# Real Time and Performance Management Techniques in SSD Storage Systems

Thesis submitted for the degree of

Doctor of Philosophy

at the University of Leicester

by

Muhammed Ziya Komsul

Department of Engineering

University of Leicester, UK

July, 2016

# Abstract

Flash-based storage systems offer high density, robustness, and reliability for embedded applications; however the physical nature of flash memory means that there are limitations to its usage in high reliability applications. To increase the reliability of flash-based storage systems, several RAID mechanisms have been proposed. However, these mechanisms permit the recovery of data onto a new replacement device when a particular device in the array reaches its endurance limit and they need regular garbage collection to efficiently manage free resources. These present concerns with response time as when a garbage collector or a device replacement is underway, the flash memory cannot be used by the application layer for an uncertain period of time. This non-determinism in terms of response time is problematic in high reliability systems that require real-time guarantees. Existing solutions to garbage collection only consider single flash chip but ignore architectures where multiple flash memories are used in a storage system such as RAID. Traditional replacement mechanisms based on magnetic storage mediums do not suit specifications of flash memory. The aim of this thesis is to improve the reliability of the SSD RAID mechanisms by providing guaranteed access time for hard real-time embedded applications.

Investigating the hypothesis, a number of novel mechanisms were proposed with the goal of enhancing data reliability in an SSD array. Two novel mechanisms solve the non-determinism problem caused by garbage collection without disturbing the reliability mechanism unlike existing techniques. The third mechanism is device replacement techniques for replacing elements in the array, increasing system dependability by providing continuous system availability with higher I/O performance for hard real-time embedded applications.

A global flash translation layer with novel garbage collection mechanisms, on-line device replacement techniques, and their associated controllers are implemented on our FPGA SSD RAID controller. Contrary to traditional approaches, a dynamic preemptive cleaning mechanism adopts a dynamic cleaning feature which does not disturb the reliability mechanism. In addition to this the garbage collection aware RAID mechanism is introduced to improve the maximum response time of the system further. On-line device replacement techniques address limitations of the device replacement and thus provide more deterministic response times. The reliability, real-time and performance of these mechanisms via trace-driven simulator for number of synthetic and realistic traces are also evaluated.

The contribution of this thesis is as follows: the presentation of novel mechanisms that enable the real-time support for RAID techniques in SSD devices, the development of a number of mechanisms that enhance the performance and reliability of flash-based storages, the implementation of these controllers, and the provision of a complete test bed for investigating these behaviours.

## Acknowledgements

Firstly, I would like to express my sincere appreciation to my supervisor, Alistair McEwan, for guiding me throughout this project, for encouraging my research, for his patience, motivation, and enormous knowledge. His valuable suggestions to this study helped me in all the time of research and writing of this thesis. Without his valuable suggestions the research work would have not been possible.

I am also thankful to Turkey Minister of National Education for funding and supporting my research.

I also thank my colleagues : Irfan Mir, Muhammad Fayyaz, Jing and Ioannis for the stimulating discussions and precious comments on my research.

Last but not the least, I would like to thank with love to my wife Hatice for standing beside me throughout my career with love and my little girl Zeynep Kubra for being such a good little baby during my study. I would also say thank to my mother, mother-in-law, father-in-law and to my brothers and sisters for supporting me spiritually throughout the Ph.D. research.

## Declaration of Authorship

I hereby declare that this thesis is my own work and that has completed during the period of registration. To the best of my knowledge, it does not contain previously published material written by another person. None of this work has been submitted for another degree at the University of Leicester or any other University.

Some parts of this thesis appeared in the following conjoint publications where I have made substantial contributions:

- A. A. McEwan and M. Z. Komsul, Reliability and Performance Enhancements for SSD RAID, Microprocessors and Microsystems, Available online 18 November 2016, ISSN 0141-9331, http://dx.doi.org/10.1016/j.micpro.2016.11.012.
- A. A. McEwan and M. Z. Komsul, "Pre-Emptive Garbage Collection for SSD RAID," 2016 Euromicro Conference on Digital System Design (DSD), Limassol, IEEE, Aug 2016, pp. 356-363.
- M. Z. Komsul, A. A. McEwan and I. Mir, "A real-time hot swapping technique for SSD RAID systems," 2016 International Conference on Applied System Innovation (ICASI), Okinawa, IEEE, 2016, pp. 1-4.
- A.A. McEwan and M.Z. Komsul." On-Line Device Replacement Techniques for SSD RAID". In Digital System Design (DSD), 2015 Euromicro Conference on, Funchal, IEEE, Aug 2015, pp. 438-444.
- M.Z. Komsul, A.A. McEwan, and I.F. Mir. "An FPGA-based Development Platform for Real-time Solid State Devices". In Information Science, Electronics and Electrical Engineering (ISEEE), 2014 International Conference on, volume 2, Sapporo, IEEE, April 2014 pp. 1198-1203.

# Contents

# List of Tables

# List of Figures

# List of Abbreviations

| | |
|---|---|
| **API** | Application Programming Interface |
| **BER** | Bit Error Rates |
| **Diff-RAID** | Differential RAID |
| **DRAM** | Dynamic Random Access Memory |
| **DRT** | Device Reconstruction Task |
| **DiskSim** | Disk Simulator |
| **ECC** | Error Correction Code |
| **EEPROM** | Electrically Erasable Programmable Read-Only Memory |
| **FTL** | Flash Translation Layer |
| **FPGA** | Field Programmable Gate Array |
| **GARM** | Garbage collection Aware RAID Mechanism |
| **GC** | Garbage Collection |
| **GFTL** | Guaranteed Flash Translation Layer |
| **HDD** | Hard Disk Drive |
| **I/O** | Input/Output |
| **LBN** | Logical Block Number |
| **LPN** | Logical Page Number |
| **MLC** | Multi Level Cell |

| | |
|---|---|
| **NvSRAM** | Non-volatile Static Random Access Memory |
| **OPMM** | On-line Parity Migration Mechanism |
| **PC** | Personal Computer |
| **PBN** | Physical Block Number |
| **PGC** | Pre-emptive Garbage Collection |
| **PPN** | Physical Page Number |
| **RAID** | Redundant Array of Independent Disk |
| **RAM** | Random Access Memory |
| **RFTL** | Real-time Flash Translation Layer |
| **RTL** | Register Transfer Level |
| **SLC** | Single Level Cell |
| **SRAM** | Static Random Access Memory |
| **SSD** | Solid State Drive |
| **TLC** | Triple Level Cell |
| **UBER** | Uncorrectable Bit Error Rate |
| **WCET** | Worst Case Execution Time |

# List of Symbols

| | |
|---|---|
| $t_{read}$ | time taken to read a page |
| $t_{write}$ | time taken to write a page |
| $t_{copy}$ | time taken to copy a page |
| $t_{er}$ | time taken to erase a block |
| $GC_{cost}$ | time taken to reclaim a block |
| $B_{valid}$ | number of valid pages in a block |
| $U(e_r)$ | the maximum execution time for read |
| $U(e_w)$ | the maximum execution time for write |
| $N_{free}$ | number of free blocks in the flash chip |
| $T_{soft}$ | soft threshold level for Preemptible Garbage Collection |
| $T_{hard}$ | hard threshold level for Preemptible Garbage Collection |
| $Th(x, y)$ | threshold ratios for a given device (x=$T_{soft}$, y=$T_{hard}$) |
| $n$ | the number of devices in the array |
| $m$ | stripe unit size of an array |
| $GC_D$ | dynamic garbage collection |
| $P_H$ | high priority garbage collection |
| $P_L$ | low priority garbage collection |
| $AGE_C$ | current ageing ratio of a device |

| | |
|---|---|
| $AGE_O$ | optimal ageing ratio of a device |
| $FBP$ | free block percentage of a device |
| $P_L^T$ | garbage collection triggering threshold for $P_L$ |
| $P_H^T$ | garbage collection triggering threshold for $P_H$ |
| $P_M^T$ | medium threshold level |
| $GCE$ | garbage collection efficiency function |
| $P_i$ | parity percentage of $i^{th}$ device |
| $h$ | $P_H$ factor |
| $p$ | length of period to trigger $GC_D$ |
| $D_i$ | $i^{th}$ stripe unit of data D |
| $D_P$ | parity data of stripe D |
| $GC_{pool}$ | devices those number of free blocks below the soft threshold |
| $n_p$ | number of devices that have number of free blocks below the hard threshold |
| $h$ | $P_H$ factor |
| $t'_{Dx}$ | access time of writing Data x |
| $F_{TH}$ | hot-swapping indicator flag |
| $F_{HB}$ | semi-hybrid RAID indicator flag |
| $F_{DR}$ | device replacement completion indicator flag |

# Chapter 1

# Introduction

An embedded system is a computing system that consists of hardware and software components. It is designed to perform dedicated functions within a larger electrical system [1]. A core inside a processor is the heart of an embedded system. The word size of the processor core can vary from 8-bit to 64-bit. Microwave ovens, cell phones, calculators, digital watches, cruise missiles, heart monitors, laser printers, radar guns, washing machines, digital cameras, traffic lights, remote controls, bread machines, and fax machines are some examples of embedded systems. A modern car is an example of a large embedded system which comprises a number of embedded controllers with dedicated functions. In such high-integrity embedded systems, requirements are getting more complex with time and resource constraints.

In general purpose computing systems, there are mainly two permanent storage types — Hard Disk Drive (HDD) and flash. HDDs are usually preferred as primary storages in Personal Computers (PCs) and servers. High-integrity embedded systems have strict constraints on storage mediums such as shock-resistance, energy consumption and size. Disks are not well suited to these constraints as they include mechanical parts. In comparison to HDD, Solid State Drives (SSDs) have no moving parts, consume less energy, are more robust against mechanical failures in the presence of

shocks/jerks and require less memory access time. In addition, the gradual decrease in flash memory price has increased the demand for use in embedded systems [2, 3]. The rationale behind the decrease cost is the use of higher density flash technology such as Multi Level Cell (MLC) and Triple-Level Cell (TLC) which provide increased capacity compared to the Single Level Cell (SLC). Due to SSD's several advantages, it is used as a primary massive storage medium in many embedded systems. Examples of these massive storage embedded systems are smart phones, tablets, digital cameras, and space applications [4–7].

While flash offers many advantages, a number of concerns still exist in flash based technology. These concerns affect its reliability and performance such as endurance limit, non-deterministic access time, and a higher number of bit errors in aged flash devices. Unlike HDDs, the lifespan of a flash cell depends on the amount of erase/write operations as these slowly wear out flash cells. The wear-out of flash cells causes data corruption when a cell exceeds its lifetime/endurance limit [8, 9]. To overcome the problem of bit-error rate, NAND[1] flash memory uses Error Correcting Codes (ECCs). However, ECCs are limited to single-bit error correction in SLC based flash memories. ECCs are insufficient for MLC and TLC devices and are not applicable for component failures due to the limited size of the metadata area (a place where the ECCs are usually stored) of a page. Conventionally, Redundant Array of Independent Disk (RAID) systems have been used to provide protection against the chip, block, and page level failures [10]. However, the RAID system is not directly applicable to flash arrays because of the risk of simultaneously wearing out of array elements [8].

To address this problem, several flash based RAID mechanisms have been proposed which enhance reliability of the system by protecting the flash memory against the chip-level failures in high integrity systems [8, 11]. These RAID techniques offer significant improvements to the reliability of flash storage systems but it does not meet

[1] A flash memory has two types - NOR and NAND. Both types are discussed in Chapter 2

the real-time requirements of embedded systems. These developed mechanisms require a regular garbage collection (GC) (which is managed by a flash translation layer (FTL)) to efficiently manage free resources. In addition, these mechanisms also require a device replacement process, when a particular element in an array reaches its endurance limit. Due to garbage collection and device replacement processes, embedded software application cannot access flash memory for an uncertain amount of time. This non-deterministic response time is not acceptable for highly reliable embedded systems where guaranteed response time is essential.

Current and future space applications demands flash based storage system in order to enhance the mission return. One such application of the flash based storage system is the spirit rover [12] which demands a real-time and reliable storage system. Due to the limited connectivity with the ground station and electrical power outages, it is required to store science data on-board the rover. Contrary to Random Access Memory (RAM) based storage system, data is stored in non-volatile memory which requires a lesser amount of electrical power and can be download at any time. Furthermore, running tasks demand to store data in real-time. A delay in the data storage/retrieval may cause a task to miss its deadline. This behaviour consequently affects the overall mission return. In this thesis, a novel flash based RAID storage architecture with real-time support is proposed, designed and developed. This mechanism supports the reliability as well as stringent real-time requirements of the high integrity embedded systems. It provides guaranteed access time to the application layer in case of ongoing garbage collection and device replacement process, thus improving system throughput.

The above mechanism is designed for use in high integrity systems but in general, it can be used for any embedded system. For example, a number of mobile multimedia applications that require guaranteed access time and high performance can benefit from the proposed approach such as mobile data compression applications. These applications play an important role in mobile multimedia as they reduce the performance

overhead of the data transfer as well as reducing the storage capacity. It is highly important for these types of applications to have high performance storage with real-time support [13]. In addition, another example can be given as large-scale sensor network applications such as the TinyDB project [14]. It includes a number of nodes in a network which are able to transmit and store sensor data. Transmitted data have to be stored in a storage with a real-time fashion where deterministic response time and continued storage availability are necessary.

## 1.1 Motivation

Although flash-based storage systems offer a number of advantages for embedded applications; flash memory has limitations to its usage in high reliability applications as consequences of garbage collection (GC) and limited lifetime of flash cells (which requires device replacement). These cause non-deterministic response times and performance degradation which are problems for high reliable systems that require real-time access guarantees.

The motivation behind this research is threefold:

**I.** Demand of highly reliable and real-time NAND flash-based SSD storage systems is continuously increasing. Examples of these applications are aircraft, spacecraft, autonomous vehicles and robotic systems.

**II.** Traditional real-time garbage collections are limited to a single memory and are not compatible with the SSD RAID techniques [15–18]. Currently, in SSD RAID systems, no solution exists which can address the problem of deterministic response time, while simultaneously maintaining its high reliability feature.

**III.** Existing solutions for device replacement mechanisms are limited to magnetic devices and ignores inherent features of flash memory such as deterministic failure and garbage collection [19–21].

## 1.2 Problem Statement

In this thesis, two main problems of flash-based RAID storage systems are addressed.

- Firstly, the non-deterministic response time issue caused by garbage collection in parity-based RAID is addressed.
- Secondly, the off-line device replacement problem in reliability enhancement mechanisms along with RAID techniques is considered.

  These research problems are further elaborated in Section 1.2.1.

### 1.2.1 Research Questions

The main research questions of the thesis are listed below:

**I.** Can the existing real-time garbage collection mechanisms for single flash memory adapt to the RAID techniques where the devices in the array are strictly controlled by the reliability mechanisms?

**II.** How can a real-time GC mechanism be developed without (or minimally) disturbing reliability mechanisms in the RAID array?

**III.** How can the downtime of flash-based RAID storages be improved further with the real-time GC mechanism?

**IV.** Can the Worst Case Execution Time (WCET) of the real-time GC mechanisms be reduced further by utilizing the concurrency of parity-based RAID architectures?

**V.** How can unpredictable GC delays in flash based RAID architectures be completely avoided without disturbing the reliability mechanisms?

**VI.** Can the existing HDD-based on-line device replacement techniques be fully adapted to flash-based mechanisms?

**VII.** How can continuous system availability be provided when a device replacement is required for the reliability mechanism?

**VIII.** How can the performance overhead of device replacement techniques be reduced in SSD RAID?

**IX.** How can the experimental results be tested and validated?

## 1.3 Scope and Objectives

The scope of the thesis is limited to the algorithms and data structures of a flash file system for high integrity embedded system where limited system resources are available. Embedded and raw NAND flash based SSDs are mainly considered throughout this thesis.

The main objectives of this research are:

- To assess existing approaches/methods/architectures for SSDs through investigation of the current state of the art.
- To propose a suitable reliable and real-time mechanism for SSDs to use on-board high integrity embedded systems.
- To analyse, evaluate, test and demonstrate the proposed mechanism through simulation.

## 1.4 Methodology

The research has started by exploring existing real-time techniques in flash memory architectures. A number of real-time techniques were examined in terms of their adaptation to the existing reliability mechanisms. Then their limitations were determined, and their pros and cons were compared. As a result of these reviews, it was revealed that the existing mechanisms were not fully efficient and unable to meet the real-time, performance and reliability requirements. Thus, a novel real-time and reliable SSD storage architecture was designed and implemented.

The research work was based on the spiral method [22]. The model provides a high amount of risk analysis. In the spiral model it is important to split the research work into different levels to minimise the risk of failure.

The work in this thesis follows several iterations of the predefined levels of the spiral model. The spiral methodology has four main phases: (I) planning phase; where all requirements of the research are gathered, (II) risk analyses phase; where a process is undertaken to identify risk and alternative solutions, (III) engineering phase; which includes development, implementation and testing, and finally (IV) evaluation phase; which includes review of the process and the next iteration plan. The main tasks are described below to achieve the research goals of this thesis:

**I.** Identify the research objects.

**II.** Develop the real-time and performance enhancement mechanisms.

**III.** Implement these mechanisms into software design.

**IV.** Develop a debug environment to simulate software design.

**V.** Analyse the simulation results.

## 1.5 Thesis Contribution

The limitations and features of flash based storages — which were discussed previously — invoke interest on researching flash technologies and their different aspects. The bottlenecks of the technology limit its usage in many application areas especially due to its real-time support issues. The contributions of this thesis are listed below:

**I.** A novel real-time *dynamic pre-emptive garbage collection* technique for SSD RAID is proposed with two novel components including *dynamic garbage collection* and *pre-emptive mode controller*. It does not only provide real-time support with higher performance to the storage but also maintains the age distribution ratios of reliability mechanisms.

**II.** A novel *garbage collection aware RAID mechanism* is proposed. It further improves the WCET and the performance of the system compared to the published techniques. Also an *on-line parity migration* feature is integrated to maintain reliability mechanisms for the *garbage collection aware RAID mechanism*.

**III.** A novel on-line device replacement framework for SSD RAID including four novel techniques— *proactive hot swapping, coordinated data migration, on-line parity redistribution*, and *semi hybrid RAID*. These techniques can replace aged devices in the array in such a way that provides continuous system availability thereby providing deterministic response time. Also the performance overhead of the RAID reconstruction process is reduced.

**IV.** The proposed mechanisms and techniques are validated by developing a test-bed for a reliable and real-time SSD-based storage system including *RAID controller, Global FTL layer,* and *dynamic page allocation*. It provides accurate timing of Inputs/Outputs (I/Os) requests to the NAND flash memory and enables to test correctness of the proposed mechanisms.

## 1.6 Thesis Overview

The structure of the thesis is carefully organized from its motivational part to the final research outcome. In this chapter a brief introduction to the subject matter, motivations behind this thesis, a research aim considering existing research challenges, and the contribution of the research are presented. The rest of the chapters are partitioned into three main parts— (1) background study and literature review, (2) contribution of this thesis to the specific research questions and (3) conclusion and future work.

The next two chapters present the background information and previous work, especially studies that address challenges in the area of reliable SSD storage systems in hard real-time applications. Chapter 2 presents the basics of flash memory, flash translation layer and its main components. It introduces parity RAID systems and some important phenomena about them such as write types. It also presents basics of the real-time systems and importance of the deterministic response time. The current challenges and trends in flash memory architectures are presented in Chapter 3. It presents real-time support, performance and reliability issues of flash based storages with respect to high integrity systems. This chapter discusses the limitations of the current solutions and highlights the research gap.

The following four chapters present the contribution of this thesis in the area of real-time support and performance of NAND flash-based SSD storage systems. Chapter 4 presents a dynamic pre-emptive GC mechanism for an existing real-time GC to provide real-time access guarantees without ignoring the strict age control mechanism of SSD RAID via the dynamic garbage collection component of the mechanism. Also, another component of the mechanism — pre-emptive mode controller — is introduced to coordinate ongoing GC operations with a holistic view of the system. It improves the real-time efficiency and performance of the system by globally coordinating the state of each garbage collector in the array. Although the mechanism provides real-time

access guarantees for SSD RAID, the enhancement in the WCET and performance of the system are limited due to not benefiting from the concurrent system architecture effectively. Therefore, a novel real-time GC mechanism for SSD RAID systems is introduced in following chapter (i.e. Chapter 5).

Chapter 5 presents a novel GC-aware RAID mechanism (GARM) which further improves the WCET and performance of the system considering the wearing out problem of flash memory. It provides deterministic response times for I/Os even where there is an active GC operation in the array with the help of its serialised garbage collection feature. The mechanism is evaluated under various types of scenarios such as partial and full stripe writes and read operations in existence of serialised GC. Moreover, the effects of the GARM on the reliability enhancement mechanisms are investigated. An on-line parity migration feature is integrated in the GARM to keep ageing ratios of elements which are affected by initially assigned parity percentages. It aims to maintain the reliability enhancements techniques while offering deterministic response time with a dynamic page allocation approach.

