS coding is sub-optimal in RAID systems because of the $repair\ problem$ [111]. Even a single chunk is failed in a RS-encoded stripe, other survivors with the equal number of data chunks require to be transferred to the RAID controller for reconstruction, incurring many times overhead in repair bandwidth and disk I/O.

ReRAM can efficiently perform matrix-vector multiplication and sum operation in a crossbar structure, which inherently fits CRS coding. With the efficient capability in both storage and computation, ReRAM crossbar has been widely studied to accelerate several kinds of applications, including graph processing [44] [120] and neural network [19] [113] [119]. Our work shares the common principle of leveraging ReRAM crossbars to accelerate computations, but we aim at accelerating erasure coding in an SSD-based RAID system.

In this chapter, we propose a novel ReRAM-optimized RAID system for accelerating CRS coding, called Re-RAID. First, Re-RAID uses the ReRAM as an alternative main memory in both RAID and SSD controllers. ReRAM enjoys low latency and low energy consumption to be a good candidate for main memory, and ReRAM has the non-volatile feature of protecting data against an unexpected power loss. In addition to being storage memory, a portion of ReRAM memories in our design are configured in computational mode to accelerate erasure coding. Second, to alleviate the computing workloads of a RAID controller for a single failure, we propose a confluent Cauchy-Vandermonde matrix as the generator matrix for encoding, by which the first parity is the bitwise XOR-summing results with all the data chunks. When a single failure happens, Re-RAID can distribute the reconstruction task to other surviving data disks, and these data disks can leverage the ReRAM memory to recover lost data by performing XOR-summing with the first parity and other surviving data chunks within a stripe. Thus, the computing workloads on traditional processor-based implementation for recovering a single failure can be greatly alleviated, including constructing a decoding matrix and performing matrix-vector multiplication.

We conduct a series of experiments to compare our Re-RAID with a conventional processor-based implementation for CRS coding. The evaluation results show that Re-RAID can significantly improve encoding performance by up to 598×, and improve decoding performance for a single failure and multiple failures by up to 44.6× and 251×, respectively. To summarize, this chapter makes the following contributions:

- We propose Re-RAID which uses ReRAM as the main memory in both RAID and SSD controllers. Re-RAID leverages ReRAM crossbars to accelerate erasure coding.
- We propose a confluent Cauchy-Vandermonde matrix as the generator matrix for encoding. Re-RAID can distribute the reconstruction task to ReRAM memory of SSDs to recover a single failure without the decoding matrix.
- We demonstrate the effectiveness of Re-RAID by conducting a set of experiments.

The rest of this chapter is organized as follows. The next section gives basic background to this study and gives a motivation example. Section 5.3 describes the framework of Re-RAID, and presents how to perform Cauchy Reed-Solomon coding on ReRAM memory. Section 5.4 evaluates the performance of Re-RAID in several key metrics. Finally, Section 5.5 discusses other related work, and Section 5.6 concludes this chapter.

5.2 Background and Motivation

In this section, we first briefly present the background of the SSD-based RAID system, Cauchy Reed-Solomon codes, and ReRAM basics. Then we use a motivation example to illustrate the efficiency of performing CRS coding on an ReRAM crossbar.

5.2.1 SSD-based RAID System

An SSD-based RAID storage system consists of a RAID controller and multiple SSDs. Striping and parity are two commonly-used RAID techniques. The RAID systems add redundancy for fault-tolerance, and the parity data is generated by a module called Parity Generator inside a RAID controller. For example, the RAID-6 in which two parties are generated for each stripe can tolerate up to two failed drives at any time. In storage arrays, the frequency of a single failure is much higher than multiple failures, occupying more than 90% of failures [101] [114] [137]. A typical SSD includes an SSD controller, a DRAM buffer, and flash memory controllers connecting to flash memory. The DRAM-based buffer is used to buffer the read and write data to accelerate access speed.

5.2.2 Erasure Coding Process

Erasure coding is usually specified in an $(k+m)$ format: k data chunks and m parity chunks are spread over $(k+m)$ SSDs, and any k of those can recover data. Reed-Solomon codes use sophisticated linear algebra operations to generate parity. As shown in Figure 5.1, a codeword vector with 5 data chunks and 3 parity chunks is generated by multiplying a vector of 5 data chunks with a generator matrix G^T. Each chunk is a w-bit word in this example. The Vandermonde matrix [60] is usually used as a part of the generator matrix G^T in an encoding process.

A stripe with an RS (k, m) erasure coding can tolerate up to m failures. Figure 5.2 shows the decoding process to recover two lost data chunks. Suppose D_0 and D_3 fail, then the rows in G^T corresponding to the failed chunks should be deleted before inverting the

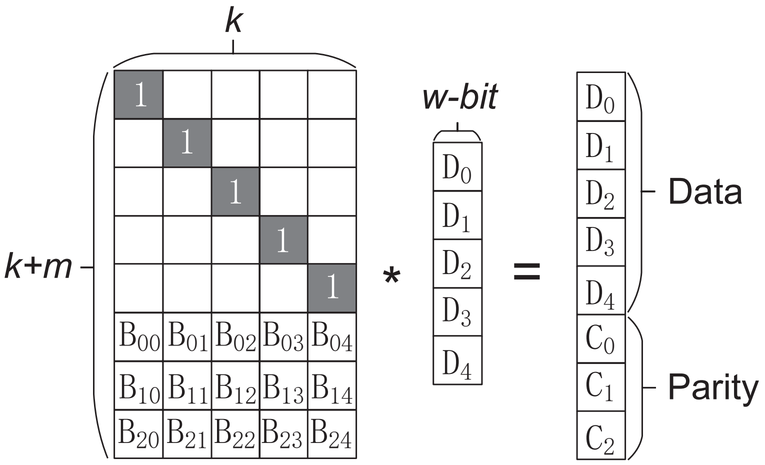
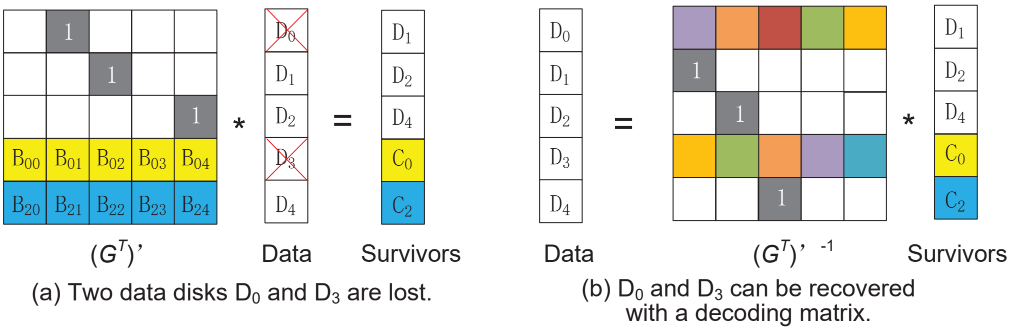


Figure 5.1: Encoding process with a generator matrix G^T.



(a) Two data disks D_0 and D_3 are lost.

(b) D_0 and D_3 can be recovered with a decoding matrix.

Figure 5.2: Decoding process with survivors and an inverse matrix.

G^T, represented as (G^T)'. In this figure, we choose three data chunks (D_1, D_2, D_4) and two parity chunks (C_0, C_2) as the survivors. According to the rule of matrix multiplication, the data can be recovered by multiplying a vector of k survivors and the inverse matrix of (G^T)'.

5.2.3 Cauchy Reed-Solomon Codes

Erasure coding partitions original data into stripes that involve data and coding information. Reed-Solomon codes are the most popular ones among erasure codes. RS codes treat data with w-bit words, and operate them as a number between 0 and 2^w-1 in $GF(2^w)$ arithmetic. Encoding and decoding with RS codes require matrix-vector multiplication and matrix inversion operations, these operations are much computational expensive in $GF(2^w)$.

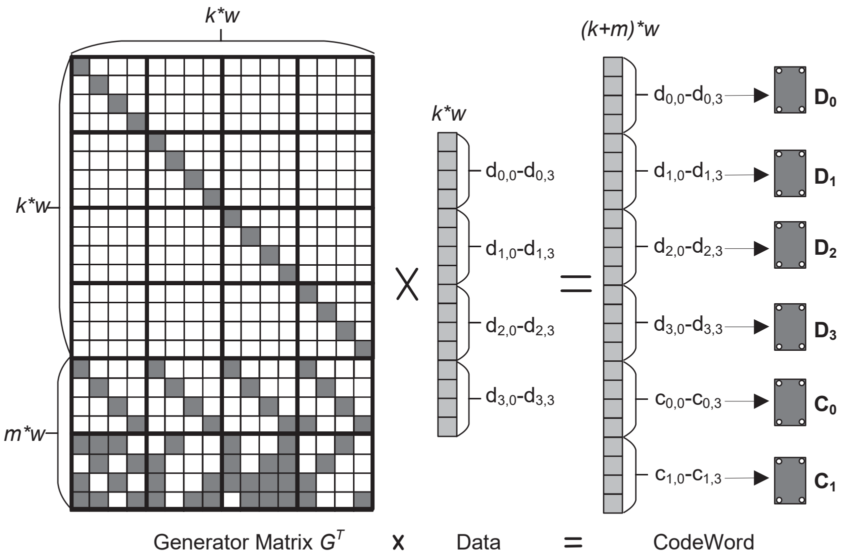


Figure 5.3: The matrix-vector representation of a Cauchy Reed-Solomon code with $k = 4, m = 2, w = 4$. Each element is one bit.

Cauchy Reed-Solomon codes covert RS codes to a code with 1-bit words. Thus, the expensive arithmetic in $GF(2^w)$ turns to be the bitwise AND and XOR operations in $GF(2)$. Figure 5.3 shows an encoding example of a CRS code. All $k * w$ data chunks are gathered to generate $m * w$ equal sized parity chunks by performing bitmatrix-vector multiplications. The set of $(k + w) * w$ data and parity chunks are separately delivered into different disks for fault-tolerance. The generator matrix usually consists of a $k * w \times k * w$ identity matrix and a $m * w \times k * w$ submatrix from an invertible matrix.

5.2.4 ReRAM Basics

A metal-oxide ReRAM cell consists of a top metal electrode, a metal-oxide resistive switch, and a bottom electrode [136]. By applying an external voltage across the ReRAM cell, the properties of conductive filament inside it change, leading to different resistances. A set of ReRAM cells can be interconnected as a dense crossbar architecture by wordlines and bitlines, which is better suited for main memory due to the small size of the area of a ReRAM cell [82] [140]. Figure 2.2 (b) in Section 2.2 shows an example of an ReRAM crossbar. When applying vector voltages V_1, V_2, ..., V_n to the wordlines, the current S_j at the end of jth bitline will represent the result of dot product operations, $\sum V_i \cdot W_{i,j}$.

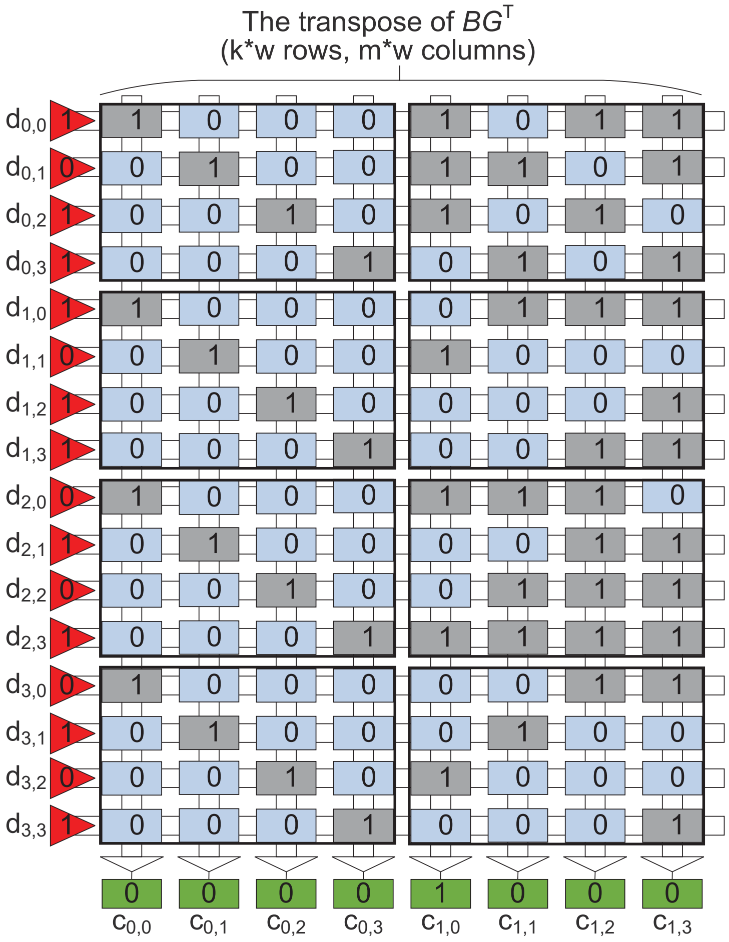

The transpose of BG^T
(k*w rows, m*w columns)
$d_{0,0}$ 1 | 1 0 0 0 | 1 0 1 1
$d_{0,1}$ 0 | 0 1 0 0 | 1 1 0 1
$d_{0,2}$ 1 | 0 0 1 0 | 1 0 1 0
$d_{0,3}$ 1 | 0 0 0 1 | 0 1 0 1
$d_{1,0}$ 1 | 1 0 0 0 | 0 1 1 1
$d_{1,1}$ 0 | 0 1 0 0 | 1 0 0 0
$d_{1,2}$ 1 | 0 0 1 0 | 0 0 0 1
$d_{1,3}$ 1 | 0 0 0 1 | 0 0 1 1
$d_{2,0}$ 0 | 1 0 0 0 | 1 1 1 0
$d_{2,1}$ 1 | 0 1 0 0 | 0 0 1 1
$d_{2,2}$ 0 | 0 0 1 0 | 0 1 1 1
$d_{2,3}$ 1 | 0 0 0 1 | 1 1 1 1
$d_{3,0}$ 0 | 1 0 0 0 | 0 0 1 1
$d_{3,1}$ 1 | 0 1 0 0 | 0 1 0 0
$d_{3,2}$ 0 | 0 0 1 0 | 1 0 0 0
$d_{3,3}$ 1 | 0 0 0 1 | 0 0 0 1
0 0 0 0 1 0 0 0
$c_{0,0}$ $c_{0,1}$ $c_{0,2}$ $c_{0,3}$ $c_{1,0}$ $c_{1,1}$ $c_{1,2}$ $c_{1,3}$

Figure 5.4: Example of performing Cauchy Reed Solomon coding on a ReRAM crossbar.

5.2.5 Motivation Example

The non-volatility, energy-saving and fast access make ReRAM memory a good candidate to be the main memory. Besides the storage capability, the ReRAM crossbars inherently have the capability of performing logic and arithmetic operations. We use an example to illustrate how to perform CRS encoding within a ReRAM crossbar, as shown in Figure 5.2.5. We choose a 16*8 ReRAM crossbar, each cell is a 1-bit SLC [83]. The input voltage at each wordline has 2 levels (1-bit), so the output at each bitline will be 6-bit (1-bit for SLC cell, 1-bit for input, and 4-bit for the number of wordlines on this 16*8 crossbar). In the encoding process, we only need the bottom $m * w$ rows of the generator matrix G^T presented in Figure 5.3, denoted as BG^T. The transpose of BG^T is mapped on this ReRAM crossbar, and the data $d_0 - d_3$ as the input are placed at the left side of wordlines. By activating the wordlines related to the input data, the outputs are the results of dot products in the ReRAM crossbar. We only take the lowest bit of the outputs as the XOR-summing result. The outputs are correct, the same as in $GF(2)$ by limiting the output to 1-bit precision at each bitline. For example, the current flowing in the first bitline is value ‘2’, which is the current sum across each cell in the first bitline, then we can obtain the result ‘0’ for $c_{0,0}$ after attaining the lowest bit of the value.

5.3 Re-RAID: A New SSD-based RAID System with ReRAM-accelerated CRS Coding

In this section, we present a new SSD-based RAID System with ReRAM-accelerated erasure coding. We first give an overview of our work, and then we present how to perform encoding and decoding with ReRAM.

5.3.1 Overview

We propose Re-RAID: a new SSD-based RAID system with ReRAM-accelerated erasure coding. Figure 5.5 depicts an overview of Re-RAID, and shows the micro-architecture of a ReRAM memory bank with a number of digital components. The traditional DRAM in both RAID and SSD controllers, a cache for holding data until it can be written to the drives, is replaced with ReRAM in the Re-RAID. Besides the non-volatile features of ReRAM, such as protecting data in the event of an unexpected power loss [77], ReRAM is also utilized to perform erasure coding so that the coding performance can be improved and the computing workloads of processors in the controllers can be alleviated. Re-RAID partitions the ReRAM memory into two modes: storage mode and computation mode. The ReRAM memory in storage mode serves NVcache function, and the memory that enters computation mode is used to encode and decode user data. In the encoding process, the transpose of a generator bitmatrix configured by the processor in the RAID controller is mapped to the multiple ReRAM crossbars in computation mode. Then the parity for a stripe can be obtained

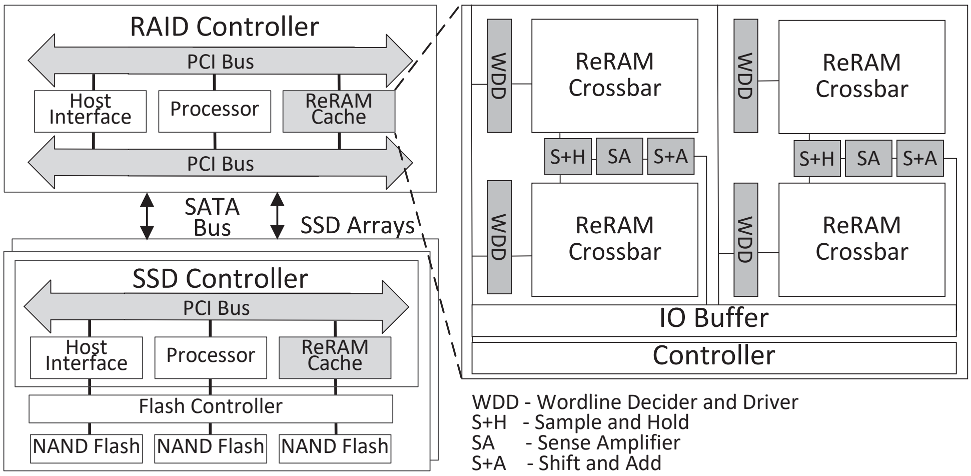


Figure 5.5: Overview of Re-RAID and microarchitecture of a ReRAM memory bank.

after activating the wordlines related to the input data and performing bitmatrix-vector multiplication on ReRAM crossbars. Similarly, in the decoding process for multiple failures, the transpose of a decoding matrix is mapped to multiple ReRAM crossbars for recovery. To further alleviate the computing workloads of the RAID controller, Re-RAID uses a confluent Cauchy-Vandermonde matrix as the generator matrix for encoding, thus the first parity is the bitwise XOR-summing results with all of the data chunks. When a single failure happens, Re-RAID can distribute the reconstruction task to other surviving data disks, and these data disks can leverage their ReRAM memory to recover lost data by performing XOR-summing with the first parity and other surviving data chunks within a stripe.