Chapter 4 and 5 focus on GC as a non-deterministic problem of SSD RAID architectures. However, the device replacement operation in SSD RAID systems prevents the whole product from being suitable for real-time system due to its non-deterministic behaviour. Therefore, in the following chapter (i.e. Chapter 6) this problem is addressed to make the system fully compatible for real-time applications. Chapter 6 presents an SSD RAID framework incorporating several novel techniques to improve the efficiency of the replacement process for hard real-time applications — a proactive hot swapping technique, a data migration technique that coordinates operations with the garbage collector, and a parity redistribution mechanism. To utilize the benefits of hot swapping, a semi hybrid RAID mechanism is also introduced that enhances performance when there is no active device replacement process.

The development of test-bed for a reliable and real-time SSD-based storage system is presented in Chapter 7. For accurate timing of disk requests a NAND flash SSD simulator is designed as an extension of the Microsoft SSD simulator. Global Flash Translation Layer (FTL) layer, dynamic page allocation schema, the RAID controller and address mapping techniques are implemented on the test bed. Also, results of experiments are presented to support the mechanisms described in previous chapters along with a number of synthetic and realistic (captured) traces.

Finally, Chapter 8 discusses the final outcomes of this thesis along with the next target of the study. The results are analysed against the objectives and the research challenges which are explained in Chapter 1. Finally, the possible future directions are pointed to exploit the outcomes of this thesis.

# Chapter 2

# NAND Flash-based Storage Systems

This chapter presents a background study of flash-based storage systems. NAND flash-based storages—also known as SSD—are a non-volatile type of memory. Flash memory has a number of properties that make it a useful storage medium for high reliability systems when compared to conventional storage devices such as HDDs. These properties include higher performance, low power consumption, shock resistance, and small physical size. Despite these issues, HDDs have been used as primary devices in PCs and laptops due to their low cost and high storage capacity.

This chapter is organized as follows: Section 2.1 presents the basics and overall structure of flash memory. Next, the structure and main components of the flash transition layer are discussed in Section 2.2. Design architecture and current research aspects of flash-based SSD architectures are described in Section 2.3. After introducing parity-based RAID mechanisms in Section 2.4, the characteristics of workload in terms of random and sequential access are described in Section 2.5. Section 2.6 defines the basics of real-time systems. Finally, a summary is given in Section 2.7.

## 2.1 Overview of Flash Memory

Flash memory is a type of Electrically Erasable Programmable Read-Only Memory (EEPROM). It retains data even when there is no power. It consists of a number of transistors with an oxide layer located among them. Flash memory stores data in a way that values of the cells in the oxide layer either change from high (1) to low (0), or vice versa, once a datum is loaded into it.

There are two distinct types of flash memory—NOR and NAND. NOR-based flash memories benefit from fast random reads, but suffer from slow write and erase operations. Due to its fast random access feature, they are commonly used for holding live data such as for code execution. On the other hand, NAND-based flash memories are more suitable for use as medium or long-term storage in embedded systems because of their fast write and higher storage density features. Only NAND-type flash memory is considered in this thesis.

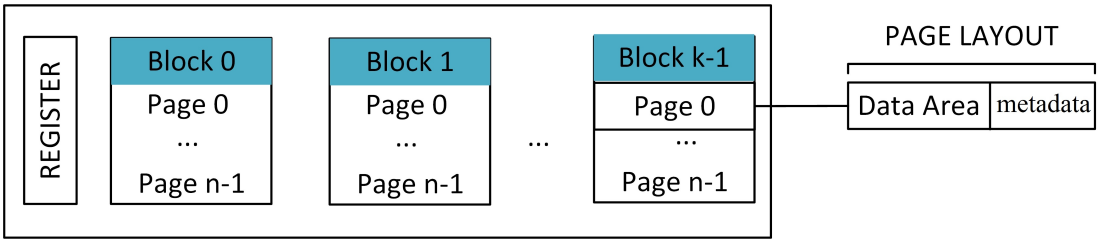



Figure 2.1: Internal structure of flash memory

Flash memory consists of multiple blocks, with each block containing of certain number of pages, as shown in Figure 2.1. A page is logically divided into two parts: a data area, for storing actual data, and a metadata area, for keeping metadata of the related page such as ECCs, logical block address, logical page address, and page status flag (valid, invalid or free). The size of the data area varies between 512-4096 bytes, while the size of the metadata varies between 16-128 bytes.

The granularity of the erase and read/write operations differ from each other in flash memory. The erase operation must be performed at the block level but the read and write operations are performed at the page level. Writing a page size datum from a register to a physical flash page takes typically 200-700 $\mu$sec. However, erase operations are considerably slower compared to write operations, and typically take around 1.5-3.0 msec as they involve erasing the whole block. The fastest operation in flash memory is the read operation which takes around 20-50 $\mu$sec. Due to the physical features of flash memory, each flash cell has a limited number of erase operations, typically between 10,000 and 100,000.

## 2.2 Flash Translation Layer

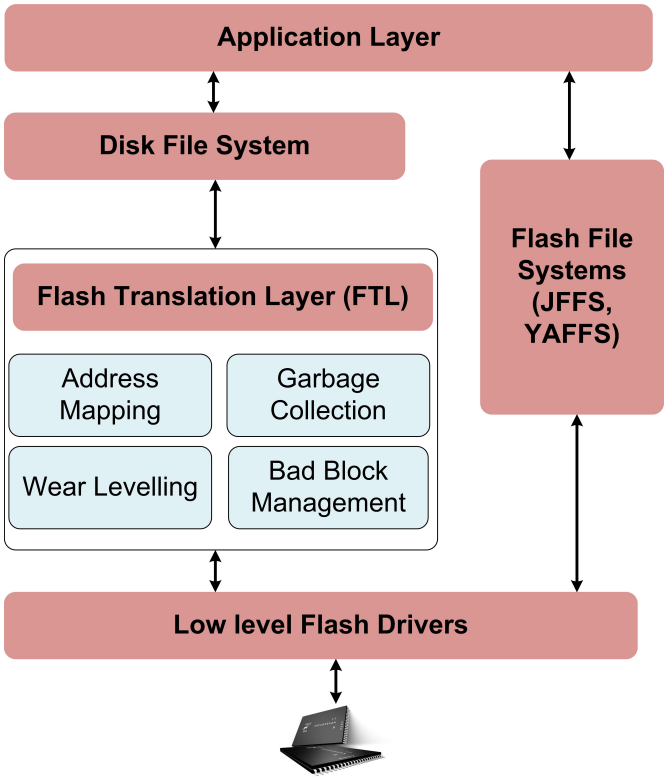



Figure 2.2: Architectural system design of flash memory

Due to flash memory's inherited features, special algorithms and data structures are required to manage I/O operations over flash memory. An architectural design for flash memory is shown in Figure 2.2 as a block diagram.

Given I/Os from the application layer, it can not directly communicate with raw flash memory. There are two ways of adapting flash memories: first, the FTL [23] and, second, flash-specific file systems such as JFFS [24] and YAFFS [25].

Flash devices can be used as normal block devices by eliminating their complicated internal structure with the help of an FTL [23]. Unlike HDDs and volatile memories (such as Dynamic RAM (DRAM)), flash memory-based storage has the erase-before-write feature. Blocks cannot be overwritten and must be erased before they can be re-used. A flash page can be in one of three different states including valid, invalid and free. If there is no data written into a page, then it is considered to be a free page. A write operation to a free page changes its state to valid. Flash memories do not perform update-in-place as in HDDs; instead, the previous page is marked as invalid and the new data is written to a new physical page location. To keep changes between logical and physical addresses, an address mapping table needs to be managed. Also, invalid pages must be reclaimed when the system suffers from a lack of free space.

These operations are handled by the FTL layer. FTL performs four different principle tasks including garbage collection to reclaim invalid pages, address mapping to keep the link between the logical and physical addresses, wear levelling for extending the life of the flash memory, and bad block management to manage expired blocks. These components will be detailed in following sections.

### 2.2.1 Address Mapping

The address mapping algorithms of the FTL are divided into three categories: page level, block level, and hybrid mapping. The techniques retain a table that keeps the

changes between the logical page number (LPN) and physical page number (PPN). The table is usually stored in two different locations: Static RAM (SRAM) memory for faster access, and the metadata area of each physical page to retain the table when there is no power. Once the file system starts up, the table in the dynamic memory is constructed by reading all the metadata areas of each page.

The page level mapping technique has been widely used in number of FTL designs due to the reduced complexity of its structure and faster access times [18, 26]. Once a read operation is given from the host with an LPN, the FTL finds its corresponding physical location by checking the address table stored in SRAM. If an update request arrives at the storage system then the mapping table first invalidates the previous physical address and assigns a new PPN for the request.

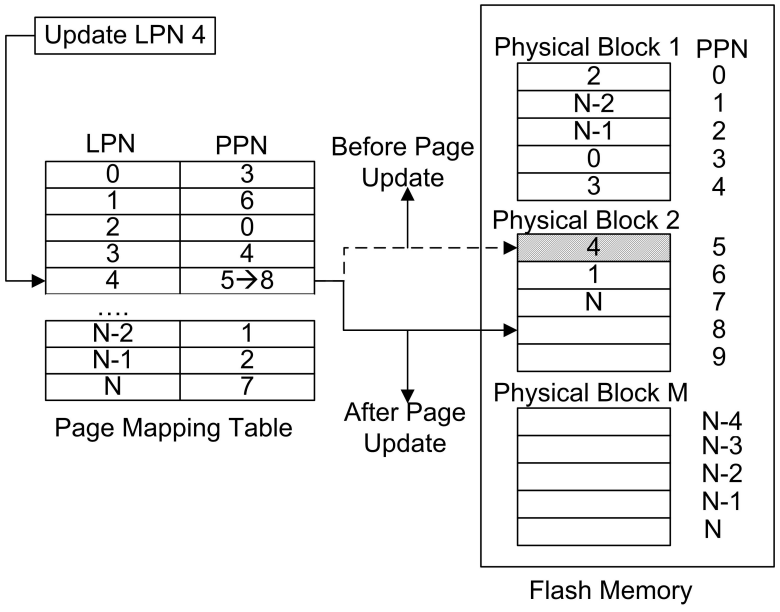



Figure 2.3: Page level address mapping

Figure 2.3 illustrates an example of an update operation in a page level address mapping algorithm. An update instruction is given to the FTL with LPN 4. First, the FTL finds its associated PPN (5) by checking the mapping table. Second, if the LPN

is already linked with an PPN, then it locates a new PPN (8) for the LPN, thereby updating the mapping table and invalidating the previous PPN (5).

Although the page level mapping offers a faster and easier way to keep changes in the PPN, it suffers from a large memory overhead due to the size of the mapping table. The size of the mapping table increases with the capacity of the flash memory, which requires a bigger size SRAM. To address this problem, block level address mapping techniques have been used in number of studies [27, 28].

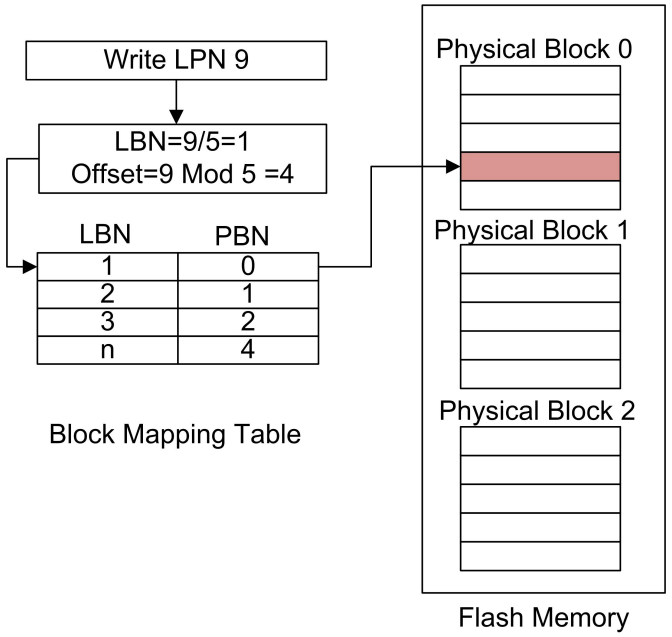



Figure 2.4: Block level address mapping

An example of a write operation in block level mapping is illustrated in Figure 2.4. Each logical block number (LBN) is linked to a physical block number (LBN). If a write request is assigned to the FTL with an LPN (9), the FTL calculates its LBN and offset, which defines the page location in the corresponding PBN. It then finds the physical block location and target page within the block with the help of the offset.

Although block level mapping reduces the size of the mapping table, it causes slower access and requires more complex architecture compared to page level mapping.

To address this, hybrid mapping techniques have been studied in the literature [29–31]. The purpose of these approaches is a balanced trade-off between page and block level mapping. Generally, hybrid mapping techniques use block level mapping to find the PBN and page level mapping to find the PPN within the PBN. The LPN is also stored in the metadata area of the corresponding physical page.

To be consistent with the previous reliability enhancement mechanism [11] and to provide simplicity and efficient garbage collection, the presented mechanism is based on the page level mapping schema. It is also noted that the page level address mapping technique is most suitable for efficient garbage collection as it does not require an expensive merge operation between the blocks [32].

### 2.2.2 Garbage Collection

In case of applying an update-in-place policy of magnetic devices on flash memory-based storage, each update operation leads an additional block erase operation, which significantly slows down system performance to an unacceptable level. To overcome this problem, flash memory allows an out-of-place update where the updated data is located to a new (free) place, invalidating the old data. After a period of time, flash memory may suffer from a lack of free space due to invalid pages. To reclaim the invalid spaces, garbage collection needs to be performed by the FTL.

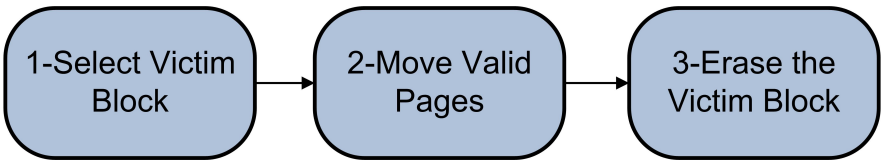



Figure 2.5: Steps of a generic GC

The steps of a typical GC process are illustrated in Figure 2.5. The GC first selects a victim block based on a selection algorithm. Usually the selection of the victim

block is decided by the greedy approach, which chooses the block that has the most number of dirty pages among the flash memory blocks. Second, the GC copies all valid pages from the victim block to a free block and updates the indexing information in the address mapping table according to the new physical locations of the valid pages. Once all pages have been transferred, the block is erased and it becomes a free block.

There are two main types of GC mechanism according to their triggering techniques [33]. The first is idle time-based, where cleaning takes place in the background. This technique significantly reduces the performance overhead of the cleaning process as it is triggered when idle time is detected in the workload. However, the detection of an idle time period requires a complex algorithm, and some workloads do not even have sufficient idle time periods for cleaning. Therefore, threshold-based cleaning was adopted, where cleaning is triggered based on the amount of free space remaining. It has a simpler design architecture and has predominantly been used in existing FTL layers. However, it may cause long and non-deterministic delays as once cleaning is triggered, incoming requests have to wait until it is over. A number of FTL mechanism were proposed to address this performance and non-deterministic behaviour of the flash memory [15–18, 34].

**Garbage Collection in Main Memory**

In computer science, garbage collection is also used for main memory (RAM) management. If objects in the memory are not being used anymore by the corresponding program then the garbage collector reclaims such spaces to provide more free memory space for new programs to be executed [35]. The main difference between flash memory's GC and main memory's GC is that while the erase operation is handled in page level in main memory, it is not possible to reclaim invalid/dirty pages without erasing the whole block in the GC of flash memory. Thus, the time cost of flash memory GC is considerably more expensive than that of main memory.

### 2.2.3 Wear Levelling and Bad Block Management

Flash memory has a limited number or erasure cycles per block of between 10,000 and 100,000, as mentioned earlier. Excessive usage of the same blocks shortens the average life of the flash memory in general. To prevent this, FTL uses wear-levelling to distribute erasure operations evenly across flash memory blocks [23]. The main purpose of the wear-levelling techniques is to distribute total erase operations evenly across the memory block to prolong the lifetime of the flash memory. To handle this, hot/cold data separation/identification techniques are principally applied [36]. Hot data—which are accessed more frequently than the cold data—are placed into less aged blocks while cold data are placed into more aged blocks to balance the age of memory blocks.

Moreover, the FTL can detect the bad blocks which occur due to manufacturing defects or natural wearing out. When a memory block becomes unreliable, the bad block management component of the FTL detects them by applying checksum techniques. After that, these blocks are marked as bad and removed from the mapping table to prevent any access to them.

## 2.3 Flash-based SSD Architecture

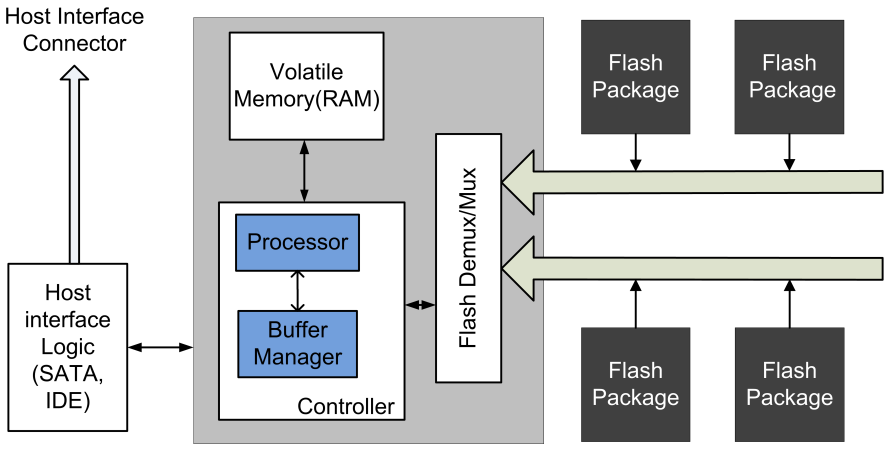



Figure 2.6: SSD internal architecture

Figure 2.6 illustrates the internal architecture of a generic flash-based SSD. It mainly consists of:

**I.** Flash packaging (commonly thought of as "the flash chip") for storing persistent data. Flash packages can be accessed in parallel within an SSD architecture, which improves the SSD performance by improving the throughput;

**II.** A buffer (which can be thought of as a cache) for holding data relevant to recent read and write requests;

**III.** A processor to handle management operations;

**IV.** Volatile memory to support the processor with storage of address mapping tables and other system data;

**V.** A bus, or buses, connecting the flash packaging with the control components.

The FTL is located between the SSD and the host system interface. It may be commonly thought of as the combination of the control software, processor, buffers, and volatile memory, although in practice there is overlap in these components with the SSD. Although a generic architecture was presented here, most off-the-shelf SSD products hide their internal designs for commercial reasons.

The current research trends regarding SSDs usually focus on these aspects: performance, reliability, endurance, capacity and cost. Multichannel parallelism of flash packages is widely used to improve both the system performance and reliability in SSD architectures [37–40]. To improve the storage capacity and reduce the system cost, MLC technology can offer benefits over SLC devices as they offer higher storage densities. However, MLC devices suffer from greater unreliability—particularly when the flash device ages—than SLC due to a lower erase endurance [41, 42].

Overall reliability of MLC flash-based storage systems suffer significantly from high Bit Error Rates (BER). One approach to enhance reliability in the case of BER is

the use of EEC. This is parity-style data that is stored in the metadata of each page of memory, and is used to cross-check, or repair, the data it relates to at the point where data is read. However, there are limits as to how much it can improve reliability as the potential to repair is limited. For this reason it is not ideal to use it as a sole mechanism when reliability is particularly crucial. MLCs also do not lend themselves to ECC techniques as the size of the metadata areas is limited.

Conventionally, RAID systems have been widely used to provide data protection against chip-level failure, and to improve integrity of storage [8, 43–46]. However, RAID can not be directly applied to SSD arrays because of the risk of wearing out individual devices in the array simultaneously [8]. The problem is addressed using a RAID architecture that enhances reliability by protecting against component failures using a load imbalancing technique [8, 11]. This enhances reliability, but ignores the deterministic response time requirements of hard real time systems—due to uncoordinated garbage collection [15] and off-line device replacement [47].

For understanding design parameters better, the building blocks of RAID systems are explained in following section.

## 2.4 Parity-based RAID systems

RAID techniques are applied across multiple drives or chips in order to prevent data loss in case of a disk failure. The data is allowed to use the redundancy inherent to the RAID system. The main aim of RAID is to provide data redundancy and faster access for I/Os over multiple storage devices. There have been several RAID techniques introduced in the literature. RAID 0, for example, stripes the data evenly across multiple devices to enhance system performance but ignores the reliability.

On the other hand, parity-based RAID levels such as RAID 4 and RAID 5 reserve a chip/drive to store redundant data. In case of a device failure, the data can be recovered

with the help of redundant data. In particular, RAID 5 evenly stripes the redundant data across the devices while a device is donated to the store all redundant data in RAID 4.