5.3.2 Cauchy Reed-Solomon Coding on Re-RAID

Figure 5.6 shows the data flow when performing CRS encoding in the RAID controller. When user data is delivered to the RAID controller, two main components, $Stripping\ Manager$ and $Parity\ Generator$, create stripes by slicing data into chunks and calculating parity with a CRS coding scheme. The data chunks within a stripe sliced by the $Stripping\ Manager$ are cached on the ReRAM memories in storage mode. Different from the traditional methods of generating parity on processors, parities are generated on the ReRAM memories in computation mode to which the data chunks within a stripe will be sent. The

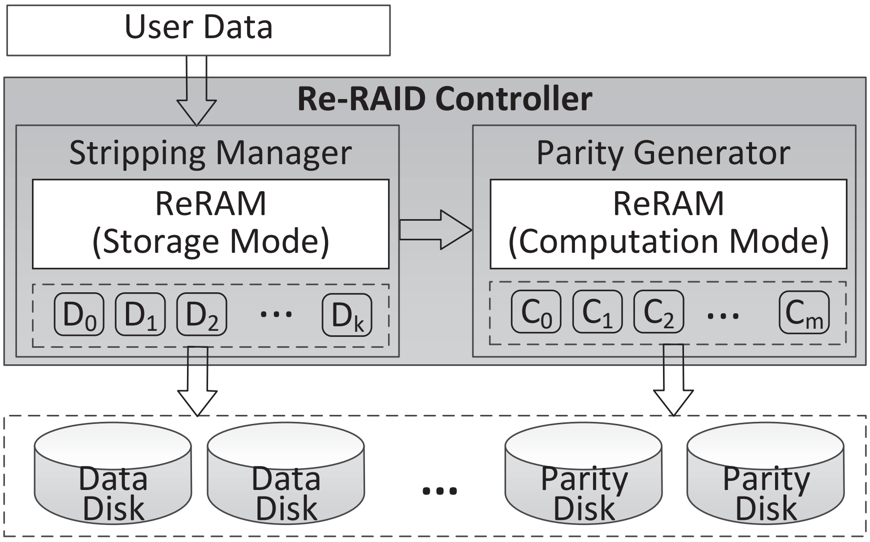


Figure 5.6: Data flow with CRS encoding on Re-RAID.

bottom $m * w$ rows of the generator matrix G^T will be mapped to the ReRAM crossbars in advance for computation, and the data chunks are turned into w-bit words at wordlines. Parity then can be attained by performing bitmatrix-vector multiplications on ReRAM crossbars. When the CRS encoding is finished for a stripe, a stripe including data part and parity part will be flushed into different SSD drives for fault-tolerance.

Re-RAID maps the transpose of BG^T on ReRAM crossbars for encoding. A BG^T bitmatrix comprises $k * w$ rows and $m * w$ columns (usually $k > m$). The values of k, m and k should be selected to accommodate various requirements of applications, including the scale, the devices layout, and the access performance [92]. Usually the values of k, m and k are not very large, so the BG^T bitmatrix can be fitted in one ReRAM crossbar. For example, Hadoop 3.0 normally adopts Reed-Solomon codes with ($k = 10, m = 4$ or $k = 6, m = 3$) [42]. Considering an 8-bit word in RS (10, 4) or RS (6, 3), a 512*512 crossbar can contain the entire BG^T bitmatrix. Moreover, Re-RAID can replicate the BG^T to multiple ReRAM crossbars, thus the encoding computation can be accelerated in a parallel way, and the memory bandwidth can be fully explored.

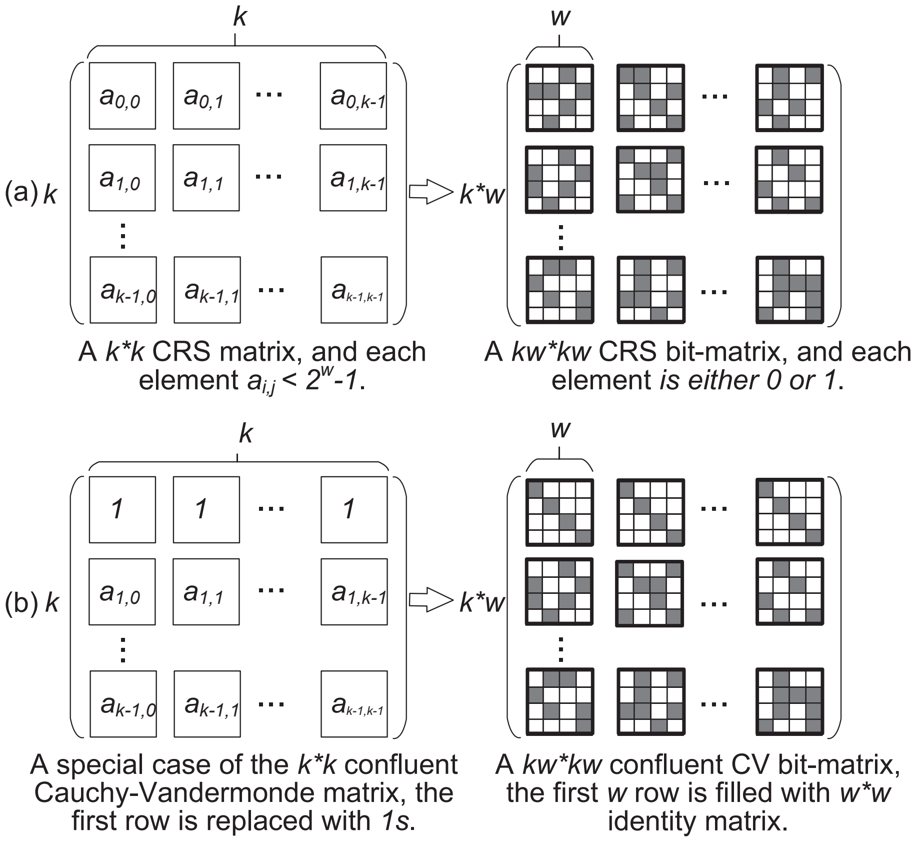

k
w
(a) k
$a_{0,0}$ $a_{0,1}$ ... $a_{0,k-1}$
$a_{1,0}$ $a_{1,1}$ ... $a_{1,k-1}$
$a_{k-1,0}$ $a_{k-1,1}$ ... $a_{k-1,k-1}$
k*w
A k*k CRS matrix, and each element $a_{i,j} < 2^w-1$.
A kw*kw CRS bit-matrix, and each element is either 0 or 1.
k
w
(b) k
1 1 ... 1
$a_{1,0}$ $a_{1,1}$ ... $a_{1,k-1}$
$a_{k-1,0}$ $a_{k-1,1}$ ... $a_{k-1,k-1}$
k*w
A special case of the k*k confluent Cauchy-Vandermonde matrix, the first row is replaced with 1s.
A kw*kw confluent CV bit-matrix, the first w row is filled with w*w identity matrix.

Figure 5.7: (a). Traditional Cauchy matrix and its bit-matrix with a given k and w. If $k \leq 2^{w-1}$, the matrix will be invertible. (b). An optimized code in our design.

5.3.3 Reconstruction on Re-RAID

• Single Failure.

A traditional way for recovering a single failure needs to construct a decoding matrix, which causes extremely high computation complexity, especially the inversion part. Moreover, the entire recovery process is executed in the processor-based RAID controller, which is not efficient if the RAID controller needs to handle other I/O tasks. To alleviate the computation workloads of a RAID controller, we propose a confluent Cauchy-Vandermonde matrix as the generator matrix for encoding by filling ones in the first row of BG^T. Then, we can distribute the reconstruction task for a single failure to SSDs, and each SSD can perform the task on ReRAM without the decoding matrix. Figure 5.7(a) shows a traditional Cauchy matrix and its $k * w$ dimension bitmatrix, and Figure 5.7(b) shows our proposed confluent Cauchy-Vandermonde matrix which comprises of $k-1$ rows of a Cauchy matrix, and a row

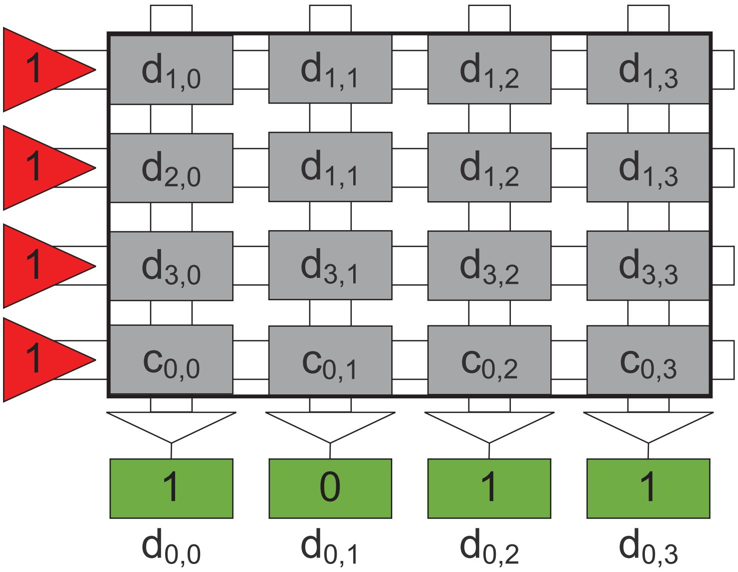


Figure 5.8: Recovering a single failure in the ReRAM crossbars on SSDs.

of ones which can be viewed as a part of Vandermonde matrix. Correspondingly, the first w rows in the bitmatrix are filled with the w dimension identity matrix. Therefore, the first parity is the XOR result of all the data chunks in the encoding process. If a single failure happens, Re-RAID can perform the bitwise XOR-summing with the first parity and other $k-1$ surviving data disks to recover. Note that filling ones in a coding matrix is not new, our contributions are that we fill ones on a CRS matrix and we decode it on ReRAM crossbars of SSDs.

The RAID controller in Re-RAID divides a reconstruction process for a single failure into a set of sub-tasks, and then delivers them to multiple SSDs which can utilize their ReRAM to recover a single failure. With the Cauchy-Vandermonde bitmatrix encoding, the first parity chunk is the XOR result of all the data chunks. Re-RAID recovers a single failure on ReRAM crossbars. The data chunks of a stripe from the surviving data disks, and its first parity chunk are gathered on a ReRAM crossbar in an SSD. Each chunk ($k * w$ bit) occupies one row in a crossbar. By activating wordlines related to a stripe with input voltage 1, the lost data can be obtained from the output at each bitline ($k * w$ bit). This is well illustrated with the example shown in Figure 5.8. Suppose the SSD D_0 in Figure 5.3 is failed. To recover it, Re-RAID requests the other surviving data $d_1 - d_3$ and the parity data c_0 to an available SSD, for example, the SSD D_1. Each chunk is mapped on a row in a ReRAM crossbar, the lost data $d_{0,0} - d_{0,3}$ can be attained by setting the input voltage of wordlines within this stripe to 1. The lowest 1-bit XOR-summing results in the bitlines are the lost data $d_{0,0} - d_{0,3}$. In addition, the recovery process can be performed with multi-level parallelism. First, Re-RAID can use ReRAM crossbars with massive memory bank-level parallelism. Second, a single failure can be recovered on different SSDs in a disk-level parallelism. This also reduces the overhead in repair bandwidth and disk I/O.

• **Multiple Failures**.

Re-RAID keeps the basic decoding rule for multiple failures: the decoding matrix multiplied by the survivors equals the original data. A confluent Cauchy-Vandermonde matrix is always invertible [130], so it can be formed into a decoding matrix for multiple failures. To recover multiple failures in Re-RAID, the collaboration between processors and ReRAM memories in a RAID controller is a promising solution. The processors construct a decoding matrix, and then map it to ReRAM memories in computation mode. By multiplying a vector of survivors and the decoding matrix on ReRAM memories, the multiple failures can be recovered. The decoding process for multiple failure is similar to the encoding process in the RAID controller.

5.3.4 Limitations

Our propose Re-RAID has several limitations. First, Re-RAID only considers a straightforward mapping scheme in which the transpose of BG^T is mapped to a ReRAM crossbar by rows. This mapping scheme incurs a low utilization rate of ReRAM crossbars if the BG^T is small-scale and the numbers of rows and columns in the BG^T are not equal. Second, Re-RAID only considers the small-scale encoding matrix so that it can fit on a single crossbar. If the encoding matrix is a large-scale one, such as k=500 and m=11 which is adopted in enterprise-level storage arrays, multiple memory banks are involved. They are challenging issues to implement a scheme that multiple ReRAM crossbars can collaborate to encode a stripe, and to design a pipeline for the computation efficiency. Third, Re-RAID fixes the

Table 5.1: The Configurations of ReRAM-based RAID controller and SSD controller.

RAID Controller	Quad-core at 1 GHz; 256MB Buffer; 600MB/s SATA Interface

SSD Controller	Quad-core at 400 MHz;
	64GB Capacity; 128MB Buffer;

ReRAM	512*512 SLC Cells/Crossbar; 4 Crossbars/Bank; read & write latency 48(ns)

number of ReRAM banks in both storage and computation mode, which is not efficient with the variation of memory usage in the controller.

5.4 Experimental Evaluation

In this section, we present the experimental setup and the evaluation results of our proposed Re-RAID.

5.4.1 Experiment Setup

The RAID and SSD controller in Re-RAID are modeled by heavily modified NVsim. The simulator implements the peripheral circuit, and encoding and decoding scheme on ReRAM crossbars. The detailed configurations of our Re-RAID system, from which the related timing parameters are derived [99] [20], are illustrated in Table 5.1. One-half of ReRAM memory is set in computation mode on both the RAID and SSD controllers.

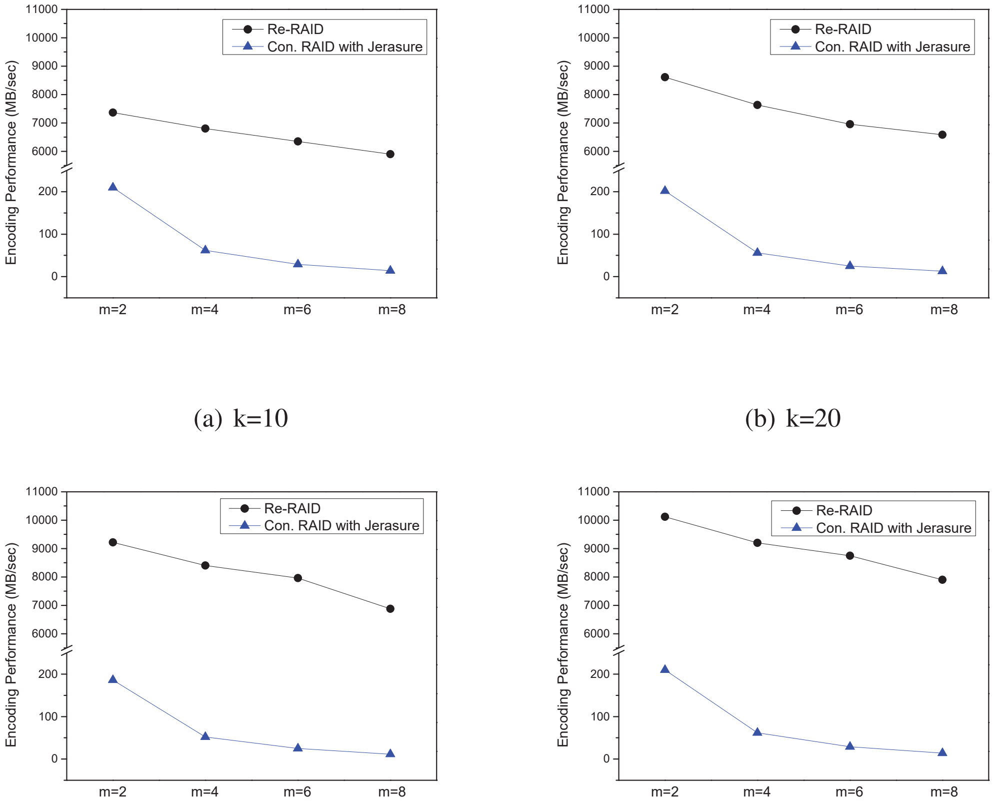
We focus on the performance of CRS coding, and we compare our Re-RAID with a dedicated hardware RAID controller. We use a modern CPU (Intel Q9550 with 2.83GHz) with the Jerasure library to simulate the processor-based implementation [93]. Note that this CPU outperforms any dedicated processor on a RAID card. We use "cauchy_good" as the coding techniques in the Jerasure library, and we only collect the coding performance to eliminate the effect of secondary storage I/O operations on different platforms. The performance is given in MB/Sec. The CRS coding performance is related to k, m, w, $packet\_size$ and $buffer\_size$. Considering fairness, the $packet\_size$ is set to $8Bytes$ due to the 64-bit machine word of CPU. The sum $k + m$ must be less than or equal to 2^w to make the coding matrix invertible, and we set $k = 10, 20, 30, 40$ and $m = 2, 4, 6, 8$. The $buffer\_size$ is the size of workload to be read at a time, and it is viewed as the level of parallelism to achieve a high throughput. The $buffer\_size$ is fixed to 8MB. We use a 1.37GB video from [86] as the input file.

5.4.2 Evaluation Results

We first show the comparisons of encoding performance in Re-RAID and a conventional processor-based RAID implementation. Then we show the reconstruction performance for a single failure and multiple failures with two implementations.

• **Encoding Performance**.