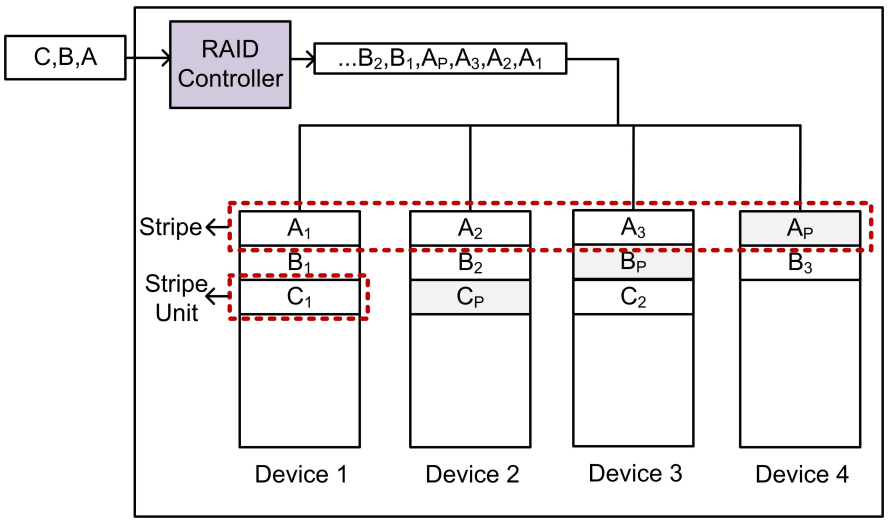



Figure 2.7: RAID 5 structure

RAID 5, for example, can be implemented with multiple devices, as shown in Figure 2.7, but at least three devices are required. Data from the host is stored in the internal buffer of the RAID controller, which then breaks down the data (A) into data stripe units ($A_1$, $A_2$, $A_3$) according to striping granularity and generates a parity stripe unit ($A_P$) by simply exclusive ORing (XORing) the data stripe units. The striping granularity can be at both the block level or page level. The parity knowledge of each stripe is evenly distributed across the devices in the case of RAID 5.

In case of a data failure on Device 1 in Figure 2.7, the RAID rebuilding process triggers the following steps: the controller detects all the logical stripe units for the corresponding stripe and reads them from their locations (Device 2, 3 and 4). Then, by XORing these stripe units, the failed data can be logically reconstructed. After calculating the failed data, new space is allocated—usually within a spare device—and the calculated data is relocated to this new device. This process is repeated for each failed stripe unit until the whole set of failed data is recovered.

## 2.5 Write Types in RAID

Flash-based SSD storage handles I/Os in a quite different way from HDDs. The main reason behind this is that SSDs do not experience seek time when accessing the physical location of the data as it does not contain a moving cluster head. Therefore, there is no performance variety between random and sequential access on a single flash memory. However, HDDs contain a moving cluster head to access the physical location of the data, which makes them slower in the case of random access I/Os to memory.

Although random access to the single flash memory does not produce additional performance overhead for single flash, it is handled in a quite different way in flash based RAID architectures. There are two types of write operation in parity RAID-based systems: *full stripe write* and *partial stripe write*. If data is written across all devices in the array, then it is considered to be *full stripe write*. It is also known as a sequential write in RAID-based storage [8]. It does not require additional read or update operations as the whole stripe is being updated.

However, parity-based RAID systems do not perform well with *partial stripe writes* which are also known as random writes. If the number of stripe units is less than n-1, then it is called as a *partial stripe write*. Any changes in a chunk of stripes requires additional update operations in parity data to the corresponding stripe. To calculate the new parity, some stripe units need to be read; these operations lead to the system suffering performance overhead.

For example, to update data $B_1$ in the stripe illustrated in Figure 2.8, the RAID controller needs to carry out the following steps:

- Read $B_P$ and $B_1$ to calculate the new parity
- Calculate new parity $B'_P$
- and write new data $B'_1$ and updated parity $B'_P$ to the array

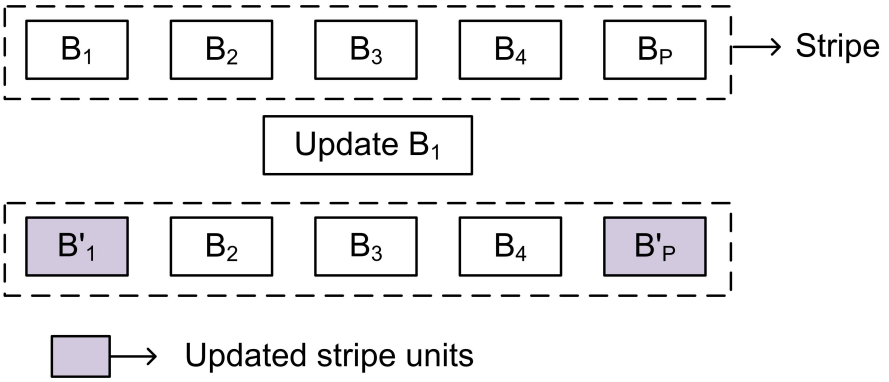



Figure 2.8: Random update operation in parity-based RAID

When any of the data stripe units are updated in a logical stripe, the parity data has to be updated. Therefore, the controller first reads the old data stripe unit B1 and the old parity data $B_P$. Then, the new parity stripe unit $B'_P$ is calculated by XORing them. Finally, the updated parity data $B'_P$ and the updated stripe unit ($B'_1$) are written into their new physical locations. The time cost for update data $B_1$ can be calculated as $(2 \times t_{read}) + (2 \times t_{write})$, where $t_{read}$ and $t_{write}$ give the read and write cost of a page of flash memory, respectively. Because of the parity update overhead of random writes, parity-based RAID techniques show lower performance.

The number of GCs in flash memory is highly affected by the type of workload. For instance, random writes and update operations of small size increase the need for GC significantly [48]. Thus, random write-dominant workloads suffer from significant performance overheads due to triggering the GC more frequently. These types of workload slow down the GC because of the need to copy more valid pages during the GC [18]. There have been a number of techniques proposed to eliminate this performance overhead, such as hot/cold data separations [49, 50]. The garbage collector also blocks all incoming request to the memory for a non-deterministic time. This is problematic for real time systems where guaranteed response times are required. The basics of real-time systems is described in following section.

## 2.6 Real-Time Systems

In a real-time system the execution time of the task is as important as the correctness of the results [51]. Hardware and software architectures must be designed to meet these requirements. The real-time system does not actually mean the system itself offers high performance. The important criterion is to meet the time constraints of the system. Examples of real-time applications can be found in control systems, transaction and multimedia systems such as spaceship control, vehicle simulation, data acquisition and image processing.

Real-time applications handle a variety of tasks in different time periods. The period might be fast for one application whilst being slow for another. For instance, the period for engine control might be measured on the millisecond timescale, while a timescale of seconds might be suitable for scientific data acquisition.

**Types of Real-Time Systems**

Real-time systems can be grouped into two categories, namely hard and soft real-time systems. The system has to respond to each task within a defined time period in hard real-time systems. If the real-time system does not response to a hard real-time task before its deadline, the system fails. Missile guidance and anti-lock breaking systems are examples of hard real-time systems.

On the other hand, the system will not fail if applications miss their deadlines in soft real-time systems. The execution deadline is also important for these systems, but failure to meet them might be tolerable. Examples of soft real-time applications are data acquisition systems and operating systems.

**Characteristics of Real-Time Tasks**

Real-time tasks fall into two main categories: Synchronous and asynchronous. Synchronous events are predictable as they occur periodically and with precise timing, such as the video and audio recording in a camcorder. On the other hand, asynchronous

events are completely unpredictable, such as I/O requests arriving at the memory. The arrival time of the request can not be estimated. It is quite challenging to provide a deterministic response time for asynchronous real-time tasks.

**Challenges in Real-Time Systems**

One of the most important concepts of a real-time system is predictability. All tasks in real-time systems should be completed before a particular, predetermined amount of time. If one of these tasks exceeds its predetermined execution time, it could cause unreliable results or even a system crash, depending on the type of real-time system. Therefore, the prediction of execution time for each task is of vital importance for real-time systems [52]. Also, real-time systems are required to detect and correct internal and environmental failures. This mechanism is crucial for such systems where the user cannot easily reach the system, such as in space craft and satellite applications.

Moreover, many real-time applications have multiple events arriving independently. In the existence of concurrent architectures and a single processor, a scheduling algorithm is needed to define the priority assigned to these events. It is challenging to develop a real-time scheduler for such types of applications.

This section shows the importance of the deterministic performance behaviour of a system for hard real time applications. Each component of the system is required to respond in a predefined time period to prevent hard real time applications failing. One of the core components of such systems is storage which has a significant effect on response time of the system. A sole flash based storage does not meet the core requirement of the hard real time applications. Therefore, a well designed mechanism is required to develop real time flash storage.

## 2.7 Summary

A background study of flash memory technology is presented in this chapter. A generic FTL and its architectural design were reviewed. It emerges the crucial role of FTL design with its main components including; address mapping, garbage collection, wear levelling, and bad block management.

One of the important role of the FTL layer is to reclaim invalid spaces in the memory which is called garbage collection. It was revealed that garbage collection causes a long and non-deterministic time delay which is problematic for high integrity systems where deterministic response time is necessary.

Next, internal architecture of a generic SSD storage system was presented. The SSD architectures were discussed in terms of performance, reliability, endurance, capacity and cost. Current RAID techniques to enhance the performance and reliability of the SSD architectures were investigated. Conventional HDD-based RAID can not be directly applied to SSD arrays because of the risk of wearing out individual devices in the array simultaneously. This problem is addressed using RAID architectures which enhances reliability, but ignores the deterministic response time requirements of hard real time systems.

For better understating of design parameters, real time systems were reviewed including their types and some common challenges in real time systems. It is observed that predictability of access time for I/Os is highly important requirement for hard real time applications and the SSD RAID architectures do not meet the requirement— due to uncoordinated garbage collection and off-line device replacement.

To conclude, flash memory based storages offer a number of advantages but real-time support issues of the memory limits its usage in hard real time applications. These issues need to be addressed to improve the reliability of flash based storages for hard real time applications.

# Chapter 3

# Real-time Support Issues for Flash Memory Storage Systems

This chapter presents real-time support issues for NAND flash-based SSD storage architectures. Generally, existing real-time garbage collectors—based on partial or preemptive cleaning policies—were proposed to provide deterministic response times for single flash memory units for a non-deterministic cleaning process. However, they do not particularly suit the reliability mechanism inherent to multiple devices used in a RAID mechanism with a strict age distribution policy. Moreover, it is clear that current reliability enhancement techniques for SSD RAID [8,11] have serious deployment issues for hard real-time systems, especially when device replacement is required.

## 3.1 Introduction

Conventionally, RAID systems have been used to provide protection against chip-level failure [10]. An array of memory components is used to enhance both reliability and performance by providing concurrent access to members of the array. Parity-based RAID levels (such as RAID 4 and RAID 5) have been widely used to enhance data

reliability. These RAID techniques reserve an element of the array to keep redundant data, and thus failures can be corrected at the chip, block or page level when an uncorrectable multi-bit error occurs, as described in Chapter 2.

However, the RAID system cannot be directly applied to flash arrays because of the risk of the wearing out of the elements of the array simultaneously [8]. To address this problem, a number of RAID-based flash storage architectures have been proposed that enhance system reliability by protecting the flash memory against chip-level failures in high integrity systems [8, 11]. They are based on an uneven parity distribution technique to prevent the simultaneous wearing out of the chips' components.

These RAID techniques offer significant improvement to the reliability of flash storage systems but it does not meet the (sometimes strict) requirements of real-time systems, where guaranteed response times are often necessary. This is primarily due to two main reasons: first, the erase-before-write characteristic of the flash memory and the management of resources as they are used; second, these techniques require devices to be taken off-line while components are replaced, so consequently these devices are of limited use in hard real-time systems.

The aim of this chapter is to present the real-time support concerns—in terms of non-deterministic response time—in NAND flash-based SSD storage systems based on parity-based RAID techniques. In addition, to point out the inherent problems of RAID in SSD storage systems, existing studies in the literature are also discussed from the perspective of hard real-time applications.

This chapter is organized as follows: serious issues with flash memory regarding garbage collection that cause unpredictable response times are discussed in Section 3.2. Following this, reliability concerns of flash memory and possible solutions to these concerns are discussed in Section 3.3. Section 3.4 presents current solutions and their limitations related to SSD RAID systems. Finally, summary of the chapter is presented in Section 3.5.

## 3.2 Unpredictable Performance on Garbage Collection

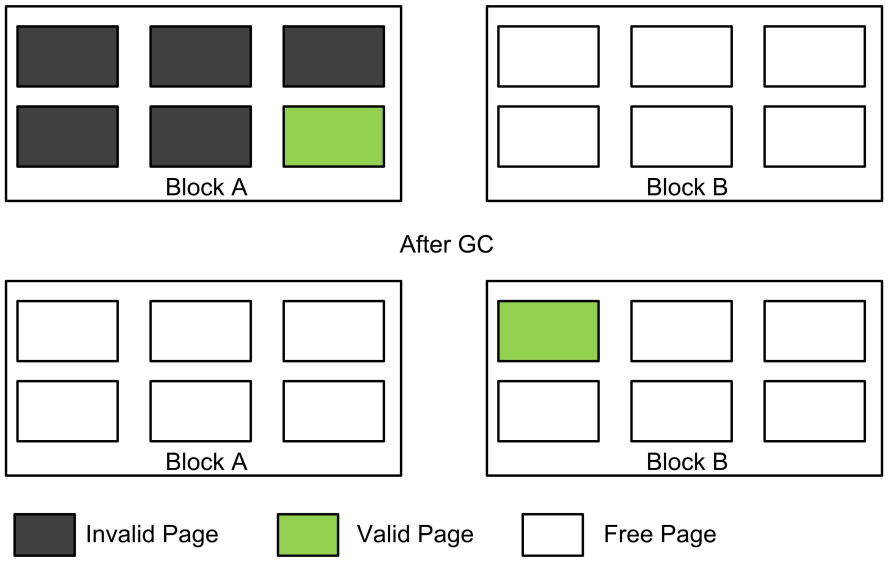



Figure 3.1: Garbage collection

This section discusses non-deterministic behaviour of GC. The basics of GC have already been presented in Chapter 2. An example of a typical GC process is illustrated in Figure 3.1. The FTL first selects a victim block based on a selection algorithm. The GC then copies all valid pages from the victim block (block A) to an empty block (block B) and updates indexing information in the address mapping table according to the new physical locations of the valid pages. Once all pages have been transferred, the block is erased and becomes a free block.

Since a generic GC is non-interruptible and includes an expensive erase operation (nearly 1.5 ms), it significantly degrades I/O performance and reduces the lifetime of the flash cells. If an instruction is assigned to the flash memory while there is an on-going GC, the instruction has to wait until GC is over since the internal register and memory channel are occupied by the GC operation in traditional FTL mechanisms. This causes an unwanted latency and non-deterministic response time, which is a bottleneck for high reliability applications where guaranteed access time is required.

The time cost of GC for a victim block is calculated based on the formula below:

$$GC_{cost} = (B_{valid} \times t_{copy}) + t_{er} \tag{3.1}$$

$GC_{cost}$ is calculated by considering the number of valid pages in a victim block ($B_{valid}$), the time taken to copy a page ($t_{copy}$), and the time penalty, or cost, for a block erasure ($t_{er}$), as stated in Equation 3.1. Although time costs are static in the formula, $B_{valid}$ can vary depending on the usage pattern of the victim block. Since GC is non-interruptible, and a request overlapping with an ongoing GC has to wait until it is over, the response time of the request becomes unpredictable. Moreover, in some cases, FTLs adopt a bulk cleaning policy where the GC is not only triggered for a single block, but a number of blocks are involved in the GC until a certain amount of free space is generated [53]. This further enlarges the problem.

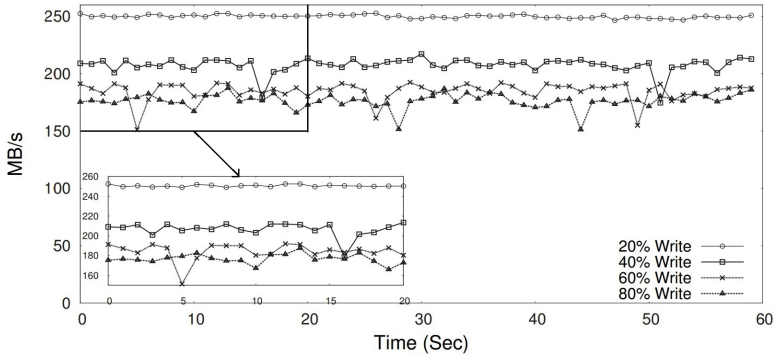



Figure 3.2: Pathological performance behaviour of individual SSDs taken from reference [34]

Moreover, a recent study reveals that performance variability of the flash based devices are more in RAID systems compared to that of single SSDs due to an uncoordinated garbage collection mechanism [34]. Figure 3.2 and Figure 3.3 show the throughput changes with different types of workloads in a MLC based single SSD device and multiple SSDs in a RAID architecture, respectively (the sub figures inside

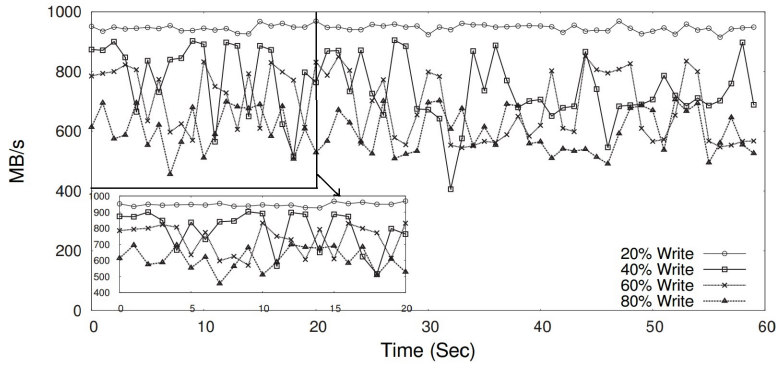



Figure 3.3: Pathological performance behaviour of multiple SSDs with RAID taken from reference [34]

main figures represent enlarged version of the first 20 seconds of the main figures). It is clear that the performance variability is significantly increased with the usage of SSD in RAID architecture. The performance variability is significantly reduced via globally coordinating the cleaning operation in reference [34] but it only considers system performance and fails to provide deterministic access time for storage.

Since the unpredictable performance on garbage collection can be tolerable for soft real-time applications, commercial off-the-shelf SSDs have been designed to provide improved performance and reliability ignoring the deterministic response time capability [54–56].

## 3.3 Reliable and High Performance NAND Flash-based Storage

One of the most important issues when using SSD devices in high reliability environments is the fact that they physically wear out over time—normally related to the number of times parts of the device are erased—meaning that data stored on the device

will become unreliable over time. This failure characteristic is different to traditional magnetic media that typically fail non-deterministically.

One approach to enhancing reliability in the case of wear-out is the use of EEC [57–59]. This is parity-style data that is stored in the metadata of each page of memory, and is used to cross-check, or repair, the data it relates to at the point where the data is read. For SLC-based flash memory Hamming codes are used, but these are not sufficient for MLC-based flash memory [60]. Moreover, high capability ECCs have been proposed for MLC-based flash memories which build on more complex algorithms and consume more power, such as Reed-Solomon [61], EVENODD [62], and Row-diagonal Parity [63]. However, there are limits to how much they can improve reliability as the potential to repair is limited, and therefore more powerful ECCs are needed for next generation flash memories [41].

For these reasons, it is not ideal to use EECs as the sole mechanism when reliability is crucial. Where storage capacity is an issue, MLC technology can offer benefits over SLC devices. However, MLC devices suffer from greater unreliability—particularly when the device ages—than SLC devices due to lower erase endurance.

Recent advances in storage technology have applied RAID technology to SSD storage in order to improve reliability, data integrity, and system performance [45, 46, 64, 65]. In particular, RAID 0-based redundancy has been studied with SSDs to provide enhanced performance [45]. Using a log structured approach — where data is not updated in its same physical location, a new copy is appended at the end of log by updating its connection link — , not only the performance but also the lifespan of the SSD RAID storage is improved [46]. To minimise the performance overhead of random workloads and improve system dependability, the delayed partial parity update schema has been proposed which improves the performance of the system up to 38% compared to traditional RAID-5 [64]. Moreover, a configurable RAID mechanism for SSDs has also been presented [65] to reduce parity data overheads thereby reducing

the read and write costs up to 84% and 56% compared to RAID 5 and RAID 6. Furthermore, an adaptive RAID algorithm and an efficient RAID mechanism have been patented to improve system reliability [66] and performance [67].

Common RAID techniques, such as RAID 4 and RAID 5, hold parity data to reconstruct original data in case of block errors [68]. However, the use of these techniques with SSDs reveals the problem of simultaneously wearing out all devices. In a typical disk-based RAID environment, data is stored across a number of devices, with one of the devices being used to store parity. When a particular device fails, the remaining devices can be used to calculate and logically reconstruct the data that was stored on the failed device. However in an SSD environment, components will typically wear out at the same rate, thus rendering all elements of the array unreliable at the same time. This problem was first stated in reference [8].

In previous works, RAID-based architectures were presented that enhanced the reliability of SSD storage systems by mitigating this problem [8, 11]. This was achieved by guaranteeing wear imbalance between each component in the array, thereby ensuring they do not reach their endurance limits simultaneously. This is achieved using two primary techniques: the first is an uneven parity distribution that ensures erases across components are distributed unevenly; the second is a device copy/swap algorithm that moves data around and manages lifespan as components reach their endurance limits. Although maximum reliability is achieved by only random writes, reference [11] shows improved reliability for both sequential and random writes via a forced random write mechanism, as presented in [69].