We evaluate the encoding performance of Re-RAID and a conventional processor-based RAID implementation (Con. RAID). Figure 5.9 displays the encoding performance of d-ifferent coding schemes. With our proposed Re-RAID, the encoding performance has a noteworthy improvement than the processor-based implementation, from 35× to 598×. This is because our Re-RAID inherently leverages the computational capability of ReRAM crossbars for XOR-based CRS encoding, and it wraps the data access within the memory, so as to reduce the data movement overhead. From the figure, we can see that the encoding performance of the processor-based implementation drops a little when m is fixed. When $m = 2$, the performance drops from 210.5MB/s with $k = 10$ to 168.7MB/s with $k = 40$. However, the performance dramatically drops from 210.5MB/s with $m = 2$ to 13.8MB/s with $m = 16$ if the k is fixed to 10. This is because the increase of m adds more access to data chunks, which brings more performance penalties in processor-based implementation. On the contrary, the encoding performance in Re-RAID improves up to 27% with the increase of k. Since the times of activating wordlines can be reduced with a larger k, thus the encoding



(a) k=10

(b) k=20

(c) k=30

(d) k=40

Figure 5.9: Comparison of encoding performance.

performance will be improved with the same parallelism. Moreover, the performance reduction with the increase of m is not overly large, up to 25.3%. The reduction is mainly related to more parity writes on ReRAM memory, since Re-RAID flushes the whole stripe until all parity chunks have prepared.

• **Decoding Performance for A Single Failure**.

Re-RAID performs reconstruction tasks for a single failure on SSDs, and we assume that the reconstruction task for a failed disk is evenly distributed to all the data disks. We encode a video by a confluent Cauchy-Vandermonde bitmatrix with $k = 10, 20, 30, 40$ and $m = 4$, and then make one of the data disks fail. Figure 5.10 shows the decoding performance to recover a single failure under the Re-RAID and the Con. RAID. We can see that our proposed Re-RAID outperforms the conventional processor-based implementation, by up to 44.6×.

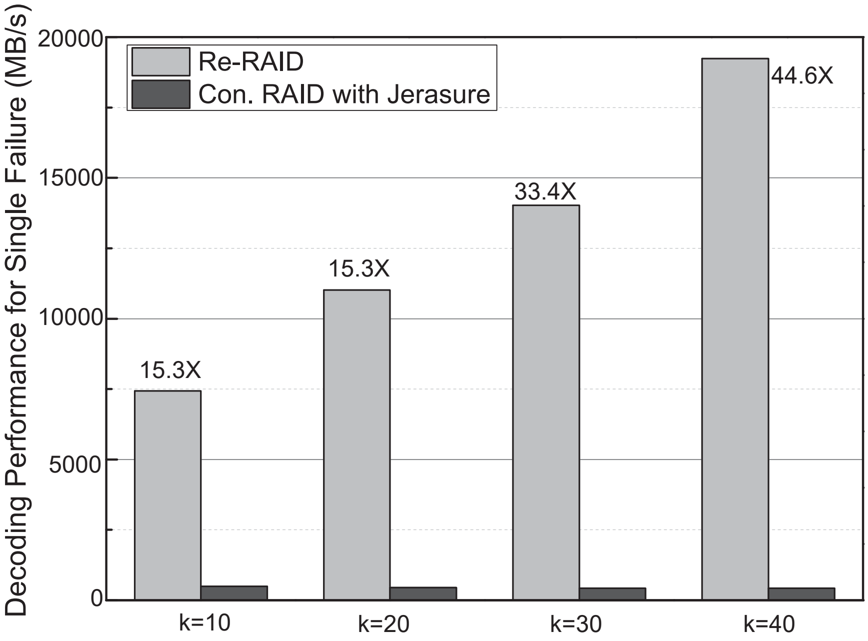


Figure 5.10: Comparison of decoding performance for a single failure.

The reason is that our Re-RAID performs the reconstruction tasks on ReRAM memory in SSDs, and achieves high parallelism on both memory-level and disk-level. Moreover, we can observe that the decoding performance drops a little with the increase of k in processor-based implementation, while our Re-RAID achieves the highest decoding performance when k is the largest. This is because a larger k makes more data chunks to perform XOR operations for recovery on an ReRAM crossbar, so the number of activating wordlines is reduced.

• **Decoding Performance for Multiple Failures**.

Re-RAID performs reconstruction tasks for multiple failures with the collaboration between processors and ReRAM memories in RAID controller. The decoding matrix related to the surviving data is mapped to multiple ReRAM crossbars with computation mode by the processors. With the $(k = 10, 20, 30, 40)$ and $(m = 4)$ encoding schemes, we fail four data disks among the disks. Figure 5.11 shows the comparison of the decoding performance for multiple failures. We can see that the decoding performance of Re-RAID is much higher than that in processor-based implementation, from $166\times$ to $251\times$. This is because Re-RAID recovers lost disks by multiplying a vector of survivors and the decoding matrix on ReRAM memories with high parallelism.

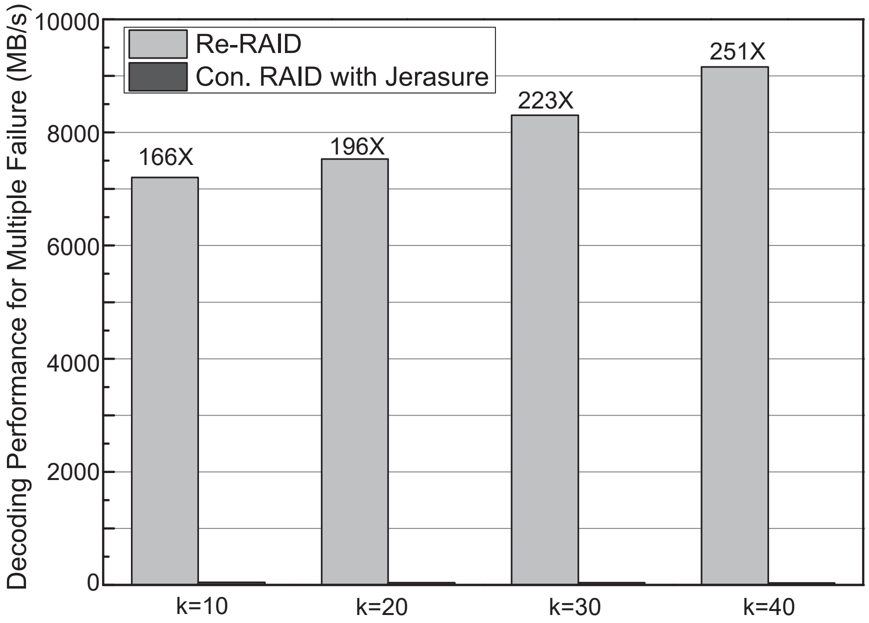


Figure 5.11: Comparison of decoding performance for multiple failures.

5.5 Other Related Work

In this section, related work involving erasure coding accelerator and active storage are discussed.

Erasure Coding Accelerator. The efficiency of performing erasure coding on traditional platforms considerably depends on the implementation of Galois field arithmetic. To accelerate erasure coding, some recent work for optimization has been studied. Kalcher et al. [53] presented a vectorized implementation for the streaming SIMD units of modern x86 processors, they implemented the table-less Galois filed multiplication on a GPU and demonstrated its efficiency. Liu et al. [68] proposed a GPU-based implementation of CRS called G-CRS. They also explored the memory access and parallelism to optimize coding performance. Brinkmann et al. [15] proposed a micro-driver architecture for CUDA-based accelerators from Linux kernel to accelerate encoding and decoding of Reed Solomon. Moreover, Lee et al. [61] and Roy et al. [106] proposed new hardware accelerator to implement Reed Solomon coding.

Active Storage. Active storage allows the computation on data inside the storage devices. Riedel et al. [103] proposed Active Disks which can run application-level code on processors of disk drives. They explored the computational power of commodity disks to execute four real-world data-intensive applications. Tiwari et al. [127] proposed Active Flash to perform data analysis on the SSD. The SSD controllers are explored to operate the resident data. Qin et al. [96] proposed an active storage framework for object-based storage device, they also proposed a hybrid approach of request-driven and policy-driven model for method execution.

5.6 Summary

In this chapter, we proposed Re-RAID for improving Cauchy Reed-Solomon coding performance of in the SSD-based RAID arrays by exploring the computation capability of ReRAM crossbars. Re-RAID uses ReRAM as the main memory in both RAID and SSD controllers, and performs encoding and decoding in ReRAM crossbars. To minimize the overhead of recovering a single failure, Re-RAID uses a confluent Cauchy-Vandermonde matrix as the generator matrix, and distributes the reconstruction task into multiple SSDs. The experimental results have shown that Re-RAID can achieve a significantly higher encoding and decoding performance compared with conventional processor-based implementation.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

In this thesis, we investigate utilizing non-volatile memories, ReRAM and NAND flash, to improve the performance of large-scale graph processing and erasure coding, and to improve the lifetime of SSD-based RAID arrays and flash memories. Specifically, this thesis is comprised of two parts for the optimization of big data applications:

- In the first part, we use ReRAM as the main memory to improve both computation and I/O performance for large-scale graphs processing. We propose a new ReRAM-based processing-in-memory architecture called RPBFS, in which graph data can be persistently stored and processed in place. We study the problem of breadth-first search. In RPBFS, we design an efficient graph mapping scheme in which a graph is distributively stored on multiple ReRAM memory banks, and we propose an efficient graph traversal algorithm with minimal data movement overhead. Moreover, we propose an analytical performance model to analyze the graph traversal efficiency on RPBFS, which can help us to identify bottlenecks and provide opportunities for our design. Experimental results show that the proposed RPBFS yields significant speedups on graph traversal performance.

- In the second part, we optimize the efficiency of big data storage with the NAND flash memory and ReRAM, resulting in a lower operational cost. In Chapter 3, we apply approximate storage via the interplay of RAID and SSD controllers to optimize the lifetime of SSDs arrays. We propose a cross-layer lifetime optimization framework, called FreeRAID. FreeRAID reuses faulty flash blocks to store approximate data, and it tightly couples the components in both RAID and SSDs controllers. With the goal of extend the lifetime of SSDs, FreeRAID combines the two techniques. First, with the knowledge of physical blocks in SSDs, FreeRAID efficiently allocates normal and faulty blocks to serve data with different error-tolerances, and makes different types of data error-isolated. Also, FreeRAID and the existing optimized RAID schemes can coalesce to further reduce write traffic on SSDs. Second, FreeRAID can dynamically switch FTL strategies on an SSD to maintain access performance and storage efficiency. Experimental results with various workloads show that FreeRAID outperforms conventional RAID solutions and FTLs.

In Chapter 4, we propose Rebirth-FTL, a pure software management in flash translation layer for the lifetime optimization of flash memories. Rebirth-FTL efficiently manages two address space, approximate space and normal space, with efficient address mapping, coordinated garbage collection and differential wear leveling. To pass the approximate information of data from userland to kernel space, we demonstrate how to pass the approximate information from applications to flash devices through the whole Linux OS in a top-down way. Moreover, we analyze the benefits of the lifetime of a flash memory with Rebirth-FTL by a lifetime model. We have implemented and deployed Rebirth-FTL on an embedded development board and a simulator to demonstrate its effectiveness.

In Chapter 5, we propose Re-RAID to optimize erasure coding performance with ReRAM in SSD-based RAID arrays. Re-RAID uses ReRAM as main memory in both RAID and SSD controllers, which allows erasure coding can be processed in ReRAM crossbars. To minimize the overhead for recovering a single failure, we propose a confluent Cauchy-Vandermonde matrix as the generator matrix, in which SSDs can leverage their ReRAM memory to recover a single lost disk, which can greatly alleviate the computing workloads of a RAID controller. For multiple failures, processors and ReRAM memories in the RAID controller work in close collaboration to recover lost data. Experimental results show that the proposed scheme improves both encoding and decoding performance compared with conventional processor-based implementations.

6.2 Future Work

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

- First, we plan to explore an optimized graph distribution scheme by considering the effect on traversal performance and the issue of storage efficiency.
- Second, we plan to explore the possibility of performing the high sparse matrix-vector multiplication operations with compressed format to support other graph algorithms, such as page-rank algorithm.
- Third, the mobile phone is a typical embedded storage system with flash memory. We will explore how to leverage the approximate storage to improve the efficiency of TrustZone in Android [23].
- Finally, we will examine the interference of erasure coded stripes from multiple clients in cloud storage systems.

REFERENCES

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. *ACM SIGARCH Computer Architecture News*, 43(3):105–117, 2016.

[2] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pages 336–348. IEEE, 2015.

[3] Berkin Akin, Franz Franchetti, and James C Hoe. Data reorganization in memory using 3d-stacked dram. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 131–143. ACM, 2015.

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

[5] Fabien Alibart, Ligang Gao, Brian D Hoskins, and Dmitri B Strukov. High precision tuning of state for memristive devices by adaptable variation-tolerant algorithm. *Nanotechnology*, 23(7):075201, 2012.

[6] Salman Aslam. Twitter by the numbers: Stats, demographics & fun facts. *Internet Stats*, 2019.

[7] Storage Networking Industry Association. Snia iotta repository. *Microsoft Enterprise Traces, Colorado Springs, Colorado (iotta. snia. org/traces/130)*, 2011.

[8] Rodolfo Azevedo, John D Davis, Karin Strauss, Parikshit Gopalan, Mark Manasse, and Sergey Yekhanin. Zombie memory: Extending memory lifetime by reviving

dead blocks. In *ACM SIGARCH Computer Architecture News*, pages 452–463. ACM, 2013.

[9] Mahesh Balakrishnan, Asim Kadav, Vijayan Prabhakaran, and Dahlia Malkhi. Differential raid: Rethinking raid for ssd reliability. *ACM Transactions on Storage (TOS)*, 6(2):4, 2010.

[10] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. Near-data processing: Insights from a micro-46 workshop. *IEEE Micro*, 34(4):36–42, 2014.

[11] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3-4):137–148, 2013.

[12] Scott Beamer, Krste Asanovic, and David Patterson. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pages 56–65. IEEE, 2015.

[13] Matias Bjørling. Extended flashsim. 2011.

[14] Mario Blaum, Jim Brady, Jehoshua Bruck, and Jai Menon. Evenodd: An efficient scheme for tolerating double disk failures in raid architectures. *IEEE Transactions on computers*, 44(2):192–202, 1995.

[15] André Brinkmann and Dominic Eschweiler. A microdriver architecture for error correcting codes inside the linux kernel. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 35. ACM, 2009.

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

[17] 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*, pages 1–6. ACM, 2014.

[18] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, page 1. ACM, 2015.

[19] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 27–39. IEEE Press, 2016.

[20] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R Ganger. Active disk meets flash: A case for intelligent ssds. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM, 2013.

[21] Ching-Che Chung and Hao-Hsiang Hsu. Partial parity cache and data cache management method to improve the performance of an ssd-based raid. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(7):1470–1480, 2014.

[22] Bram Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72, 2003.

[23] Tim Cooijmans, Joeri de Ruiter, and Erik Poll. Analysis of secure key storage solutions on android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, pages 11–20. ACM, 2014.

[24] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 1–14. USENIX Association Berkeley, CA, USA, 2004.

[25] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[26] Jinhua Cui, Youtao Zhang, Liang Shi, Chun Jason Xue, Weiguo Wu, and Jun Yang. Approxftl: On the performance and lifetime improvement of 3d nand flash based ssds. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2017.

[27] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[28] Hoang Dau, Iwan Duursma, Han Mao Kiah, and Olgica Milenkovic. Repairing reed-solomon codes with multiple erasures. *IEEE Transactions on Information Theory*, 2018.

[29] Leonard Eugene Dickson. *Linear groups: With an exposition of the Galois field theory*. Courier Corporation, 2003.

[30] Xiangyu Dong, Cong Xu, Norm Jouppi, and Yuan Xie. Nvsim: A circuit-level performance, energy, and area model for emerging non-volatile memory. In *Emerging Memory Technologies*, pages 15–50. Springer, 2014.

[31] Dell EMC. Introduction to dell emc xtremio x2 storage array, 2018.

[32] Facebook. Facebook reports third quarter 2018 results. Technical report, Facebook, 2018.

[33] Li Fei-Fei, Rob Fergus, and Pietro Perona. Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories. *Computer vision and Image understanding*, 106(1):59–70, 2007.

[34] Amherst Laboratory for Advanced System Software. *Umass trace repository*, 2012.

[35] Yingxun Fu and Jiwu Shu. D-code: An efficient raid-6 code to optimize i/o loads and read performance. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 603–612. IEEE, 2015.

[36] Yingxun Fu, Jiwu Shu, Xianghong Luo, Zhirong Shen, and Qingda Hu. Short code: An efficient raid-6 mds code for optimizing degraded reads and partial stripe writes. *IEEE Transactions on Computers*, 66(1):127–137, 2017.

[37] Jie Guo, Zhijie Chen, Danghui Wang, Zili Shao, and Yiran Chen. Dpa: A data pattern aware error prevention technique for nand flash lifetime extension. In *2014 19th Asia and South Pacific Design Automation Conference*, 2014.

[38] Qing Guo, Karin Strauss, Luis Ceze, and Henrique S Malvar. High-density image storage using approximate memory cells. In *ACM SIGPLAN Notices*, volume 51, pages 413–426. ACM, 2016.

[39] Zhaohui Guo, Yuuki Nishikawa, Roberto Yusi Omaki, Takao Onoye, and Isao Shirakawa. A low-complexity fec assignment scheme for motion jpeg2000 over wireless network. *IEEE Transactions on Consumer Electronics*, 52(1):81–86, 2006.

[40] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. *DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings*, volume 44. ACM, 2009.

[41] Venkatesan Guruswami and Mary Wootters. Repairing reed-solomon codes. *IEEE Transactions on Information Theory*, 63(9):5684–5698, 2017.

[42] Hadoop and Apache. Hadoop 3.0, 2017.

[43] Volker Hampel, Peter Sobe, and Erik Maehle. Experiences with a fpga-based reed/solomon-encoding coprocessor. *Microprocessors and Microsystems*, 32(5-6):313–320, 2008.

[44] Lei Han, Zhaoyan Shen, Duo Liu, Zili Shao, H Howie Huang, and Tao Li. A novel reram-based processing-in-memory architecture for graph traversal. *ACM Transactions on Storage (TOS)*, 2018.

[45] Lei Han, Zhaoyan Shen, Zili Shao, H Howie Huang, and Tao Li. A novel reram-based processing-in-memory architecture for graph computing. In *Non-Volatile Memory*

Systems and Applications Symposium (NVMSA), 2017 IEEE 6th, pages 1–6. IEEE, 2017.

[46] Lei Han, Zhaoyan Shen, Zili Shao, and Tao Li. Optimizing raid/ssd controllers with lifetime extension for flash-based ssd array. In *Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 44–54. ACM, 2018.