## 3.4 Current Challenges and Solutions

A number of works have been conducted to improve the efficiency and reliability of NAND flash-based SSD storage systems over the past decades. Multi-chip paral-

lelism and RAID technologies have been used with SSDs to improve their throughput [38–40, 43–46]. On the other hand, to improve the data reliability of flash-based storages, there have been many mechanisms investigated including wear levelling, ECCs, erasure codes, flash file systems, hot-cold data identification, load balancing, parity-based RAID, and real-time garbage collection mechanisms [15–18, 24, 25, 49, 70–75].

One of the main aims of commercial products is to increase the I/O bandwidth with multichannel architectures, which enables the interleaving of data over multiple flash chips [55, 56]. Moreover, to exploit internal parallelism of flash-based storage, several techniques have been introduced in the literature [37, 76]. However, these techniques mainly focus on enhancing system performance, ignoring guaranteed upper bounds of I/O access latency and data reliability.

In the last decade, various techniques have been introduced proposing various system architectures in order to overcome the non-deterministic behaviour of the flash memory [15–18]. In particular, the real-time aspects of flash memory first studied by [15] propose that real-time garbage collection by using a garbage collector thread for each real-time task. It guarantees free space for incoming real-time tasks to be executed without an unpredictable GC delay. It is based on a page address mapping technique. However, it requires file system support to predict bounded access time, which limits its usage.

Choudhuri et al. [16] proposed a real-time FTL (the term real-time FTL is sometimes referred to in this thesis as real-time GC) that guarantees an upper bound for I/O operations in NAND flash, called GFTL. A partial block cleaning policy is applied with the help of extra free blocks in order to reduce the upper bound cost of write operations. The GFTL always keeps extra blocks for write operations to be executed. It uses a 1:1 block mapping between logical and physical blocks. If a corresponding block is full then data is stored in a write queue which always has to be available.

The key algorithm of GFTL is partial block cleaning. The partial block cleaning policy divides the GC process into a number of partial steps. The time length of each step should not be longer then the longest atomic operation of flash memory, which is the erase operation. Also, the time length is equal to the period time for real-time tasks. It introduces three main phases for garbage collection, namely read, write and block erase. Although it provides upper bounds for real-time tasks, it needs an extra copy operation which increases average system response time and the number of physical write operations.

Moreover, Qin et al [18] proposed a real-time flash translation layer called RFTL to guarantee a worst-case response time. RFTL uses a distributed partial garbage collection policy similar to GFTL. However, RFTL not only focusses on worst-case response time but also considers the average response time of the system.

The main difference between GFTL and RFTL is the method by which garbage collection is achieved. GFTL retains a global GC queue for the whole memory, while GC is scattered over logical blocks in RFTL. A hybrid-level mapping technique is adopted in RFTL. Every logical block corresponds to three different physical blocks, namely the primary block, the replacement block and the buffer block. The primary block is the first block used for a write operation. When it becomes full, all incoming requests are handled using the buffer block. The replacement block is used when GC is triggered, as the purpose of freeing a block is that valid pages need to be moved. The functionality of the blocks can be changed dynamically depending on the requirements of the mechanism.

Moreover, to remove the unpredictable latency of GC, a pre-emptive GC (PGC) mechanism has been proposed in [17]. Unlike RFTL and GFTL, the mechanism does not require additional memory to offer predictable access times, which makes it attractive for embedded systems where system resources are limited. Two types of pre-emptive mechanisms are offered, a semi pre-emptive GC, which is only available at

predefined pre-emption points, and fully pre-emptive GC, which enables immediate suspension of GC to serve requests. Since the flash technology does not fully support suspending ongoing requests, the feasibility of the fully preemption GC mechanism is discussed in reference [17]. More details regarding PGC will be given in Chapter 4.

On the other hand, a method guaranteeing real-time access to flash memory has been patented [77], the research into which also investigated this problem at the single chip level. Moreover, garbage collection-related performance and reliability issues of flash memory-based devices that ignore the non-deterministic access time have been addressed in a number of patents [78, 79].

### 3.4.1 Real-Time Support Concerns in Real-Time FTLs with RAID

Although real-time FTLs offer a guaranteed response time and enhanced system performance for I/Os in the existence of active GC, they only consider the non-determinism problem at the single chip level, not on flash storage with multiple chips with RAID architecture. This creates a number of real-time support concerns on a storage system with RAID architecture.

First, if a real-time garbage collector is allowed free rein over erase operations while garbage collecting, this would affect the strict management of the lifespan of each SSD in the array that is conducted to provide enhanced reliability. An uncontrolled cleaning would result in unwanted ageing ratios, which reveals a reliability problem for mechanisms where strict control over ageing ratios of flash memories are necessary. One of the key points in developing an efficient GC mechanism is to determine an optimum threshold level for the cleaning process, especially for the GC mechanisms which are triggered based on remaining free space [80]. However, it is challenging to provide an optimum threshold level for an efficient cleaning mechanism as it is highly dependent on incoming workload weights. Also, the experimental

Table 3.1: Maximum response times of existing real-time FTLs

| **Technique** | **WCET** |
|---|---|
| GFTL [16] | $t_{er}$+*max* {U($e_w$), U($e_r$)} |
| RFTL [18] | *max* {$t_{er}$+ U($e_w$), U($e_r$)} |
| PGC [17] | $t_{er}$+*max* {(U$e_w$), U($e_r$)} |

results—which will be presented in Chapter 4—indicate that there is no "optimum threshold" for each concurrent unit of the array to maintain the reliability mechanism.

Second, since real-time FTLs only focus on the problem at the single chip level, the improvement in WCET is limited because there is no way that an I/O can avoid the erase operation of the GC in the proposed techniques. The main idea of existing techniques is to partition or pre-empt the GC process. As the GC can only be portioned/pre-empted at the granularity of the atomic operations of the flash memory, none of the existing proposed techniques present a means of avoiding the most expensive atomic operation of flash memory, namely the erase operation. In the worst-case scenario, a request has to wait for an erase command to finish, which is nearly ten times more expensive than a write operation, limiting the improvement in WCET and significantly reducing system performance. For the request to be serviced takes U($e_w$) or U($e_r$), which represents the maximum execution time for the write and read operations, respectively. As illustrated in Table 3.1, all WCETs have inherently expensive $t_{er}$ costs (time taken for an erase operation).

To maintain the reliability enhancement mechanisms while offering a deterministic access guarantee to flash-based storage, a dynamic GC triggering mechanism could be a possible solution. Depending on the current ageing level of the devices in the array, GC operations can be postponed or triggered in advance. WCET and the performance of the storage mechanism can be further improved by benefiting from concurrent system architecture and redundant data of the parity-based RAID mechanisms.

### 3.4.2 Real-Time and Performance Concerns of the Reliability Enhancement Mechanisms

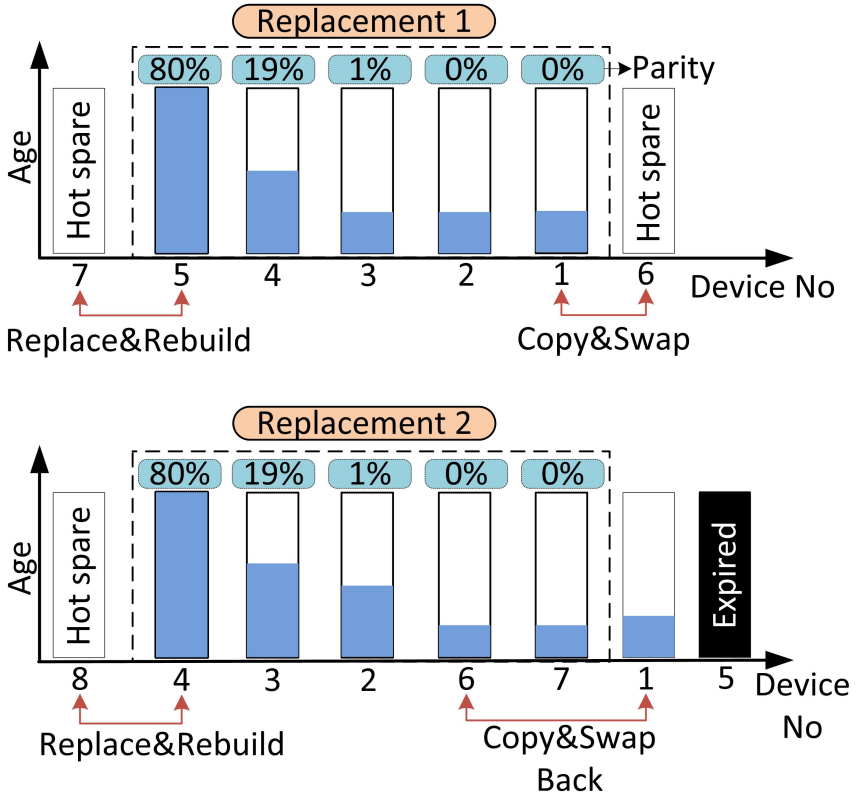



Figure 3.4: Device replacement operation

Reliability enhancement mechanisms require devices to be taken off-line while components are replaced—consequently, these devices are of limited use in hard real-time systems. This presents serious deployment issues for hard real-time applications.

The device replacement and copy/swap operations of the reliability mechanism of reference [11] are illustrated in Figure 3.4. The upper level of the diagram (Replacement 1) represents the system during the first device replacement process, and the lower level (Replacement 2) represents a later state of the system—during the second device replacement process, along with actual parity distributions and ageing levels of each device, based on the parity ageing formula in reference [8]. In the first replacement

process, the most aged device (SSD 5) needs to be changed with a hot spare device (SSD 7) when it reaches its endurance limit; however the need to actively ensure that an uneven parity distribution is still maintained, and that the correct device ageing is achieved for subsequent replacements. This is achieved using a copy/swap operation involving SSDs 1, 2, and 6. At this point, the least aged device (1) is taken off-line (and remains off-line until the next replacement process begins), and is replaced with device 6. The effect of this is that the age distribution between device 1 and the next youngest active device (2) is allowed to increase. In any subsequent device replacement operation, devices 1 and 6 are swapped back.

Furthermore, parity data has to be redistributed as part of a replacement process. In this example, the parity assignment across the devices is (80%, 19%, 1%, 0%, 0%), which means that device 5 holds 80% of the parity, device 4 holds 19%, and so on. When device 5 is replaced by device 7, 61% of the parity needs to be moved from device 7 to device 4 in order that it becomes the most parity-dense; a similar domino effect subsequently ripples down the system. As a result, all the parity data on the most aged device is first reconstructed on device 7 and then shifted to other devices. These operations necessarily increase the write amplification factor and device replacement time. Write amplification refers to the additional writes caused by operations such as garbage collection and wear levelling, and is formulated as the ratio of the total number of write operations performed to the write operations requested by the host [81]. This presents limitations in hard real-time environments.

Device replacement mechanisms are usually triggered when idle time periods or low density streams are detected in a workload. The garbage collection technique could be using the idle time detection approach, and therefore there will be potential overlap between the garbage collection process and the swapping operation in the target device, which will slow down the I/O rate further when replacing a device. To provide an on-line and efficient RAID reconstruction mechanism, a number of studies have

been conducted based on traditional magnetic hard disks such as [19–21]. Although they offer on-line reconstruction, these are not fully compatible with solid state RAID systems. For example, these only focus on the replacement process in case of a sudden (non-deterministic) device failure. However, unlike magnetic hard disks, the failure of solid state devices due to wearing out can be predicted using the number of erase operations performed and thus an expensive reconstruction operation can be replaced with a low cost data migration process.

There have been several techniques proposed to replace solid state devices in case of failure. Differential RAID (Diff-RAID) [8] increases the reliability of the SSD RAID by distributing parity data unevenly across the elements in an array. It also provides a device replacement which shifts parity according to the next parity assignment—however a significant difference is that it applies a reconstruction method based on magnetic devices and therefore increases write amplification and device replacement time due to additional parity movement operations. To reduce the parity data overheads, a configurable RAID mechanism for SSDs is presented in [65]. The mechanism stores less important data using RAID 0 (which does not provide redundant data for recovery in case of failure) while more important data is stored using parity based RAID levels. Although this reduces performance overheads incurred from parity data, it does not provide a replacement policy for a complete device failure—only partial levels of data recovery.

To guarantee average I/O latency while migrating data, reference [82] presents a control-theory approach which dynamically adjusts the speed of data migration by periodically measuring I/O performance of the magnetic storage devices. Thus it migrates the majority of data during idle time periods or low density streams. Also, reference [83] presents an idle time detection method that achieves zero impact on the foreground application whilst rebuilding the RAID. The garbage collector technique presented in the reliability mechanism in [11] also use this idle time detection

approach and so there would be a potential overlap between garbage collection and the replacement operation.

Since the failure characteristic of flash memory is different to that of traditional magnetic media, which typically fail non-deterministically, a proactive device replacement strategy can be proposed, where the performance overhead of device reconstruction can be replaced with an efficient data migration mechanism.

## 3.5 Summary

In this chapter, existing real-time support and reliability mechanisms for flash based storages were reviewed and analysed. First, the real time support concerns in flash-based SSD storage systems have been presented. It was revealed that flash based SSD systems do not meet the real-time constraint of the hard real-time systems due to the unpredictable GC performance. To address the non-determinism problem of flash memory, a number of real time GC mechanisms are exploited to provide guaranteed access. However, they only consider single flash memory ignoring architectures of multiple flash memories based storage system where it is required to strictly control the ageing rate of multiple devices. It was also observed that the performance variability for multiple memory devices storage system is more in comparison to a single memory device due to uncoordinated GC.

Then reliability concerns in flash based storages and their possible solutions were discussed. Two possible solutions were revealed: ECCs and RAID mechanisms. It was concluded that RAID mechanism is more capable of achieving a certain level of reliability especially for the massive storage systems where new technologies of flash memory are required. Especially the parity based RAID mechanisms with a load imbalancing techniques offer improved reliability compared to the traditional RAID mechanisms which have the high possibility of simultaneous device failure. In addi-

tion, a number of existing approaches aimed at providing online and efficient RAID reconstruction based on traditional magnetic hard disks. However, online reconstruction mechanisms are not fully compatible with SSD RAID systems. Furthermore, these mechanisms only focus on the offline replacement process, which is inefficient.

# Chapter 4

# Dynamic Pre-emptive Garbage Collection

This chapter introduces a dynamic pre-emptive garbage collection (Dynamic PGC) mechanism into the reliability enhancement mechanism. It dynamically adjusts the existing pre-emptive garbage collection [17] with due consideration for the optimum age distribution percentages of SSDs in the RAID array according to the reliability mechanism. The proposed mechanism provides deterministic access guarantees for I/O without ignoring the reliability mechanism. It also improves the real-time efficiency of the system by dynamically synchronizing the GC modes of each device in the array. For the rest of this thesis, the term SSD is should be considered interchangeable with the terms flash device, device, and flash based storage.

## 4.1 Introduction

GC operation has a significant negative impact on the performance and lifetime of flash storage. GC causes non-deterministic response times, which presents bottlenecks for real-time applications. Also, GC operation includes an erase operation, which reduces

the lifetime of the flash cells as they physically wear out. Therefore, the flash block that is going to be garbage collected needs to be chosen carefully.

A recent enhanced GC mechanism included the pre-emptive GC mechanism, where incoming requests are served at predefined pre-emption points of a GC process [17]. The algorithm consists of two GC triggering threshold levels to distinguish the priority of the GC process over I/O requests. This mechanism clearly identifies the upper bounds for operations on a single NAND chip. However, it is challenging to set suitable thresholds for pre-emptive GC where the flash memory is being used in a reliable concurrent architecture, such as the reliability mechanisms in a RAID array where strict management over the ageing ratios of the elements in the array is required. Since each element has a different workload weight due to an unevenly assigned parity percentage in the RAID mechanism, it is expected that a stable threshold level would deviate from the targeted ageing distribution ratio for each SSD, and thus the reliability mechanism may be disturbed.

In previous studies it has been noted that the number of GCs required is highly affected by the types of workload. For instance, random writes and update operations of small sizes increase the number of GC operations significantly [48]. Also, these workloads slow down the GC due to a high number of valid pages being copied during the GC [18]. The reliability mechanism focuses, in particular, on workloads which mainly consist of random writes, which themselves increase the number of cleaning operations performed.

The contribution of this chapter can be summarised as follows: the dynamic pre-emptive garbage collection is proposed for SSD RAID. It has two main components. First is dynamic garbage collection which not only considers the age distribution percentages of the reliability mechanism, but also adjusts the threshold levels for efficient GC operation to reduce the associated timing cost. Second, a pre-emptive mode controller is presented to coordinate ongoing GC operations with a holistic view of the

system. This pre-emptive mode controller reduces the down time of the array during erase operations, thereby causing fewer I/Os operations to be delayed.

The chapter is structured as follows: In Section 4.2, the fundamentals of existing PGC are discussed, followed by an empirical demonstration of the limitations of the PGC over the reliability mechanism in Section 4.3. The architectural design of the mechanism is discussed in Section 4.4. To address the associated limitations, a dynamic pre-emptive garbage collection for flash RAID array is proposed in Section 4.5. Finally, chapter is summarised in Section 4.6.

## 4.2 Pre-emptive Garbage Collection

GC is a non-interruptable process used to reclaim invalid spaces in flash memory which creates bottlenecks for real-time applications. To eliminate this behaviour, a set of work has recently been presented in the literature which proposes a pre-emptive GC schema that permits I/Os to be serviced by pre-empting active GC [17]. A GC process includes number of atomic flash operations such as read, write and erase. The mechanism only allows for the suspension of the cleaning operation between such operations, and thus it is referred to as semi-PGC. The study also explores the feasibility of a fully pre-emptive GC by assuming a flash memory that supports suspend/resume commands for atomic flash operations. In this study, only semi-PGC is considered as it is realizable, and will be referred to merely as PGC for the remainder of this thesis.

Figure 4.1 illustrates possible pre-emption points for any given GC operation. Firstly, valid pages from the selected block are copied into an available free block. A page copy operation consists of reading, transferring, and writing a page and updating the corresponding mapping table. The page movement cost can be eliminated if both blocks are located on same flash chip. After migrating all valid data, invalid pages are subsequently reclaimed with an erase command.

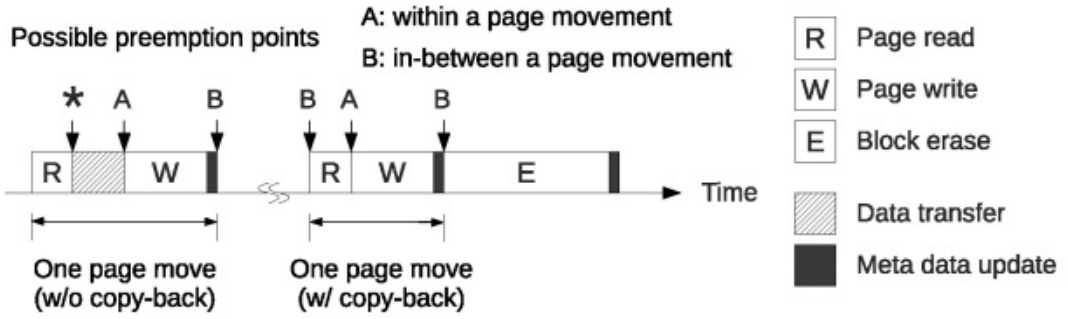



Figure 4.1: Pre-emption point in PGC, taken from reference [17]

PGC provides two different pre-emption points, which are denoted by A and B and are illustrated in Figure 4.1. If there is a request in the queue, then a higher priority is given to the request over the partial GC process and it is serviced immediately. Pre-emption point A is located just before the write operation while point B is positioned between page copy operations. At point B, any kind of I/O request can be serviced, but at point A only write requests are allowed if flash memory supports pipelining commands of the same type, because the page buffer is already occupied at point A by a read operation to complete the copy operation. However, at point B, the memory buffer is free and can therefore be used for any type of instruction.

**States of PGC**

To prevent the system crashing due to lack of free memory space as a result of continuous pre-emption on cleaning, the PGC offer a number of states. The states differ from each other in terms of their permission level regarding the GC. Figure 4.2 illustrates the state diagram of those states. The states of PGC are defined as:

- **state 0:** GC disable
- **state 1:** GC enable with PGC schema
- **state 2:** GC enable with PGC but page consuming operations not permitted
- **state 3:** GC enable but all requests are prohibited except high priority requests.

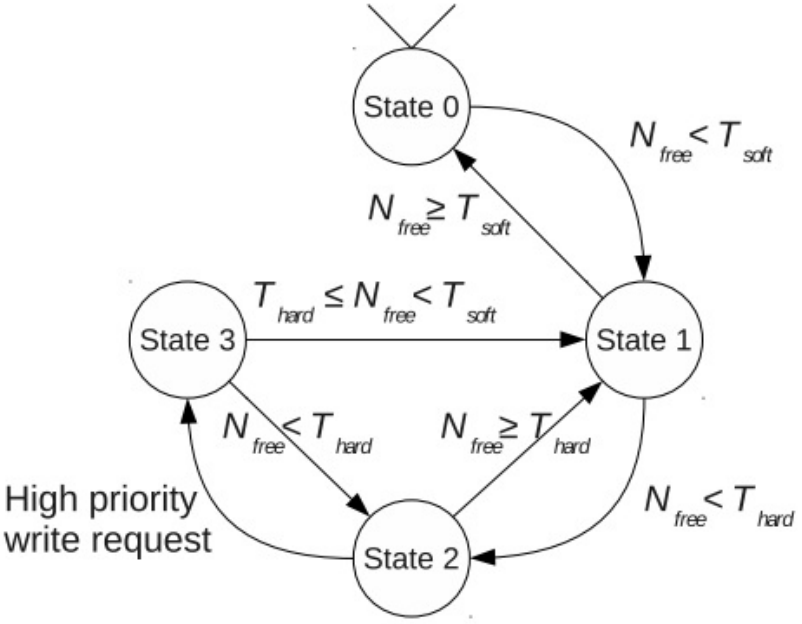



Figure 4.2: PGC state diagram, taken from reference [17]

Initially flash memory starts with state 0 where a GC operation is not allowed as there are still plenty of free locations in the memory. I/O can be served without any delay in this state. If the number of free blocks in the flash chip, ($N_{free}$) decreases below the soft threshold, ($T_{soft}$), the PGC changes its state from 0 to 1. If $N_{free}$ becomes larger than $T_{soft}$, the state is changed back from 1 to 0. PGC is activated in state 1. GC can be suspended to serve incoming requests in this state. All I/O requests have higher priority over the partial GC operations. However, if $N_{free}$ decreases under the hard threshold, $T_{hard}$, this indicates the quantity of $N_{free}$ is at a critical level and GC is immediately required and the system goes into state 2. In state 2 all memory-consuming requests—especially write and update—are prohibited. Only read requests can be served as these do not occupy free memory locations. However, if a high priority write request arrives in state 2, the system changes from state 2 to 3 to allow this request to be served. In state 3, ongoing GC is finished, and then high priority requests are handled. After

finishing the cleaning process in state 3, the system will switch to state 1 or 2 according to the number of free blocks generated.

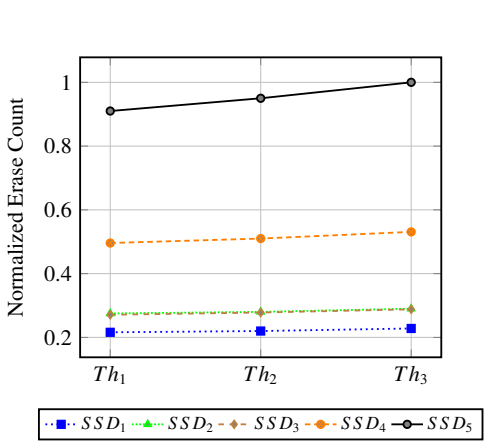
The PGC offers an upper bound for I/O response time for a single flash memory unit. In the worst-case scenario, if the host request encounters an erase operation of the GC process, which is the longest atomic process in flash, it will need to wait until this finishes and then the given host request (write or read) can be handled. Thus, the time penalty for these operations is calculated as the sum of the erase latencies, with maximum time being taken to serve the given request.

## 4.3 Limitations of PGC with SSD RAID

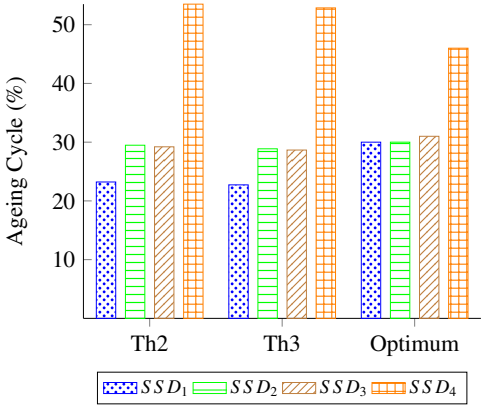
One of the key points for an efficient cleaning mechanism is the determination of an optimum threshold level for initiating cleaning. This is particularly true of the PGC, which provides deterministic response times without the need for additional memory space. However, the mechanism does not provide an optimum threshold level for cleaning as it is highly dependent on incoming workloads. Also, the following experiments indicate that there is no optimum threshold for each unit of the array in order to maintain the reliability mechanism.

To experimentally observe the effects of the PGC on the ageing ratios of the devices in the array, ageing tests were conducted with a realistic workload (financial) by using the MSR SSD simulator [37]. As the reliability mechanism is performed using maximum reliability enhancement with random write-dominant traces, the financial trace [84] was chosen. The experiments were based on an SSD array incorporating five devices based on the redundancy mechanism where parity data is distributed unevenly across the devices in the array [11].

Figure 4.3a illustrates the trends in erase counts and age distribution percentages by varying threshold levels for the financial workload. The thresholds levels refer to




(a) Normalized erase counts of SSDs by varying GC triggering thresholds




(b) Comparison of SSD age distributions varying GC triggering thresholds and with optimal age distribution

Figure 4.3: The effects of PGC over the reliability mechanism

the percentage of free space remaining in the corresponding memory component. The threshold levels were chosen as $Th_1$ (3, 2), $Th_2$ (5, 3), and $Th_3$(8, 5) for the pre-emptive GC, where the first ratios (3, 5, 8) represent the threshold level for a lower priority GC ($T_{soft}$), while the GC tasks gain a higher priority after exceeding the second ratios (2, 3, 5) ($T_{hard}$), as discussed in Section 4.2. The normalized total erase counts of each SSD increase from $Th_1$ to $Th_3$ as expected. It can be noted that the threshold levels have a considerable impact on the total erase count of the devices, especially for the device which holds the most parity data.

However, the age distributions of SSDs are not same as expected optimum ageing levels, which are calculated based on the ageing formula in [8] according to assigned parity distribution ratios (80%-$SSD_5$, 19%-$SSD_4$, 1%-$SSD_3$, 0%-$SSD_2$, 0%-$SSD_1$). Figure 4.3b shows the variation between the desired ageing levels and the realistic ageing distributions of each SSD with the financial trace under preemptible GC schema with same parity distribution percentages. It is clear that the ageing cycle of each SSD in the array is different from the that of the optimum case. For instance, $SSD_4$ suffers 13% more wear than its optimum level, while SSD 1 suffers 21% less wear at $Th_2$.

This could possibly increase the chance of data loss in the array for further device replacement operations and thus the reliability enhancement would not be maintained. The diversity in the ageing level of the SSDs is due to following reasons:

- Optimum reliability is achieved with pure random writes, where a parity data block wears $n - 1$ times faster than that of an actual data block (where $n$ is the number of devices in the array). In fact, realistic random write-dominant traces not only consist of pure random writes but also include a small portion of sequential (full stripe) and large size random writes—the data size can be between single stripe unit size, ($m$ kB), and full stripe size, ($m \times (n - 1)$ kB). For example, the average request size of the financial trace was measured as 3.6 kB but the standard deviation of I/Os was 6.6 kB where $m$ is equal to 4 kB. The figures show that realistic traces not only consist of pure random writes but also includes a number of different sizes write operations. As the size of the data write instruction increases, the ageing impact factor of the parity chunks degrades from $(n - 1)$ to 1.

- Garbage collection triggering thresholds affect the number of erasure operations, as seen in Figure 4.3a, and so an uncoordinated cleaning process affects the ageing distribution.

To create the wear imbalance for SSDs in the array for sequential write requests, a forced random write mechanism is presented in reference [69]. This mechanism converts sequential writes into random writes and creates an additional parity for each random write. The mechanism achieves better reliability for sequential writes, but the performance of the system degrades in the presence of sequential writes as the response time increases and becomes non-deterministic due to the forced random write mechanism. Also, the mechanism does not provide an optimum cleaning threshold for the reliability mechanism.

To address this problem, a dynamic pre-emptive garbage collection is presented with two main components. First, a dynamic garbage collection (referred to as $GC_D$) mechanism is presented which not only provides a guaranteed response time to the system but also considers the age distribution percentages of the reliability mechanism. Second, to improve the downtime of the PGC in SSD RAID, the pre-emptive mode controller is presented whereby the garbage collector can enter varied modes of operation. The term downtime is used to refer to the total amount of time that the garbage collector consumes (across the array) during erase operations.

## 4.4 System Architecture

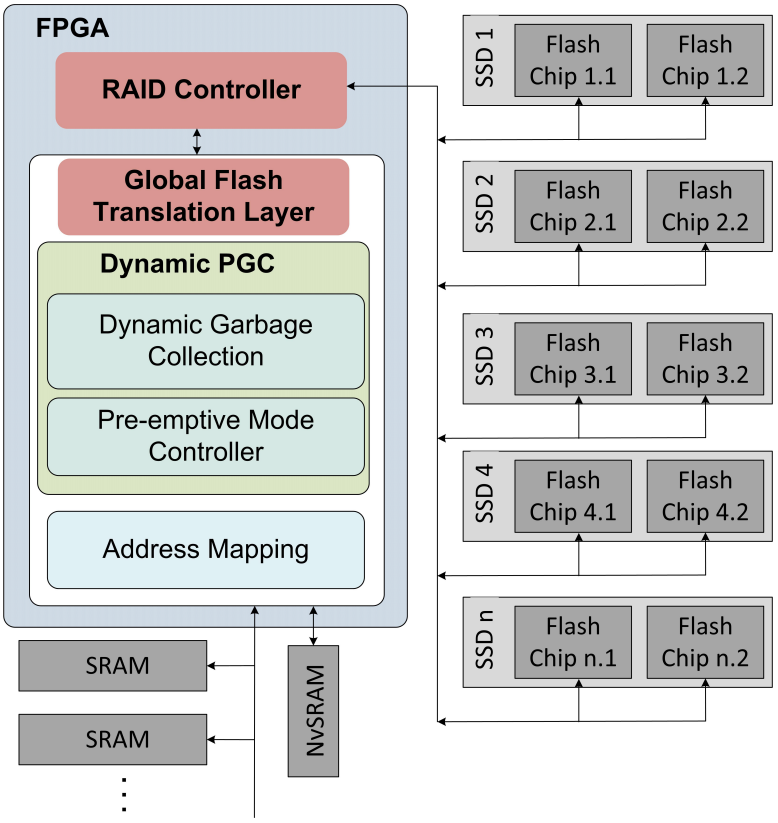



Figure 4.4: Architectural design of the PGC with RAID

Conventional FTL techniques usually allocate an independent and identical garbage collector for each device in an array, each of which has its own channel and internal register, as in reference [26]. Unlike conventional approaches, the presented mechanism has to meet requirements of reliability and real-time attributes. To maintain consistency with the reliability mechanism, the architecture first presented in reference [85] was adopted, as illustrated in Figure 4.4. This includes SSDs for storing data, memory components (SRAM, NvSRAM) for metadata, and storage management components based on Field Programmable Gate Arrays (FPGA). The FPGA-based management component includes two main blocks.

The first main block is the RAID controller. The RAID controller is responsible for partitioning the actual data into chunks and generating the redundant data for associated data chunks. The second main block is the global flash translation layer. This manages flash-specific operations in the array. It has two main blocks: address mapping, dynamic pre-emptive garbage collection (labelled as Dynamic PGC). To dynamically reconfigure the management components, the global FTL layer analyses the metadata information in each device. More details will be given about the system architecture in Chapter 7. The dynamic PGC has two main components which will be explained in the following section.

## 4.5 Dynamic Pre-emptive Garbage Collection

In this section two main components of Dynamic PGC are explained as follows:

### 4.5.1 Dynamic Garbage Collection

**Definition 4.5.1** *High Priority Garbage Collection (*$P_H$*)*

*If cleaning is urgently required in a device due to a lack of free space then high priority collection is triggered where all I/Os to the memory are blocked by the GC.*

**Definition 4.5.2** *Low Priority Garbage Collection ($P_L$)*

*Low priority collection can be interrupted by incoming requests at pre-emption points and so provide response time bounds for I/Os.*

The dynamic garbage collector adjusts triggering thresholds using metadata (number of erasures, and device usage in terms of valid, invalid, and free space) on each device. Two types of cleaning are employed, depending on priority levels—high ($P_H$), and low ($P_L$). The definitions of these are given in Definition 4.5.1 and Definition 4.5.2. The dynamic garbage collector adjusts triggering thresholds for both types of cleaning to maintain the age distribution ratios with minimal deviation from the optimum levels for random write-dominant traces. It also has the ability to postpone garbage collection if the cleaning process involves high levels of data migration (due to a large amount of valid pages in the victim block).

**Definition 4.5.3** *Most aged device*

*The device that handled the most write operations in SSD storage and thus contains the highest BER.*

**Definition 4.5.4** *Current ageing ratio ($AGE_C$)*

*The current ageing ratio of a device shows the current ageing level of a device with respect to the most aged device in the array. It is calculated as the ratio of the current erasure number of the corresponding device to the that of the most aged device.*

**Definition 4.5.5** *Optimal ageing ratio ($AGE_O$)*

*The optimal ageing ratio of a device shows the desired ageing level of a device with respect to the most aged device in the array for the maximum reliability in terms of prevention of simultaneous failure. It is calculated based on the ageing formula presented in reference [8] with initially assigned parity distribution percentages.*

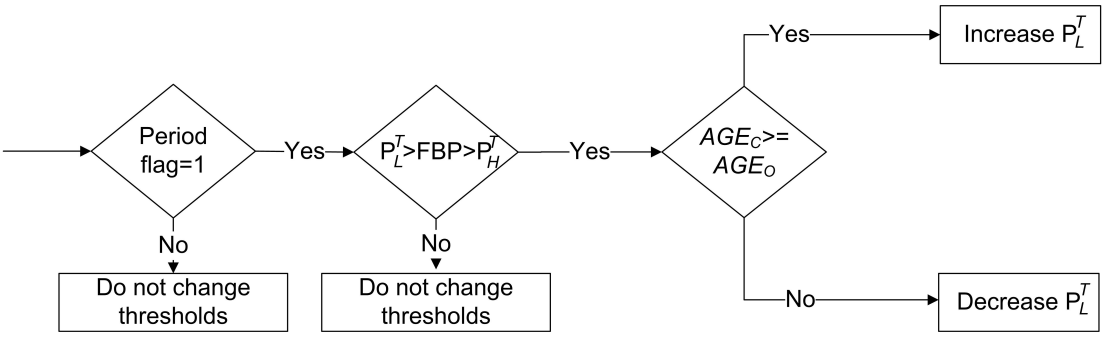



Figure 4.5: Dynamic GC triggering thresholds adjustment

Figure 4.5 shows the algorithm used to adjust garbage collection triggering thresholds for low priority cleaning. It periodically adjusts these thresholds and the current ages of each device. The erase count of the most aged device (see Definition 4.5.3) is considered as a metric to measure the period. The length of the period is configurable and can be initially assigned by the user. If *period flag* is high, indicating that the period length is completed, than the evaluation process is started to adjust the thresholds. Before serving each free space-consuming operation, the garbage collector checks the free block percentage for each target device (denoted $FBP$ in Figure 4.5). If it lies between the two triggering thresholds ($P_L^T > FBP > P_H^T$) it compares the current ageing ratio of the corresponding SSD, $AGE_C$ (see Definition 4.5.4), with its optimal level, $AGE_O$ (see Definition 4.5.5). If this is not optimum, it changes the ageing speed of the device by adjusting the threshold level; the $P_L^T$ is decreased if the current ratio is greater than its optimum level, and increased if it is less than its optimum level.

However, the higher GC threshold level may cause an earlier cleaning process on a block with a high number of valid pages. This causes the following bottlenecks:

- More valid pages in the victim block causes more data migration and thus the completion time of the cleaning process is prolonged.
- Less invalid spaces will be reclaimed and the GC efficiency might be quite low.

To address these problems, a new feature is introduced which is able to delay GC if cost efficiency—in terms of time taken to completion—is relatively low. Details of initiating/postponing the cleaning process may be explained as follows:

**Initiation/postponement of GC requests**

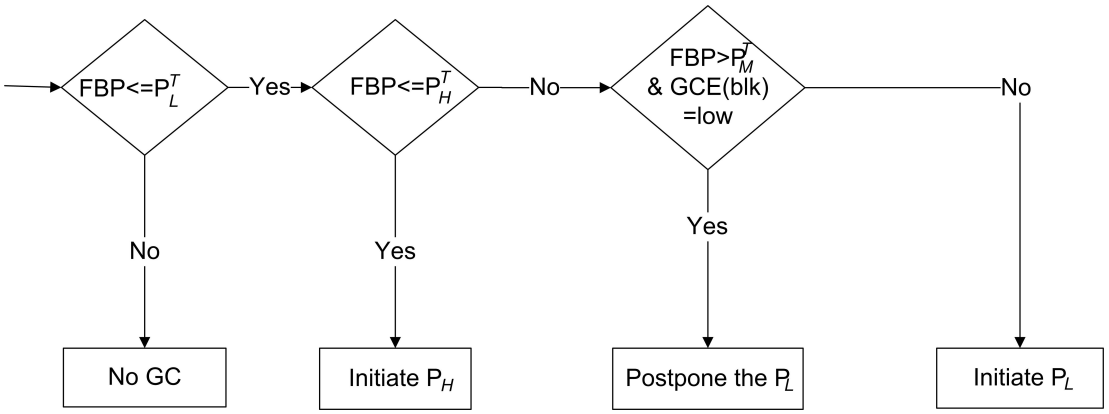



Figure 4.6: Initiation/Postponing of GC requests

Figure 4.6 shows the decisions to initiate/postpone a garbage collection request. If the amount of free space in the corresponding memory is higher than the triggering threshold, $P_L^T$, then GC is disabled. However, if it is lower, the evaluation process is started. In the second step of the algorithm, $FBP$ is compared with $P_H^T$ to define whether an urgent cleaning is required or not. If it is not required, the $GC_D$ checks as to whether or not cleaning should continue or be postponed. If the free block ratio is tending towards $P_H^T$ then postponing cleaning would result in long delays due to entering high priority cleaning levels. To prevent this, a medium threshold level ($P_M^T$) is introduced. Before postponing a cleaning process, $GC_D$ checks if the free block percentage is greater than this medium threshold. If it is lower, then pre-emptive collection is triggered as the free block ratio is nearing high priority levels. If it is greater than the medium threshold, the $GC_D$ further checks the efficiency of cleaning the victim block. The victim block is determined based on a greedy approach, i.e., selecting the block with the highest amount of invalid pages among sealed blocks (blocks with no free

pages). The efficiency function ($GCE$) determines the efficiency of cleaning the dirtiest block based on the amount of invalid and the total number of pages in the victim block. If the proportion of the invalid pages is low—when the number of invalid pages is less than the predefined threshold—it is considered a high cost cleaning process and is postponed. In the case of a low cost cleaning—where the number of invalid pages is high—then the partial tasks of low priority cleaning are generated and inserted into the I/O queue of the storage device.

As the performance overhead of high priority cleaning is much higher and its completion time is non-deterministic compared to low priority, high priority cleaning is avoided as much as possible, only being initiated when the system is starved due to a free space shortage. For this reason, the threshold for high priority cleaning ($P_H^T$) is set as low as is practical. Dynamic threshold calculations for both low and high priority cleaning are described in following section.

**Thresholds Calculation**

$GC_D$ calculates low priority thresholds for a given device $i$ using three parameters: the existing low priority threshold ratios of device $i$ (denoted $P_L^T.i$), the current age ratio of device $i$ (denoted $AGE_C.i$), and the optimum age ratio of device $i$ (denoted $AGE_O.i$). The purpose of these calculations is to establish where the low priority threshold for each device needs to be moved in order to adjust the ageing speed of the given device by increasing or decreasing the activation of low priority garbage collection.

$$(AGE_O.i) = \frac{(P_i \times (n-1)) + (100 - P_i)}{(P_n \times (n-1)) + (100 - P_n)} \tag{4.1}$$

The parameters used in the equations are calculated as follows. $P_L^T.i$ is the ratio of the amount of free space to total space for device $i$ required to trigger low priority garbage collection. $AGE_C.i$ is calculated as the ratio of the current erasure number of the $i^{th}$ device to the that of the most aged device (defined here as *device n* where $n$

also represents the total number of devices). $AGE_O.i$ represents the age ratio of the $i^{th}$ device to the most aged device for the optimum case, which is calculated based on the ageing ratio equation used in reliability enhancement mechanisms as shown in Equation 4.1. In the equation $P_i$ and $P_n$ represents the parity percentage of devices $i$ and $n$ (most aged) respectively.

$$(P_L^T)_i' = P_L^T.i + (AGE_O.i - AGE_C.i) \tag{4.2}$$

The threshold calculation (Equation 4.2) for devices other than *device n* (i.e., the oldest) takes into account the difference between the $AGE_C.i$ and $AGE_O.i$. The difference between them is summed with the current threshold ratio ($P_L^T.i$) to calculate the new threshold ($(P_L^T)_i'$). For example, if $AGE_{C4}$ (arbitrarily chosen) is measured as 0.465, then $AGE_{O4}$ will be 0.461 (the target optimum ratio). This figure shows that device 4 has aged faster than its optimum level. If user set the initial value of $(P_L^T)_4$ as 0.1, the $(P_L^T)_4'$ will be calculated as 0.096. The new threshold is decreased and thus the ageing speed of the device is reduced. For clarity, $(P_L^T)_4'$ then replaces $(P_L^T)_4$ as the current threshold ratio in the second iteration.

$$(P_L^T)_n' = P_L^T.n - (\sum_{i=1}^{n-1}(AGE_O.i - AGE_C.i)) \tag{4.3}$$

However, Equation 4.2 will be ineffectual for the most aged device (*devicen*). This is because $AGE_C.n$ and $AGE_O.n$ are always equal to each other as the most aged device is used as a reference point to calculate the ageing proportion of each device in the array. It is important to adjust this threshold to obtain optimum ageing levels. Thus, Equation 4.3 is introduced, which considers the overall ageing trends each of the devices in the array to change the threshold level of *device n* ($P_L^T.n$). If the overall trend is high, then $(P_L^T)_n'$ is also increased to adjust the ageing speed to obtain ideal ratios. For example, if the overall trend ($\sum_{i=1}^{n-1}(AGE_O.i - AGE_C.i)$) is measured as (-0.02), it shows

that either the ageing speed of the most aged device is slow or the rest of the devices have increased ageing speeds. Equation 4.2 is able to slow down the ageing speed of the corresponding device, but clearly this will not work for the most aged device. Equation 4.3 speeds up the ageing speed of the most aged device. taking the current $P_L^T.n$ as 0.1, then the new ratio of the threshold $(P_L^T)'_n$ will be 0.12, which will increase the triggering threshold and reduce the ageing speed for the most aged device.

$$P_H^T = h \times P_L^T \tag{4.4}$$

$$0 < P_H^T < P_M^T < P_L^T < 0.5 \tag{4.5}$$

On the other hand, $P_H^T$ is calculated based on $P_L^T$ and the $P_H$ factor, which is a predefined user variable and denoted $h$ in Equation 4.4. The equation indicates that the $P_H^T$ is proportional to $P_L^T$ and changes dynamically with the updating of $P_L^T$. The relationship between the thresholds and their bounds are given in Inequality 4.5.