[47] Robin Harris. Ssd reliability in the real world: Google's experience. Technical report, ZDNet, 2016.

[48] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 78–88. IEEE, 2011.

[49] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, page 10. ACM, 2009.

[50] Joe Jeddeloh and Brent Keeth. Hybrid memory cube new dram architecture increases density and performance. In *VLSI Technology (VLSIT), 2012 Symposium on*, pages 87–88. IEEE, 2012.

[51] Djordje Jevdjic, Karin Strauss, Luis Ceze, and Henrique S Malvar. Approximate storage of compressed and encrypted videos. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 361–373. ACM, 2017.

[52] Xavier Jimenez, David Novo, and Paolo Ienne. Wear unleveling: improving nand flash lifetime by balancing page endurance. In *Proceedings of the 12th USENIX conference on File and Storage Technologies*, pages 47–59. USENIX Association, 2014.

[53] Sebastian Kalcher and Volker Lindenstruth. Accelerating galois field arithmetic for reed-solomon erasure codes in storage applications. In *2011 IEEE International Conference on Cluster Computing*, pages 290–298. IEEE, 2011.

[54] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014.

[55] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[56] Asiya Khan, Lingfen Sun, and Emmanuel Ifeachor. Video quality assessment as impacted by video content over wireless networks. *International Journal on Advances in Networks and Services*, 2(2&3), 2009.

[57] Jaeho Kim, Jongmin Lee, Jongmoo Choi, Donghee Lee, and Sam H Noh. Improving ssd reliability with raid via elastic striping and anywhere parity. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2013.

[58] Jesung Kim, Jong Min Kim, Sam H Noh, Sang Lyul Min, and Yookun Cho. A space-efficient flash translation layer for compactflash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, 2002.

[59] Sohyun Koo, Se Jin Kwon, Sungsoo Kim, and Tae-Sun Chung. Dual raid technique for ensuring high reliability and performance in ssd. In *2015 IEEE/ACIS 14th International Conference on Computer and Information Science (ICIS)*, pages 399–404. IEEE, 2015.

[60] Jérome Lacan and Jérome Fimes. Systematic mds erasure codes based on vandermonde matrices. *IEEE Communications Letters*, 8(9):570–572, 2004.

[61] Jung H Lee, Jaesung Lee, and Myung H Sunwoo. Design of application-specific instructions and hardware accelerator for reed-solomon codecs. *EURASIP Journal on Applied Signal Processing*, 2003:1346–1354, 2003.

[62] Jure Leskovec and Andrej Krevl. {SNAP Datasets}:{Stanford} large network dataset collection. 2015.

[63] Bingzhe Li, Meng Yang, Soheil Mohajer, Weikang Qian, and David J Lilja. Tier-code: An xor-based raid-6 code with improved write and degraded-mode read performance. In *2018 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–10. IEEE, 2018.

[64] Jing Li, Chao-I Wu, Scott C Lewis, Jackie Morrish, Tien-Yen Wang, Richard Jordan, Tom Maffitt, Matthew Breitwisch, Alejandro Schrott, Roger Cheek, et al. A novel reconfigurable sensing scheme for variable level storage in phase change memory. In *Memory Workshop (IMW), 2011 3rd IEEE International*, pages 1–4. IEEE, 2011.

[65] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *Proceedings of the 53rd Annual Design Automation Conference*, page 173. ACM, 2016.

[66] Yongkun Li, Helen HW Chan, Patrick PC Lee, and Yinlong Xu. Elastic parity logging for ssd raid arrays. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 49–60. IEEE, 2016.

[67] Yongkun Li, Patrick PC Lee, and John CS Lui. Analysis of reliability dynamics of ssd raid. *IEEE Transactions on Computers*, 65(4):1131–1144, 2016.

[68] Chengjian Liu, Qiang Wang, Xiaowen Chu, and Yiu-Wing Leung. G-crs: Gpu accelerated cauchy reed-solomon coding. *IEEE Transactions on Parallel & Distributed Systems*, (7), 2018.

[69] Duo Liu, Tianzheng Wang, Yi Wang, Zili Shao, Qingfeng Zhuge, and Edwin H-M Sha. Application-specific wear leveling for extending lifetime of phase change memory in embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(10):1450–1462, 2014.

[70] Duo Liu, Yi Wang, Zhiwei Qin, Zili Shao, and Yong Guan. A space reuse strategy for flash translation layers in slc nand flash memory storage systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(6):1094–1107, 2012.

[71] Hang Liu and H Howie Huang. Enterprise: Breadth-first graph traversal on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 68. ACM, 2015.

[72] Hang Liu, H Howie Huang, and Yang Hu. ibfs: Concurrent breadth-first search on gpus. In *Proceedings of the 2016 International Conference on Management of Data*, pages 403–416. ACM, 2016.

[73] Xiaoxiao Liu, Mengjie Mao, Beiye Liu, Hai Li, Yiran Chen, Boxun Li, Yu Wang, Hao Jiang, Mark Barnell, Qing Wu, et al. Reno: A high-efficient reconfigurable neuromorphic computing accelerator design. In *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pages 1–6. IEEE, 2015.

[74] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 273–282. ACM, 2013.

[75] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proceedings of the 11th USENIX conference on File and Storage Technologies*, pages 257–270. USENIX Association, 2013.

[76] Yixin Luo, Yu Cai, Saugata Ghose, Jongmoo Choi, and Onur Mutlu. Warm: Improving nand flash memory lifetime with write-hotness aware retention management.

In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pages 1–14. IEEE, 2015.

[77] Bill Lynn and Ansh Gupta. Non-volatile cache for host-based raid controllers. *Dell Technical Write Paper*, 2011.

[78] Dimitrov Martin and Strickland Carl. Intel power gadget. *Intel Corporation*, 7, 2016.

[79] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. In *ACM SIGPLAN Notices*, volume 47, pages 117–128. ACM, 2012.

[80] Nooshin Mirzadeh, Yusuf Onur Koçberber, Babak Falsafi, and Boris Grot. Sort vs. hash join revisited for near-memory execution. In *5th Workshop on Architectures and Systems for Big Data (ASBD 2015)*, number EPFL-TALK-209111, 2015.

[81] Sangwhan Moon and AL Reddy. Does raid improve lifetime of ssd arrays? *ACM Transactions on Storage (TOS)*, 12(3):11, 2016.

[82] Dimin Niu, Cong Xu, Naveen Muralimanohar, Norman P Jouppi, and Yuan Xie. Design of cross-point metal-oxide reram emphasizing reliability and cost. In *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*, pages 17–23. IEEE, 2013.

[83] Dimin Niu, Qiaosha Zou, Cong Xu, and Yuan Xie. Low power multi-level-cell resistive memory design with incomplete data mapping. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pages 131–137. IEEE, 2013.

[84] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C Hoe, José F Martínez, and Carlos Guestrin. Graphgen: An fpga framework for vertex-centric graph computation. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 25–28. IEEE, 2014.

[85] NVIDIA. Cuda toolkit documentation. Technical report, NVIDIA, 2017.

[86] Sangmin Oh, Anthony Hoogs, Amitha Perera, Naresh Cuntoor, Chia-Chih Chen, Jong Taek Lee, Saurajit Mukherjee, JK Aggarwal, Hyungtae Lee, Larry Davis, et al. A large-scale benchmark dataset for event recognition in surveillance video. In *Computer vision and pattern recognition (CVPR), 2011 IEEE conference on*. IEEE, 2011.

[87] Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H Noh. Improving performance and lifetime of the ssd raid-based host cache through a log-structured approach. In *1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, page 5. ACM, 2013.

[88] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. Energy efficient architecture for graph analytics accelerators. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 166–177. IEEE, 2016.

[89] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[90] Daniel Palomino, Muhammad Shafique, Altamiro Susin, and Jörg Henkel. Thermal optimization using adaptive approximate computing for video coding. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pages 1207–1212. IEEE, 2016.

[91] Yubiao Pan, Yongkun Li, Yinlong Xu, and Zhipeng Li. Grouping-based elastic striping with hotness awareness for improving ssd raid performance. In *2015 45th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 160–171. IEEE, 2015.

[92] James S Plank, Kevin M Greenan, and Ethan L Miller. Screaming fast galois field arithmetic using intel simd instructions. In *FAST*, pages 299–306, 2013.

[93] James S Plank, Scott Simmerman, and Catherine D Schuman. Jerasure: A library in c/c++ facilitating erasure coding for storage applications-version 1.2. *University of Tennessee, Tech. Rep. CS-08-627*, 23, 2008.

[94] James S Plank and Lihao Xu. Optimizing cauchy reed-solomon codes for fault-tolerant network storage applications. In *Network Computing and Applications, 2006. NCA 2006. Fifth IEEE International Symposium on*, pages 173–180. IEEE, 2006.

[95] Seth H Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. Ndc: Analyzing the impact of 3d-stacked memory+ logic devices on mapreduce workloads. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pages 190–200. IEEE, 2014.

[96] Lingjun Qin and Dan Feng. Active storage framework for object-based storage device. In *Advanced Information Networking and Applications, 2006. AINA 2006. 20th International Conference on*, volume 2, pages 97–101. IEEE, 2006.

[97] Zhiwei Qin, Yi Wang, Duo Liu, and Zili Shao. Demand-based block-level address mapping in large-scale nand flash storage systems. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 173–182. ACM, 2010.