Initially the same threshold level is assigned to each device. When the most aged device, which retains the majority of the parity data, reaches a predefined age level then the current age ratio of all other devices in the array is calculated. The garbage collector periodically updates these calculations, and assigns updated threshold levels to each individual device until the most aged device reaches its endurance limit. The period of updates is dependent on the number of erasure operations performed on the oldest device, being performed every $p$ number of erasure operations. The value of $p$ is configurable; a higher value decreases the ageing deviations from optimum levels, but the total number of erasures increases as well.

This dynamic garbage collection maintains the reliability enhancement mechanism by dynamically adjusting cleaning thresholds. Dowtime and performance related issues of the PGC are addressed in following section with a pre-emptive mode controller.

### 4.5.2 Pre-emptive mode controller

Pre-emptive garbage collection provides more deterministic response times for a single device. However, its usage in a RAID array can result in an increased number of requests arriving during erase operations, and consequently the performance of the array suffers. This is because, in a RAID array, an individual pre-emptive garbage collector does not have a global view of the array, but read and write operations must, due to the nature of RAID mechanisms, affect the whole array.

For a RAID mechanism which uses an uneven parity distribution mechanism, it is normal that devices which retain more parity data will be garbage collected more in the case of random write-dominant workloads. Therefore, if pre-emptive garbage collection is naively applied on the RAID mechanism, different devices can be in different garbage collection states because of their varied workload rates. In this section, a technique is presented whereby the mode of the garbage collector is controlled in order to ameliorate this problem and improve real-time efficiency.

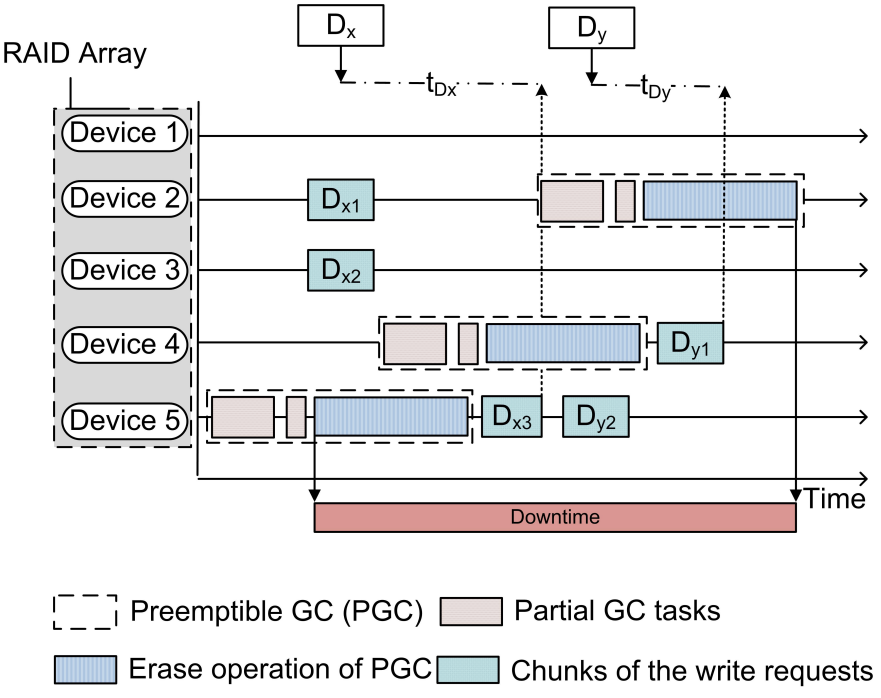



Figure 4.7: Behaviour of PGC with RAID

An example of the problem stated above is presented in Figure 4.7. A RAID array incorporating five flash-based devices was assumed, where Device 5 holds the most parity data. When write operation $D_x$ arrives from the host, Device 5 is in a state where garbage collection is enabled, and the others are in a state where it is disabled. The writing of stripe $D_{x3}$ overlaps the ongoing erase operation on Device 5, but not on the other stripes. This means that the rate determining step of writing $D_x$ will be based on the erase times of Device 5. The same problem is true for any overlapping requests, as further shown by request $D_y$—even though the request arrives when Device 5 is not being garbage collected. This scenario is highly probable within the reliability technique, particularly in the case of random write-dominant traces.

To eliminate this problem, the pre-emptive mode controller makes decisions about garbage collection by taking a view of the system as a whole. It considers all devices in the array when selecting a garbage collection state for an individual device. The goal of the pre-emptive mode controller is to reduce the downtime of the system garbage collection by dynamically coordinating the states of each garbage collector.

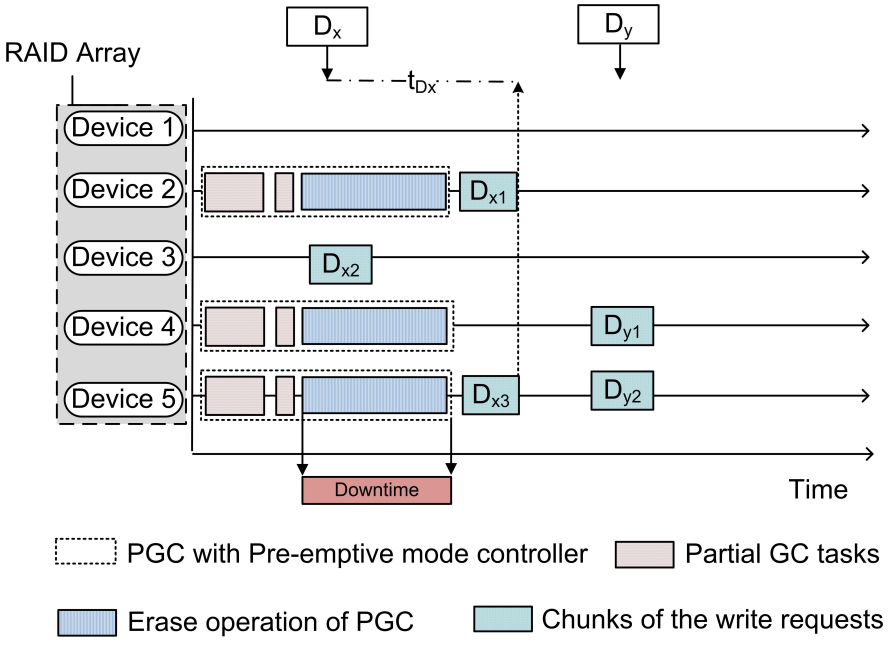



Figure 4.8: Benefits of the global pre-emptive mode controller

The effects of the pre-emptive mode controller can be seen in Figure 4.8. Unlike naive pre-emptive garbage collection, the proposed mode controller changes the state of the garbage collector by taking into consideration not only the individual device but other devices in the system. Initially, garbage collection is disabled across all devices as there is sufficient free space. When space on a given device falls below the soft threshold, then garbage collection may be enabled for that device; however, the mode controller also checks the other devices, and forces them to perform garbage collection if they have some free blocks. The resultant parallel cleaning operation minimises the amount of cleaning that might need to be done when the system is in a state where it cannot respond to I/O requests. Downtime is reduced, and the rate determining step for any delayed I/O operation reduces accordingly, as shown in Figure 4.8.

However, forced mode changes can result in an overly aggressive early cleaning on a block with a large number of valid pages (i.e., a small number of invalid pages). This can reduce the lifetime of a device more quickly and increase the cleaning cost as more valid pages are migrated. To overcome this, the mode controller checks the efficiency of cleaning a particular device before forcing the garbage collector to initiate a cleaning process. This reduces the number of unnecessary erases.

## 4.6 Summary

In the first part of the chapter, a number of ageing tests were conducted to experimentally observe the effects of the existing pre-emptive real time garbage collection mechanism. The results showed that the existing mechanisms have undesirable effects on the reliability while providing deterministic response times for access.

In the second part of the chapter, a novel real-time dynamic pre-emptive GC mechanism was proposed for SSD RAID arrays. The proposed mechanism offers deterministic access times that meet the storage demands of hard real-time embedded systems

thereby increasing system throughput. In addition, it incorporates strict age distribution management to maintain a highly reliable SSD storage system. The dynamic GC component of the mechanism dynamically adjusts the thresholds levels for each SSD to achieve the optimum ageing distribution ratios for the reliability mechanism. It periodically updates garbage collection triggering thresholds for each device in the array to change their ageing speed considering current and optimum ageing levels of devices. A new feature of GC efficiency function is introduced, which automatically delays the cleaning process based on the cleaning cost. In addition, this efficiency feature enhances the lifespan and performance of the systems while simultaneously minimising limitations of the dynamic pre-emptive GC.

Although, the dynamic GC component of the mechanism offers enhanced reliability, it ignores real time efficiency and performance aspects of the system. To enhance these aspects, a new component of the pre-emptive mode controller was incorporated in the proposed dynamic pre-emptive GC mechanism. The pre-emptive mode controller globally coordinates garbage collection states of each device with a holistic view of the system. The pre-emptive mode controller reduces the down time of the array during erase operations, thereby causing fewer write operations to be postponed compared to other published techniques.

# Chapter 5

# Garbage Collection aware RAID Mechanism

This chapter presents real-time and reliability enhancements for flash-based RAID architectures that, firstly, provide deterministic response times using a GC-aware data allocation and, secondly, it maintains the reliability of the storage system by strictly controlling ageing ratios of devices in the array using an on-line parity migration feature. The mechanism and components are explicitly expressed in a diagrammatic way. Throughout this thesis, the term GARM is used to reflect the unique real-time mechanism used to provide deterministic response times for flash-based RAID architectures. The term "random writes" is used to refer to partial stripe writes while "sequential writes" refers to full stripe writes, as described in Section 2.5. The terms are interchangeable as used throughout this thesis.

## 5.1 Introduction

The contribution of this chapter can be summarised as follows: a novel mechanism is explored that enhances data reliability in a SSD array for both random and sequential

workloads. The mechanism solves the non-deterministic problem in flash-RAID systems and provides guaranteed access time for I/O operations. It also strictly manages the parity distribution percentages across the devices in the array, thereby maintaining the reliability mechanism of the storage system.

The chapter is structured as follows: in Section 5.2, the details for GC-aware RAID mechanisms are explained. The mechanism is evaluated under all possible types of I/Os in case of ongoing garbage collections in the array. First, the effects of the mechanism under all types of read operations are presented in Section 5.2.2. Next, a novel serialised garbage collection policy is explained in Section 5.2.3. Then GC-aware random and sequential write operations are presented in Section 5.2.4 and Section 5.2.5, respectively. Section 5.2.6 presents the WCET analysis of the mechanism compared to the existing techniques. The on-line parity migration feature of the mechanism is discussed in Section 5.2.7. Finally, the chapter is summarised in Section 5.3.

## 5.2 GC-aware RAID mechanism

The reliability enhancement mechanism offers significant improvement to the reliability of flash storage systems but it does not meet the (sometimes strict) requirements of real-time systems where guaranteed response times are often necessary. This is primarily due to the erase-before-write characteristic of the flash memory and the management of resources as they are used. This non-deterministic behaviour should be eliminated for high-reliability flash storage systems.

In the previous chapter a real-time dynamic pre-emptive GC for flash-based RAID was proposed but its real-time improvements over concurrent architectures are limited. However, the GARM provides guaranteed response times for I/O operations by eliminating them from being blocked by an ongoing GC operation. The proposed mechanism can achieve higher bandwidths and lower the WCET over the existing real-time

garbage collection schemes by reallocating the target device or using the benefit of redundant data to service incoming requests. These techniques require an additional and more complex address mapping schema compared to the dynamic PGC. Details of the address mapping tables used in the architectures are as follows:

### 5.2.1 Address Mapping Tables

The mechanism contains two levels of mapping scheme. The first level of address mapping is managed by the RAID controller is named the *stripe mapping table* and keeps the link between the logical address of the request and its corresponding devices. For example, given a write request from the host, the RAID controller breaks the data into page stripes, generates the parity and determines the target devices. In existing RAIDs, stripe units are linearly placed in an array regardless of the internal status of the devices. However, the mechanism dynamically allocates GC-free devices [1] to the incoming write requests and creates the *stripe mapping table*. It is a page level striping where the size of stripe and parity units are equal to the page size of flash memory.

Second, the FTL records an *address mapping table* between the logical address and physical location. A separate page-level mapping table is adapted for each device in the array. Details of the page level mapping has been described in Section 2.2.1.

The address mapping tables described above are stored in SRAM memory. Separate SRAMs for each flash-based device have been proposed to store address mapping tables. Also, a NvSRAM is introduced to store part of metadata in the architecture. NvSRAM maintains the status of all flash pages (valid, invalid or free) of all devices in the array. The mechanism is examined under all types of I/Os with ongoing GC processes in the array. Read and write features of the mechanism are described in following sections.

[1] Here GC-free device refers a device with no ongoing GC for current I/O request

### 5.2.2 GC-aware Read Operation

**Definition 5.2.1** *Overlapped Stripe Unit*

*If a stripe unit of the I/O operation is blocked by an ongoing cleaning process in the RAID array, this stripe unit is referred to as an overlapped stripe unit.*

Parity-based RAID techniques have the ability to recover data in the instance of a single device failure. GARM utilizes the redundant data in the RAID array to calculate overlapped stripe unit (see Definition 5.2.1). If a read operation needs to access a device with ongoing GC, the data is treated as a failed data by the controller. When the host sends a read request to the flash array, the GARM follows these steps:

1. Check the stripe mapping table to find target devices of the array for the read
2. If there is an ongoing GC amongst the target devices then read the rest of stripe units, including the parity stripe unit, from the stripe instead of waiting to finish the ongoing GC process in the victim device
3. Recalculate the failed data on the victim device by XORing the remaining chunks in the stripe.

For a real-time read operation, the mechanism simply avoids sending a command to the victim device. For example, assume that there is an active GC process within Device 3 when a read request for $D_2$ and $D_3$ is assigned, as shown in Figure 5.1. Normally, to read the data from Device 3, the request has to wait until Device 3's GC process completes. However, the GARM is aware of the ongoing GC process in Device 3 and therefore does not send a read command to this device. Instead, it reads the rest of the stripe units ($D_1$, $D_2$, $D_4$) including the parity data ($D_P$) and calculates the data $D_3$ by XORing $D_1$, $D_2$, $D_4$, and $D_P$. This operation has only the calculation overhead, which is negligible. The mechanism offers higher performance and guaranteed access

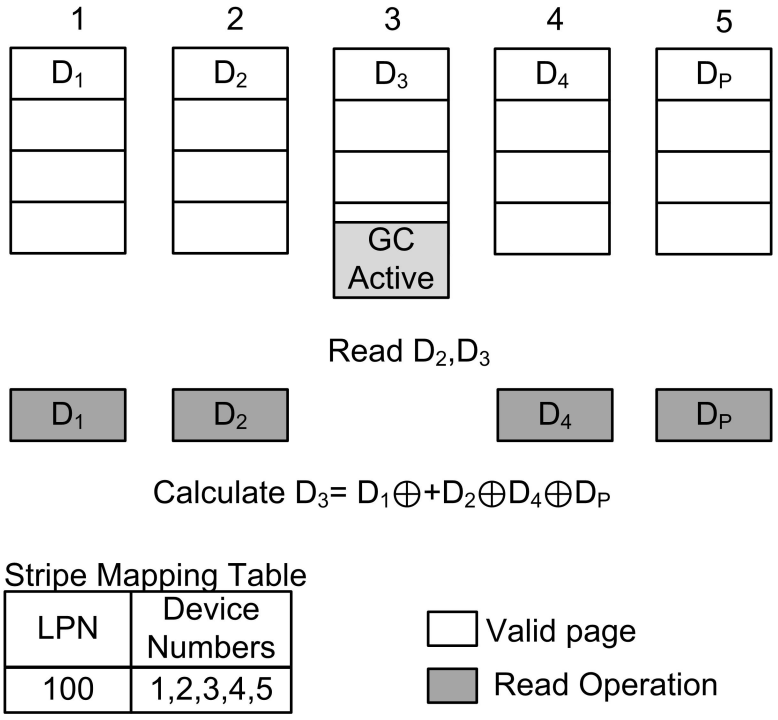



Figure 5.1: Real-time read operation

times for a read operation even when there is an ongoing GC process in the array. This technique can be applied to both random and sequential read operations.

Although the mechanism provides deterministic response time for read operations, it is not applicable if multiple garbage collectors are activated in the same time period. This is because the parity technique used in the reliability is mechanism only able to recover a single blocked data at a time. Therefore, a novel serialised GC feature is introduced where it is guaranteed that there can be only single garbage collector activated at a time in the memory array where each device has its own garbage collector.

### 5.2.3 Serialised Garbage Collection

The garbage collection process has a significant impact on the response time of the storage. There are many ways to implement GC operations over a flash RAID array. To achieve guaranteed performance, the serialised GC technique is proposed.

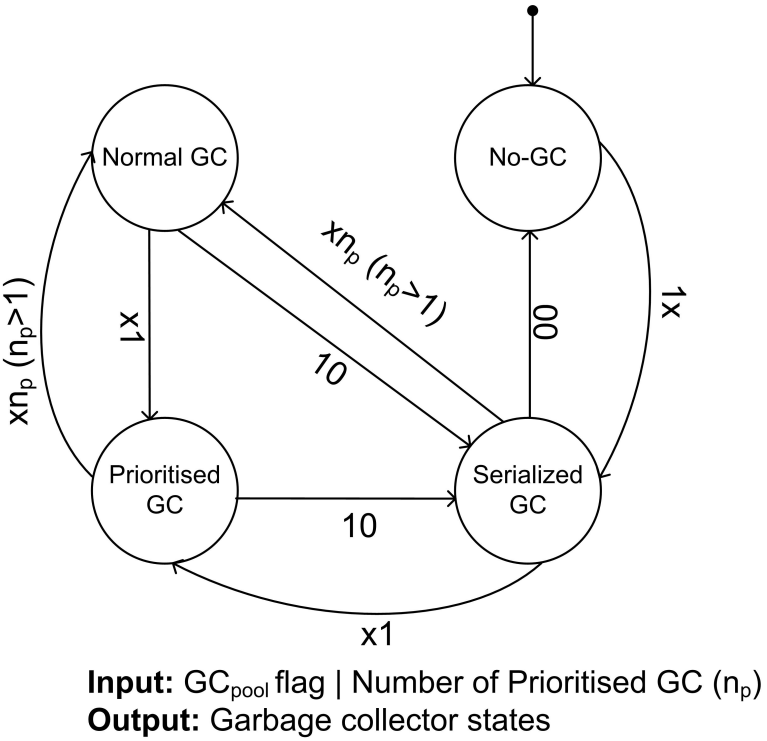



Figure 5.2: State diagram of serialised GC

The RAID mechanism keeps a parity bit per stripe to recover data in case of data failure in RAID 5. The benefit of the redundant data is utilized to eliminate the non-deterministic feature of the GC process. The main idea behind the technique is that there can be only one garbage collector active in the array of flash-based devices in order to combine the benefit of redundant data with random I/O operations. The serialised GC ensures that only a single garbage collector is activated in the array at a time. It globally schedules garbage collectors across the array to achieve this target. While providing this, it has to make sure that none of the devices fails due to lack of free pages as a consequence of serialised cleaning policy. Therefore a number of GC states are introduced as showed in Figure 5.2. It is shown in the form of a finite Moore state machine (same model is adapted for each state diagram throughout the thesis). The system can be in one of four states of GC:

- **No GC:** Garbage collector is not activated in any device.
- **Serialised GC:** Garbage collector can be enabled but it is only allowed for a single device at any given time in the array.
- **Prioritised GC:** Serialised GC is enabled but another device has higher priority for cleaning.
- **Normal GC:** GC can be served on multiple devices in the array at the same time.

Traditional GC techniques on RAID architecture usually have two states: No GC and Normal GC. To switch state from No GC to Normal GC, each device in the array internally initiates its garbage collector once the number of free blocks exceeds a predefined threshold or idle time is detected. This threshold is used to change state from No GC to serialised GC, and is referred to as the soft threshold. In the design, whenever the number of free blocks on a device decreases below the soft threshold, it is added to the GC pool ($GC_{pool}$). The $GC_{pool}$ flag is set high and the state is changed to the serialised GC state. Devices in the $GC_{pool}$ are cleaned in such a way that only one device permitted for cleaning. Devices in the $GC_{pool}$ has same probability of being cleaned. If there are no devices left in $GC_{pool}$ after employing serialised GC state the system changes back to *No GC* state.

However, the situation might arise where more than one device has to activate the garbage collector during same time period. To prevent the system deadlock in this scenario, two more threshold levels are defined. The first, the hard threshold level, indicates that the level of the free space in the corresponding device is critically low. For example if only a single device reaches the hard threshold then the state changes to the *prioritised GC* state. In this state more time is allocated to clean it up by increasing its priority level. This state is quite useful for RAID arrays where a member of the array is expected to be cleaned more frequently than others due to its parity ratio, such

as the reliability mechanisms and RAID 4 where parity distribution percentages are (80,19,1,0,0) and (100,0,0,0,0) respectively.

Moreover, the circumstance might arise where there are multiple devices simultaneously exceeding the hard threshold. In this case, the system changes state to *Normal GC*, where multiple GC is allowed in parallel, as in the conventional technique. After producing enough free pages the state changes back *Serialised GC* or *prioritised GC* depending on the number of devices which exceed the hard and soft thresholds.

The serialised GC is only initiated when there is more than a single device in the $GC_{pool}$, which indicates the devices need to be cleaned, though not urgently. Unless multiple devices request GC at the same time, the GARM operates as a normal reactive GC where there is no shift (suspend) operation required for a single GC. The controller checks the $GC_{pool}$ for every page-consuming request. If there are multiple devices from the array in the pool, then the controller schedules all GC requests according to their priority levels. This operation is iterated until a device has a large enough number of free blocks. Then, the device is removed from the pool and the controller reschedules GC processes with the devices that still remain in the pool. If there is only one device left in the pool, then the system goes back to the No GC state.

### 5.2.4 GC-aware Random Writes

When there is a random write request, the RAID controller does not allocate all the devices of the array to serve the request. For random writes it is guaranteed that there will always be a device that will not be busy with the write operation. GC-aware random write is presented to utilise this feature to eliminate the overhead of the uninterrupted GC process for random writes.

The technique categorizes the random write operations into two main groups: a new random write and an update operation. A new random write refers a random write

that targets a free stripe in the array. An update operation indicates a write operation which targets fully or partially filled stripes.

**GC-aware New Random Write**

The effects of the GARM with new random write operations for a given flash array are examined with respect to time. The response times of two different random write operations, ($D_x$ and $D_y$), are examined over both a standard mechanism (non-GARM) and GARM. The mechanisms are evaluated under two different scenarios including single and multiple ongoing GC processes.

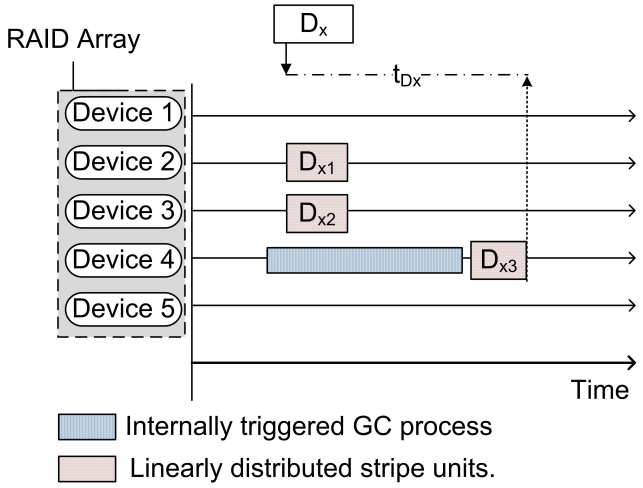



Figure 5.3: Random new write operations with an ongoing GC on non-GARM

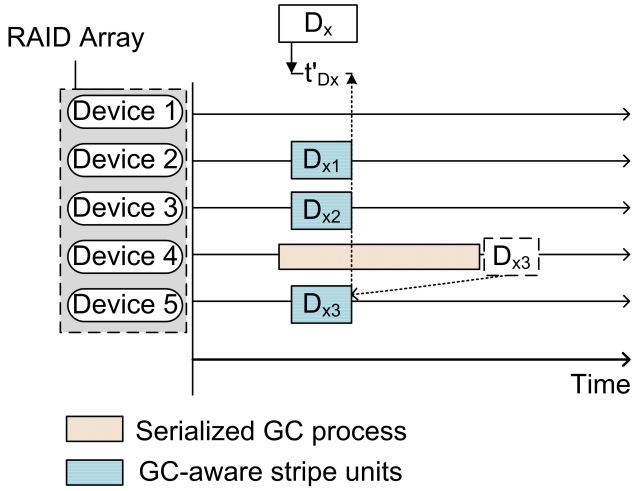



Figure 5.4: Random new write operations with an ongoing GC on GARM

In the first case, a random write operation $D_x$ with three stripe units is given to RAID array with 5 devices. The response time of the operation, ($t_{Dx}$), becomes quite expensive due to the latency of the stripe unit, ($D_{x3}$), which is targeting Device 4 with an ongoing GC process in the RAID mechanism, as shown in Figure 5.3. In the example, the system's resources are not effectively utilized because of the lack of communication between the FTL layer and the RAID controller. However, the GARM cost-effectively locates the overlapped stripe unit across one of the available devices in the array, as shown in Figure 5.4. With the GARM, it can be clearly seen that the access time of data $D_x$, ($t'_{Dx}$), is significantly improved and became deterministic by dynamically locating the overlapped stripe unit ($D_{x3}$) to a GC-free device of the array (Device 5). It also ensures that a device does not contain a multiple member of a same logical stripe to maintain reliability.

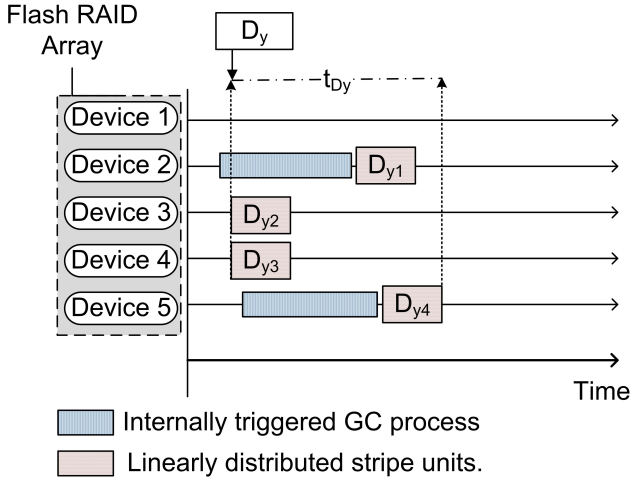



Figure 5.5: Random new write operations with multiple ongoing GCs on non-GARM

In the second example, write data ($D_y$) is delayed due to two GC processes by Device 2 and Device 5, as shown in Figure 5.5. There are two overlapped stripped units, ($D_{y1}$ and $D_{y4}$). The GARM can relocate an overlapped stripe unit to a GC-free device with the help of the serialised GC. The GC process on Device 5 is shifted to a later time when the GC process has completed in Device 2 as depicted in Figure 5.6.

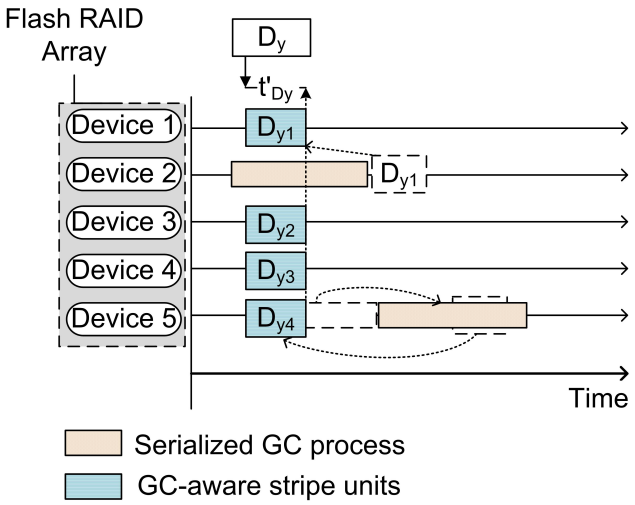



Figure 5.6: Random new write operations with multiple ongoing GCs on GARM

Therefore, overlapped stripe unit $D_{y1}$ can be relocated to an available device (Device 1) and $D_{y4}$ is guaranteed not to be blocked by an ongoing GC process since the GC operation on Device 5 is shifted. As a result, the response time of the data $D_y$ ($t'_{Dy}$) is significantly improved with the GARM.

**GC-aware Random Update Operation**

A partial update operation to a stripe lead parity recalculation process in parity-based RAID systems as explained in Section 2.5. The GARM considers usage patterns (partial or full) in the targeted stripe for a real-time update operation.

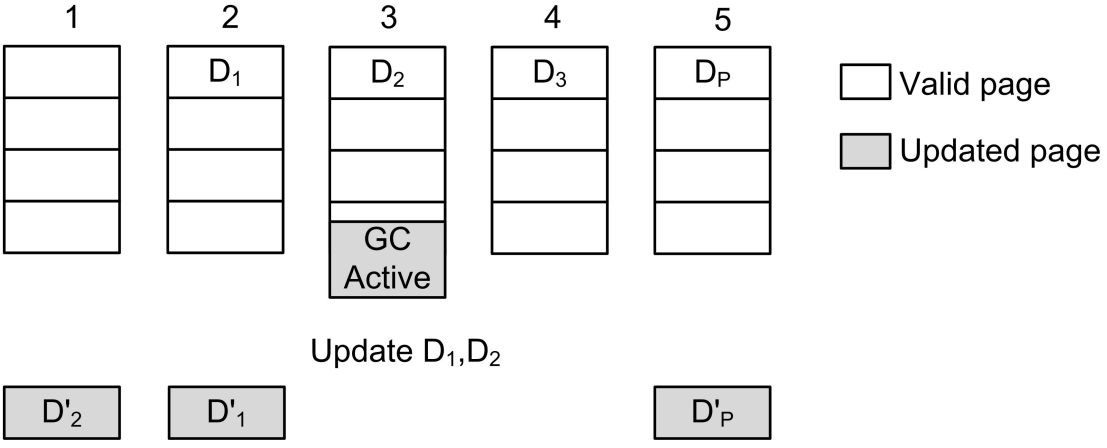



Figure 5.7: GC-aware random update operation for a partial stripe

Firstly, real-time update for a partial stripe operation is investigated. Figure 5.7 illustrates the changes when the host sends a random update request to stripe unit $D_1$ and $D_2$. The GARM first checks whether the target devices for the update operation have an active GC operation, or otherwise. Here, the target device of data $D_2$ has an active GC process. In this instance, the mechanism allocates a GC-free device for the update request $D_2$ instead of waiting to finish the expensive GC process. To assign a device for the overlapped stripe unit, the controller checks the stripe mapping table to find an appropriate device which does not contain any member of the stripe; the usual update operation is then started. The controller reads $D_3$ to calculate new parity $D'_P$. Then it writes the new data $D'_2$ to Device 1, the new data $D'_1$ to Device 2 and an updated parity $D'_P$ to Device 5 while invalidating $D_1$, $D_2$ and $D_P$. Also, the stripe mapping table is updated according to the changes in device numbers of the stripe.

To invalidate the overlapped stripe unit without accessing its physical page location, the benefits of the page status table is used which is stored in NvSRAM memory. Existing FTL techniques usually store status information about a page in its metadata area. To invalidate a page, physical access to flash memory is required. However, the technique described here stores a part of the metadata in NvSRAM to keep the status of all pages in the array and thus physical access to the invalidated page is not required.

Table 5.1: Changes in the page status table in NvSRAM for a random update operation

| LPN | Device ID | Status (Before) | Status (After) |
|---|---|---|---|
| 100 | 0 | 00 | 10 |
| 100 | 3 | 10 | 11 |

In the system architecture, NvSRAM is partitioned into n sections where n is the number of devices in the array. 2-bits in NvSRAM are reserved for each page in the array. The least significant bit represents whether a particular page is a valid or invalid, and the second bit indicates if it is free to use or not. The status of $D_2$ is illustrated

before and after the update operation in table Table 5.1. The update operation for a partial stripe can be employed without any performance overhead.

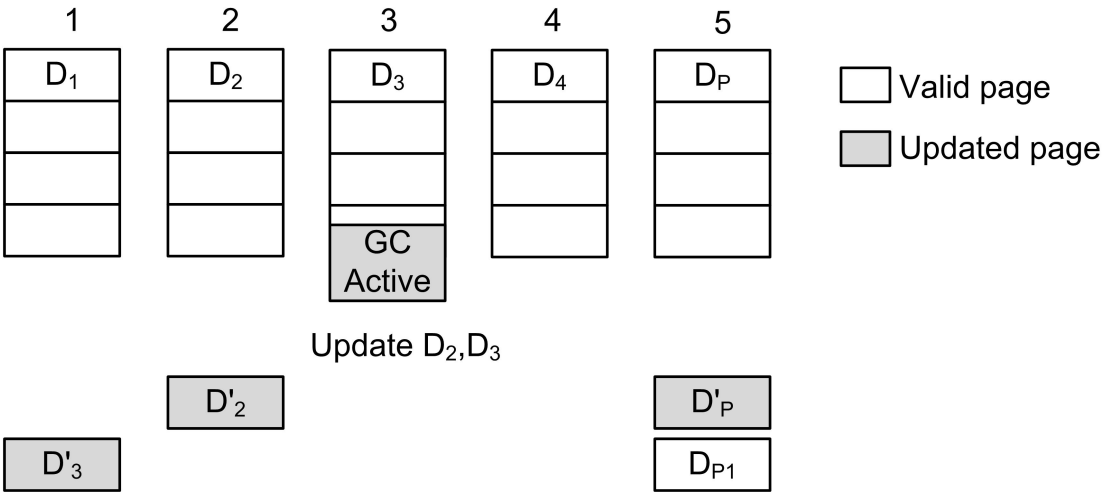



Figure 5.8: GC-aware random update operation for a full stripe

A random update operation over a full stripe is depicted in Figure 5.8. A scenario is considered where one of the target devices of the update request is engaged with a GC process. Also, there is no available device to relocate the overlapped stripe unit as all devices in the array already store a member of the stripe. To overcome this, the overlapped stripe behaves as a new random write.

For example, let us assume that there are five flash devices in the array and a GC process is active in Device 3 when the update request arrives. First, the controller calculates new parity $D'_P$. Then stripe data $D'_2$ and the updated parity are stored in Device 2 and Device 5, respectively. Since the Device 3 is blocked by an ongoing GC process, the controller locates the $D'_3$ to a new device. Moreover, the overlapped stripe unit ($D'_3$) behaves as a different random write because a device cannot retain more than a single stripe unit for a logical stripe. The data $D'_3$ is placed on Device 1 and its associated parity is updated accordingly on a different stripe. This operation causes an extra write operation ($D_{P1}$) and results in the performance overhead of an

update operation. However, compared to a long GC process which includes an erase operation, this most certainly has the lower impact on system performance.

### 5.2.5 GC-aware Sequential Writes

Until now, it has been assumed that a given workload only consists of random writes. A sequential write to a flash array updates the whole stripe to prevent creating a wear imbalance and thus reducing the reliability of the system. Parity blocks wear the same as data blocks in the case of sequential writes. However, random writes have the ability to imbalance write traffic across the devices of a flash array. This provides an opportunity to differentiate the age of the devices in the array to prevent simultaneous wearing. It may be noted that maximum reliability is provided with a workload which only consists of random writes [8]. It is also known that many realistic workloads are dominated by random writes [86].

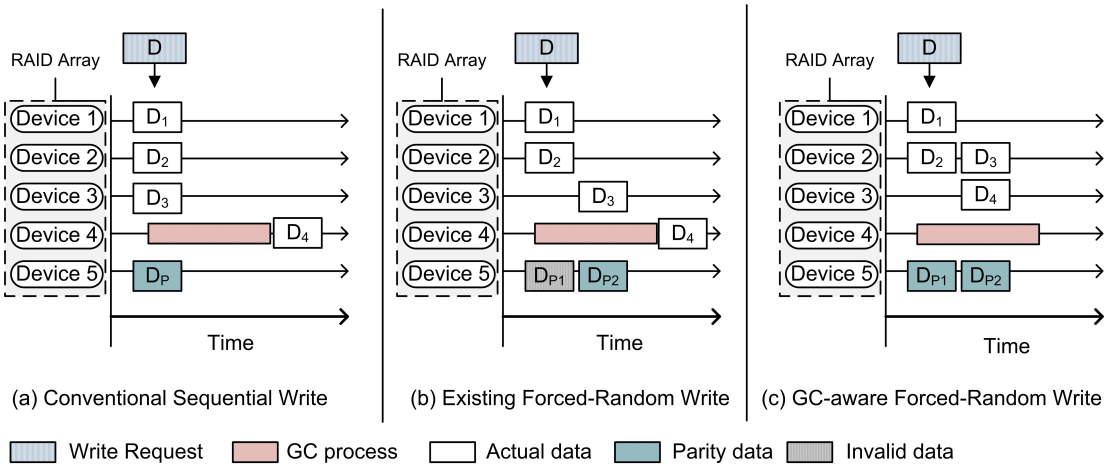



Figure 5.9: Comparison of sequential write and forced random write techniques in a flash-based array with an ongoing GC

This mechanism offers a guaranteed response time for random writes and any type of read operation, as discussed in the previous section. To utilise the benefit of random writes, previous work in the literature has discussed a technique which converts

a sequential write into random writes [87]. This technique is adapted with GARM by relocating partitioned random writes onto GC-free devices.

Figure 5.9 illustrates an example of a sequential write, an existing forced-random write and a GC-aware forced random write with an ongoing GC process in figure sections a, b and c, respectively. The aim of the existing forced random write technique is to create a wear-imbalance among the devices in the array for new and update data requests. As illustrated in Figure 5.9(b), there is an extra write operation to the parity device (Device 5) compared to the sequential write in Figure 5.9(a).

In the existing forced random write approach, it is assumed that the sequential write is divided into two equal random writes. First, two data stripes ($D_1$ and $D_2$) and a partial parity data ($D_{P1}$) are written to same stripe. Second, data $D_3$ and $D_4$ are written with full parity knowledge ($D_P$) of $D_1$, $D_2$, $D_3$ and $D_4$ while invalidating the partial parity ($D_{P1}$). Although the existing forced random write technique offers enhanced reliability compared to the sequential write, it still suffers from non-deterministic access latency due to ongoing GC in Device 4.

Therefore, the existing forced random write technique is further improved for adoption in the GARM. With the knowledge of the ongoing GC process, the second part of stripe D is located across one of the GC-free devices as illustrated in Figure 5.9(c).

In the example, Device 2 stores multiple data stripes which belong to same stripe. To be able to reconstruct data D in case of failure in Device 2, the mechanism needs to keep two partial parities ($D_{P1}$ and $D_{P2}$) for stripe D. Also, all indexing information is updated in the stripe mapping table. The GC-aware forced random write scheme not only increases system reliability but also provides guaranteed access times for sequential workloads.

### 5.2.6 WCET Analyses

One of the bottlenecks of flash-based storage is performance variability. When a request experiences a long latency due to a GC process, the response time will be non-deterministic, as explained in Chapter 2. The technique facilitates significantly reduced WCET compared to the existing solutions described in the literature. In this section, the worst-case response time of the GARM is analysed.

Table 5.2: WCET comparison of existing mechanism with GARM

| **Technique** | **WCET** |
|---|---|
| GFTL [16] | $t_{er}$+*max* {U($e_w$), U($e_r$)} |
| RFTL [18] | *max* {$t_{er}$+ U($e_w$), U($e_r$)} |
| PGC [17] | $t_{er}$+*max* {(U$e_w$), U($e_r$)} |
| GARM | *max* {U($e_w$), U($e_r$)} |

The proposed mechanism is examined using the same terminology used in the existing literature for consistency [16–18]. U($e_w$) and U($e_r$) denote the upper page write and page read bounds on the flash memory, respectively. These variables depend on several factors, such as the type of address mapping table used, the method by which metadata is stored, and the existence, or otherwise, of the buffer.

The WCETs are compared in Table 5.2 through various techniques. There are two important states that need to be considered. The first is where the WCET of the mechanism is of serialised and prioritised GC states, where only one GC is enabled at any given time over the RAID array in order to relocate an overlapped stripe unit. If the system changes state to Normal GC due to a lack of free pages, then the WCET of the mechanism would be increased. Existing techniques [16, 17] do not take this scenario into account, thus the WCET of the mechanism presented has been analysed over all GC states except Normal GC. Second, it should be noted that previous FTL mechanisms have only proposed solutions at the single device level. The work reported herein is, to the best of my knowledge, the first and only study that investigates the

WCET of flash-based storage at the multi-device level. The benefit of multi-device architecture is utilized with RAID.

The underlying principle behind existing techniques is to partition or preempt the GC process. As the GC can only be portioned/preempted at the granularity of the atomic operations of the flash memory, none of the existing techniques propose solutions to avoid the most expensive operation, that of erase. In the worst-case scenario, a request would have to wait for an erase command to finish, which is nearly 10 times more expensive than a write operation. As showed in Table 5.2, all WCETs include expensive $t_{er}$ except for GARM.