[98] Zhiwei Qin, Yi Wang, Duo Liu, Zili Shao, and Yong Guan. Mnftl: An efficient flash translation layer for mlc nand flash memory storage systems. In *Proceedings of the 48th Design Automation Conference*, pages 17–22. ACM, 2011.

[99] Morteza Ramezani, Nima Elyasi, Mohammad Arjomand, Mahmut T Kandemir, and Anand Sivasubramaniam. Exploring the impact of memory block permutation on performance of a crossbar reram main memory. In *Workload Characterization (IISWC), 2017 IEEE International Symposium on*. IEEE, 2017.

[100] Ashish Ranjan, Swagath Venkataramani, Xuanyao Fong, Kaushik Roy, and Anand Raghunathan. Approximate storage for energy efficient spintronic memories. In *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pages 1–6. IEEE, 2015.

[101] KV Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers. *ACM SIGCOMM Computer Communication Review*, 44(4):331–342, 2015.

[102] Pedro Reviriego, Juan A Maestro, and Mark F Flanagan. Error detection in majority logic decoding of euclidean geometry low density parity check (eg-ldpc) codes. *IEEE transactions on very large scale integration (VLSI) systems*, 21(1):156–159, 2013.

[103] Erik Riedel, Garth Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia applications. In *Proceedings of 24th Conference on Very Large Databases*, pages 62–73. Citeseer, 1998.

[104] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[105] Pooja Roy, Rajarshi Ray, Chundong Wang, and Weng Fai Wong. Asac: Automatic sensitivity analysis for approximate computing. In *ACM SIGPLAN Notices*, volume 49, pages 95–104. ACM, 2014.

[106] Sourav Roy, Martin Bucker, W Wilhelm, and BS Panwar. Reconfigurable hardware accelerator for a universal reed solomon codec. In *Circuits and Systems for Communications, 2002. Proceedings. ICCSC'02. 1st IEEE International Conference on*, pages 158–161. IEEE, 2002.

[107] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, page 22. ACM, 2013.

[108] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, volume 46, pages 164–174. ACM, 2011.

[109] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. Approximate storage in solid-state memories. *ACM Transactions on Computer Systems (TOCS)*, 32(3):9, 2014.

[110] Samsung. 16gb f-die nand flash multi-level-cell (2bit/cell). 2011.

[111] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. Xoring elephants: Novel erasure codes for big data. In *Proceedings of the VLDB Endowment*, volume 6, pages 325–336. VLDB Endowment, 2013.

[112] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, et al. Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization. In *Microarchitecture (MICRO), 2013 46th Annual IEEE/ACM International Symposium on*, pages 185–197. IEEE, 2013.

[113] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 14–26. IEEE Press, 2016.

[114] Zhirong Shen, Jiwu Shu, Patrick PC Lee, and Yingxun Fu. Seek-efficient i/o optimization in single failure recovery for xor-coded storage systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(3):877–890, 2017.

[115] Hojun Shim, Yongsoo Joo, Yongseok Choi, Hyung Gyu Lee, and Naehyuck Chang. Low-energy off-chip sdram memory systems for embedded applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 2(1):98–130, 2003.

[116] Majid Shoushtari, Abbas BanaiyanMofrad, and Nikil Dutt. Exploiting partially-forgetful memories for approximate computing. *IEEE Embedded Systems Letters*, 7(1):19–22, 2015.

[117] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. Ieee, 2010.

[118] Peter Sobe and Peter Schumann. A perfomance study of parallel cauchy reed/solomon coding. In *Architecture of Computing Systems (ARCS), 2014 Workshop Proceedings*. VDE, 2014.

[119] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. Pipelayer: A pipelined reram-based accelerator for deep learning. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 541–552. IEEE, 2017.

[120] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Graphr: Accelerating graph processing using reram. *arXiv preprint arXiv:1708.06248*, 2017.

[121] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending ssd lifetimes with disk-based write caches. In *FAST*, volume 10, pages 101–114, 2010.

[122] Daniel Stodolsky, Garth Gibson, and Mark Holland. Parity logging overcoming the small write problem in redundant disk arrays. In *ACM SIGARCH Computer Architecture News*, volume 21, pages 64–75. ACM, 1993.

[123] Pure Storage. Everything in one all-flash array.

[124] Hairong Sun, Pete Grayson, and Bob Wood. Quantifying reliability of solid-state storage from multiple aspects. *Proceedings of the Storageconference.*, 2011.

[125] Yuliang Sun, Yu Wang, and Huazhong Yang. Energy-efficient sql query exploiting rram-based process-in-memory structure. In *Non-Volatile Memory Systems and Applications Symposium (NVMSA), 2017 IEEE 6th*, pages 1–6. IEEE, 2017.

[126] Hitachi Data Systems. Hitachi virtual storage platform g series. Technical report, 2018.

[127] Devesh Tiwari, Simona Boboila, Sudharshan S Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. Active flash: towards energy-efficient, in-situ data analytics on extreme-scale machines. In *FAST*, pages 119–132, 2013.

[128] Peter Trifonov. Low-complexity implementation of raid based on reed-solomon codes. *ACM Transactions on Storage (TOS)*, 11(1):1, 2015.

[129] Dmitriy Vatolin, Alexey Moskvin, Oleg Petrov, and Nicolay Trunichkin. Msu video quality measurement tool, 2009.

[130] Zdeněk Vavřín. Confluent cauchy and cauchy-vandermonde matrices. *Linear algebra and its applications*, pages 271–293, 1997.

[131] Akshat Verma, Ricardo Koller, Luis Useche, and Raju Rangaswami. Srcmap: energy proportional storage using dynamic consolidation. In *Proceedings of the 8th USENIX conference on File and Storage Technologies*. USENIX Association, 2010.

[132] Wei-Lin Wang, Tseng-Yi Chen, Yuan-Hao Chang, Hsin-Wen Wei, and Wei-Kuan Shi-h. Minimizing write amplification to enhance lifetime of large-page flash-memory storage devices. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[133] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 11. ACM, 2016.

[134] 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. *IEEE Transactions on Multi-Scale Computing Systems*, 2(1):17–29, 2016.

[135] Debao Wei, Libao Deng, Liyan Qiao, Peng Zhang, and Xiyuan Peng. Peva: A page endurance variance aware strategy for the lifetime extension of nand flash.

IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 24(5):1749–1760, 2016.

[136] H-S Philip Wong, Heng-Yuan Lee, Shimeng Yu, Yu-Sheng Chen, Yi Wu, Pang-Shiu Chen, Byoungil Lee, Frederick T Chen, and Ming-Jinn Tsai. Metal–oxide rram. *Proceedings of the IEEE*, 100(6):1951–1970, 2012.

[137] Liping Xiang, Yinlong Xu, John Lui, and Qian Chang. Optimal recovery of single disk failure in rdp code storage systems. *ACM SIGMETRICS Performance Evaluation Review*, 38(1):119–130, 2010.

[138] Cong Xu, Pai-Yu Chen, Dimin Niu, Yang Zheng, Shimeng Yu, and Yuan Xie. Architecting 3d vertical resistive memory for next-generation storage systems. In *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, pages 55–62. IEEE Press, 2014.

[139] Cong Xu, Xiangyu Dong, Norman P Jouppi, and Yuan Xie. Design implications of memristor-based rram cross-point structures. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6. IEEE, 2011.

[140] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. Overcoming the challenges of crossbar resistive memory architectures. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 476–488. IEEE, 2015.

[141] Cong Xu, Dimin Niu, Naveen Muralimanohar, Norman P Jouppi, and Yuan Xie. Understanding the trade-offs in multi-level cell reram memory design. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–6. IEEE, 2013.

[142] Lihao Xu and Jehoshua Bruck. X-code: Mds array codes with optimal encoding. *IEEE Transactions on Information Theory*, 45(1):272–276, 1999.

[143] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L Greathouse, Lifan Xu, and Michael Ignatowski. Top-pim: throughput-oriented programmable pro-

cessing in memory. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 85–98. ACM, 2014.

[144] Guangyan Zhang, Guiyong Wu, Shupeng Wang, Jiwu Shu, Weimin Zheng, and Keqin Li. Caco: An efficient cauchy coding approach for cloud storage systems. *IEEE Transactions on Computers*, 65(2):435–447, 2016.

[145] Hang Zhang, Nong Xiao, Fang Liu, and Zhiguang Chen. Leader: Accelerating reram-based main memory by leveraging access latency discrepancy in crossbar arrays. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pages 756–761. IEEE, 2016.

[146] Jialiang Zhang, Soroosh Khoram, and Jing Li. Boosting the performance of fpga-based graph processor using hybrid memory cube: A case for breadth first search. In *FPGA*, pages 207–216, 2017.

[147] Jianlong Zhong and Bingsheng He. Medusa: Simplified graph processing on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, 2014.

[148] Qiuling Zhu, Berkin Akin, H Ekin Sumbul, Fazle Sadi, James C Hoe, Larry Pileggi, and Franz Franchetti. A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing. In *3D Systems Integration Conference (3DIC), 2013 IEEE International*, pages 1–7. IEEE, 2013.

[149] Qiuling Zhu, Tobias Graf, H Ekin Sumbul, Larry Pileggi, and Franz Franchetti. Accelerating sparse matrix-matrix multiplication with 3d-stacked logic-in-memory hardware. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6. IEEE, 2013.