The GARM significantly reduces the WCET by ensuring that none of the instructions is blocked by an ongoing GC process. Therefore, all requests can be directly serviced, with a cost of only $U(e_w)$ or $U(e_r)$.

### 5.2.7 On-line Parity Migration

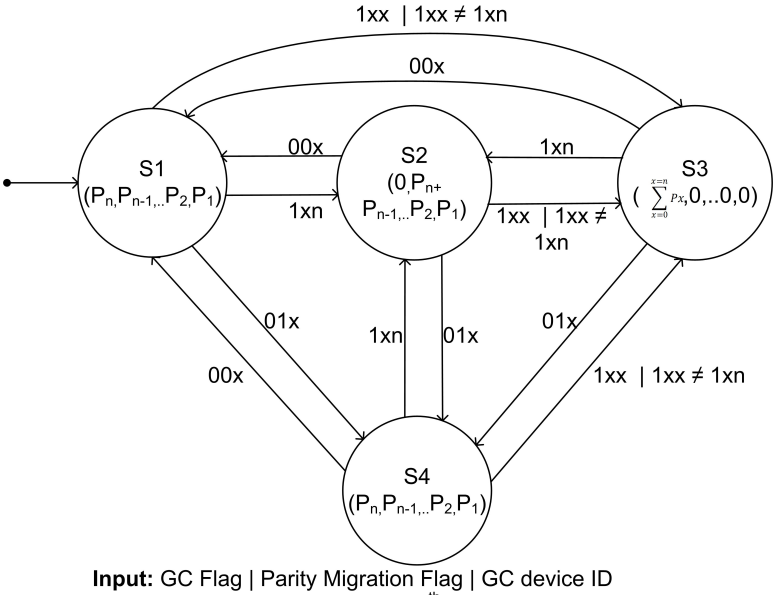



Figure 5.10: Parity migration states

While the GARM offers guaranteed performance for incoming requests, its use is not well-suited to existing reliability mechanisms, where the ages of the devices in the array are strictly controlled by initially assigned parity distribution percentages. When there is no garbage collector active, the GARM does not disturb these initially assigned parity percentages; However, the proposed mechanism redirects incoming I/O requests to GC-free devices in the case of an active GC in the target device of an array, which results in degenerate parity distributions. Since the most aged device performs more GC then the other devices, its default parity percentage decreases whilst that of the others increases.

It is critical to maintain the system's reliability while guaranteeing access time for high reliability applications. To control parity distribution percentages, four main parity migration states will now be presented. Parity migration states are illustrated in Figure 5.10. The system can be in one of the following parity distribution states:

- **S1:** Garbage collector disabled and parity migration not required.
- **S2:** Garbage collector enabled on the most aged device. All its parities are redirected to the second most aged device.
- **S3:** Garbage collector enabled on any other device. All their parities are redirected to the most aged device.
- **S4:** Garbage collector disabled but parity migration is required.

The states are defined with inputs of *GC flag, parity migration flag and GC device ID*. The *GC flag* refers to the cleaning status of the storage; it becomes high if the garbage collector is activated in any device in the array. The *GC device ID* indicates the device number that is cleaning at any given moment. Finally, the *parity migration flag* shows whether a parity migration operation is required (1) or not required (0). Outputs of each state produce different parity distribution percentages.

Conventional SSD-based RAID mechanisms have only one state, S0, where stable parity percentages ($P_n$,$P_{n-1}$,..,$P_2$,$P_1$) are assigned and not changed until a device replacement process is required, where $n$ is the number of devices in the array. In order to take the control of the migration of the parities in case of an active cleaning process, two more states are introduced: S2 and S3. S2 indicates that the most aged device is performing a cleaning operation, and S3 indicates a cleaning process on any other device in the array. Furthermore, to reinstate the current parity levels to the default values (80,19,1,0,0), an on-line parity migration (OPM) is introduced in state 4.

If the *GC flag* becomes high for the most aged device (*GC device ID=n*) in the array, incoming parities targeted at the most aged device (n) are then redirected to the second most aged device (n-1). In this case, the parity distribution percentages of the S2 can be assigned as (0,$P_n$+$P_{n-1}$,..,$P_2$,$P_1$). Otherwise, if the cleaning process is required on any of the other devices in the array, then all parities targeting those devices are redirected to the most aged device during the cleaning period to reduce changes in the default parity distribution. Although the state of S3 helps to recover the parity ratio of the most aged device back to its desired level, it may not fully recover this percentage. Therefore, an additional state is introduced: the on-line parity migration state, S4.

The changes in parity distribution are usually led by the GC of the proposed mechanism. Therefore, after each serialised GC period, the OPM checks the parity distributions on the devices. If these amounts are not comparable with the initial distribution percentages, then the *parity migration flag* becomes high and the OPM is activated.

The OPM first calculates the amount of parity data that needs to be migrated and determines the devices which have different parity amounts than their default values. The migration progress is operated from a source device, which has extra parity than its default percentage, towards the target device, that has less parity than its initial value.

The parity migration of OPM is investigated under all cases of a target stripe. A stripe can be in one of the following states: free, partially or fully occupied by valid

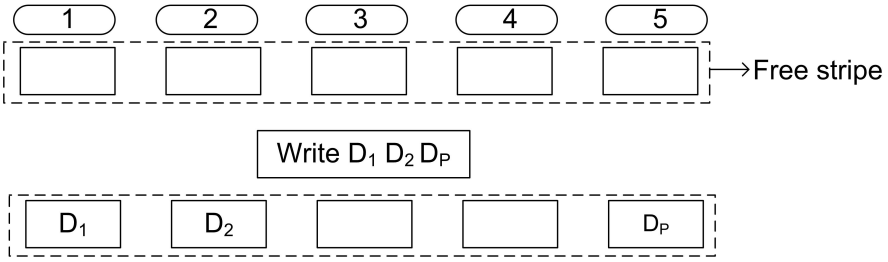



Figure 5.11: OPM with a free stripe

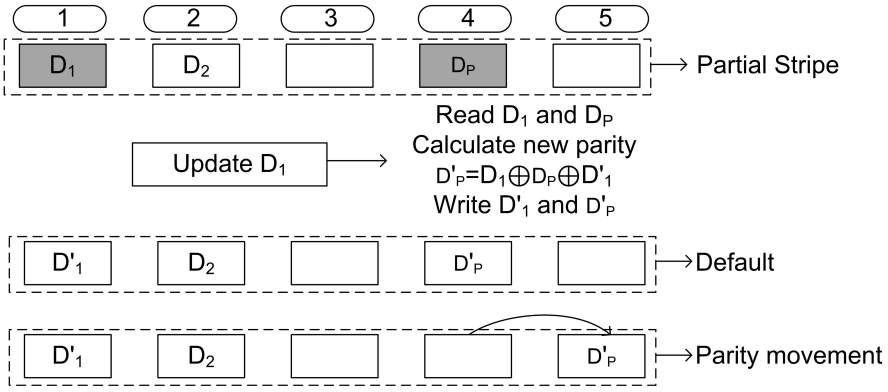



Figure 5.12: OPM with a partial stripe

data. Whenever a write operation arrives at a free stripe when the system is in state S3 or S4, the OPM relocates the parity data on the target device as shown in Figure 5.11. In the second case, if a write operation targets a partial stripe and the target parity device is free, then the OPM invalidates the old parity data and relocates new parity on the new target device as shown in Figure 5.12. These operations do not cause any additional time delay to write requests.

In the worst-case scenario, however, if the targeted parity device contains a data block which is associated with parity and if the data is not updating this parity, then the data and new parity need to be swapped, as illustrated in Figure 5.13. This operation costs an additional copy (read+transfer+write) latency to the write request. The OPM continues until the *parity migration flag* becomes low, when all required parities are migrated, or the *GC flag* becomes high.

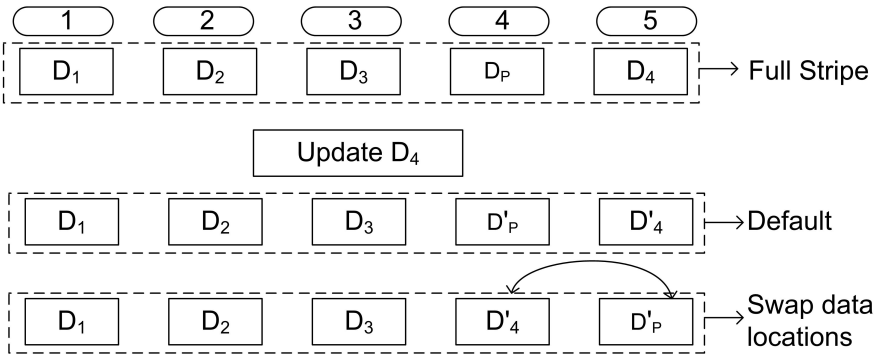



Figure 5.13: OPM with a full stripe

## 5.3 Summary

In this chapter a novel real-time GC-aware RAID mechanism was proposed. Unlike existing published techniques, the proposed mechanism completely avoids performance variability caused by garbage collection in flash-based RAID architecture. This indirectly improves Worst Case Execution Time and performance of the system.

The proposed mechanism was discussed under all possible I/O scenarios including random and sequential write / read operations over free, partial and full stripes of array. To read data from the device with active garbage collection, an intelligent read feature was introduced which uses the parity data to rebuild blocked data due to garbage collection. For random write operations, the proposed new feature of dynamic allocation policy, dynamically allocates the stripes of the incoming data to GC-free devices in the array. For sequential writes, where all elements of the array need to be accessed, a new feature of enhanced forced random write was proposed which converts sequential writes into random writes and dynamically locates them over the array.

Another issue in case of simultaneously multiple cleaning processes in an array, the existing architectures cannot serve real-time read requests as parity data only capable to rebuild single blocked data at a time. Therefore, a new feature of serialised garbage collection was introduced in the proposed architecture, which ensures single

garbage collector active at any time. This feature provides a number of different states to globally organise GC operations across the array.

To maintain the reliability mechanisms while offering deterministic response time, a new feature of on-line parity migration was proposed. The age control techniques of the reliability mechanisms are defined by initially assigned parity percentages for each device. Therefore, the on-line parity migration keeps the initially assigned parity ratios among the devices in an array and thus the reliability mechanism is maintained.

In conclusion, the proposed garbage collection-aware RAID mechanism provides a number of novel features to address the real time and reliability concerns of the flash memory for high integrity systems.

# Chapter 6

# On-line Device Replacement Techniques

This chapter presents techniques that allow for the replacement of aged devices in an SSD array in such a way that the system provides continuous system reliability and deterministic response times during the replacement process. The performance overhead of the reconstruction process is also improved using a novel data migration policy. The techniques also consider device replacement time and utilization of redundant data in old devices to improve the device replacement time and system performance further.

## 6.1 Introduction

In previous work, a novel RAID-based architecture was presented to enhance the reliability of an SSD storage system by mitigating the problem of rendering the whole array unreliable at the same time [11]. It achieves this by guaranteeing wear imbalance between each component in the array and thereby ensuring they do not reach their endurance limits simultaneously. This is achieved using two primary techniques. The first is an uneven parity distribution that ensures erases across components are

distributed unevenly, and the second is a device copy/swap algorithm that moves data around and manages lifespan as components reach their endurance limits. However the limitation of this architecture is that this copy/swap operation requires the array to be taken off-line. Whilst this mechanism significantly enhances the reliability of the storage, it restricts its usage in hard real-time systems as it is not able to serve requests during the replacement period. The extent of the downtime depends on the amount of data and parity that needs to be moved. Furthermore, it does not consider flash-specific operations such as garbage collection and metadata management, both of which may affect the real-time characteristics of a system. The limitations of the device replacement with regards to the reliability mechanisms were discussed in Chapter 3.

The contribution of this chapter is an SSD RAID framework incorporating several novel techniques to improve the efficiency of the replacement process in SSD RAID for hard real-time applications: a proactive hot-swapping technique, a data migration technique that coordinates operations with the garbage collector, and a parity redistribution technique. To utilize the benefits of hot-swapping, a semi-hybrid RAID technique is also introduced that enhances performance when there is no active device replacement.

**Definition 6.1.1** *Write Amplification Factor*

*The write amplification factor is used to calculate the ratio of the additional write operations which are caused due to internal management components of flash memory such as garbage collection, wear levelling and data migration.*

The proactive hot-swapping technique increases system availability and facilitates guaranteed response times as the storage does not need to go off-line while replacing a device. The data migration technique exploits the internal operations of flash memory to reduce performance overheads during device replacement. The parity redistribution technique ensures that parity percentages are redistributed across the devices during the replacement process, reducing the write amplification factor (see Definition 6.1.1)

and device replacement time. Finally, the semi-hybrid RAID technique utilizes valid redundant data on old devices to improve the I/O performance of the system for read operations during once a device replacement is over.

The chapter is structured as follows: Section 6.2 presents the architectural design of the system. Section 6.3 presents proactive hot-swapping, Section 6.4 presents the approach to data migration, and Section 6.5 presents the parity redistribution technique. Section 6.6 presents the semi-hybrid RAID architecture into which these enhancements have been incorporated. A summary of the chapter is given in Section 6.7.

## 6.2 Architectural Design

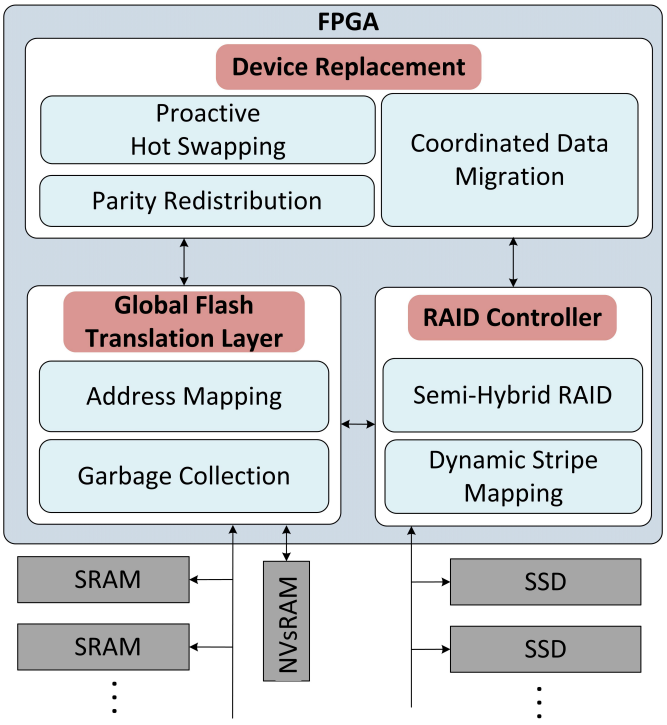



Figure 6.1: Architecture block diagram

The architecture of the system and its internal relationships are illustrated as a block diagram in Figure 6.1 and is based on reference [88]. It consists of the SSDs,

memory components to store metadata information (SRAM and NvSRAM), and the management components. The management component consists of three main blocks.

The first block—labelled Device Replacement—contains the proactive hot-swapping, the coordinated data migration, and the parity redistribution. It monitors the Global FTL in order to make decisions about activating or deactivating these functions. According to the outputs of these functions, the RAID controller effectively relocates both the incoming requests and the partial replacement tasks across the array. Also, it may suspend or reschedule these operations according to their priority levels to avoid possible overlaps.

The second block is referred to as a Global FTL as it takes a holistic view of the whole array, rather than a view of a specific SSD device. The third block—the RAID controller—provides the primary RAID functionality via the semi-hybrid RAID and the dynamic stripe mapping function. The semi-hybrid controller dynamically reconfigures components in the array, and the global view of the array, after device replacement in order to improve read performance. More details will be given about these blocks in Chapter 7.

## 6.3 Proactive Hot-Swapping

If an individual device reaches its endurance limit, and therefore shows a high bit error rate, data integrity becomes unreliable. Therefore, the device is considered expired and in need of reconstruction by the RAID controller. On-line RAID reconstruction generates additional I/O to the storage and so reduces the system's performance. To eliminate the overhead of this on-line reconstruction, a proactive real-time hot-swapping technique is introduced that facilitates on-line migration of the data from the old device to the hot spare device before the old device reaches a critical bit error rate.

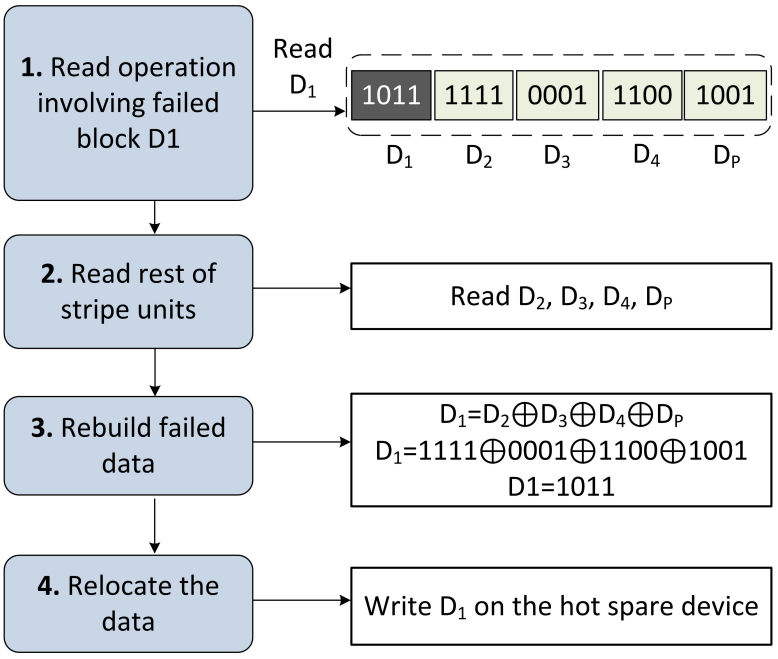



Figure 6.2: Example read operation involving a failed block during rebuilding

Incoming write requests targeting the failed device do not detrimentally affect performance significantly as they are relocated to the hot spare device, and only the stripe mapping table needs to be updated. However, reading data from the failed device causes extra I/O interaction with the storage. For example, Figure 6.2 illustrates a logical stripe with a failed data D1. A read operation to D1 triggers the following steps:

1. The read operation $D$ arrives and metadata indicates that $D_1$ has failed;
2. Read $D_2$, $D_3$, $D_4$, and parity $D_P$;
3. Calculate $D_1$ using $D_2 \oplus D_3 \oplus D_4 \oplus D_P$;
4. Write reconstructed $D_1$ to the hot spare device.

When a data read request targets the failed device, it costs $n-2$ more reads and an extra write operation (where $n$ represents the number of devices in the array), thereby slowing down the average response time. In order to reduce this overhead, the proactive technique starts data migration before the old device reaches its endurance limit.

A device with proactive hot-swapping can be in one of the following states:

- **S0:Enable**: The device is accessible to all I/O instructions.
- **S1:Read Only**: The device is only accessible to read instructions.
- **S2:Disable**: The device is not accessible to any I/O.

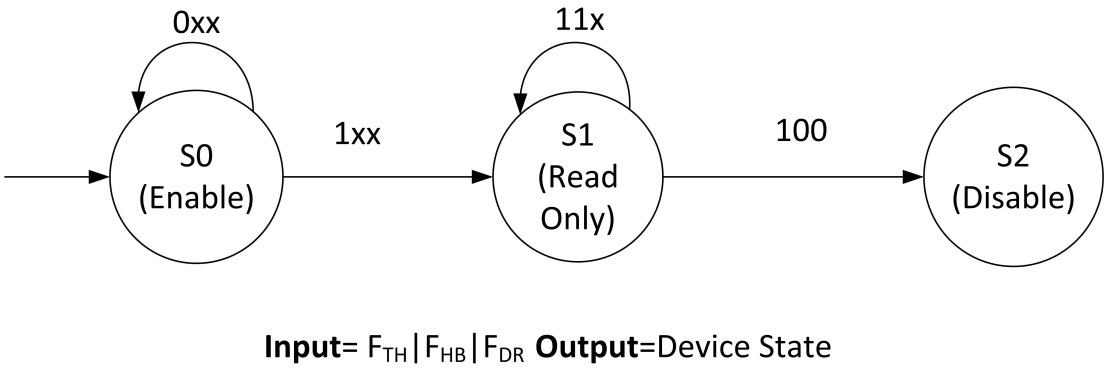



Figure 6.3: State transactions of a device in the hot-swapping technique

Conventional hot-swapping techniques have only adopted state S0 and S2. Initially, all devices are configured to be in a state of *enable*. To change the state of a device from *enable* to *disable*, its ageing level, which is proportional to the number of erasures performed, is considered. Unlike existing techniques, the new technique introduces the *read only* state, as shown in Figure 6.3. A number of input flags are considered to change the state of devices including $F_{TH}$, $F_{HB}$, and $F_{DR}$. $F_{TH}$ becoming high when the corresponding device crosses the triggering threshold for hot-swapping. The hybrid flag ($F_{HB}$) indicates whether the old device will participate in the new RAID configuration for the semi-hybrid RAID or not. Finally, the device replacement flag ($F_{DR}$) shows if the device replacement process is active or not.

To change the state from *enable* to *read only*, only $F_{TH}$ is considered. If $F_{TH}$ become high (1), then the device is marked to be swapped and the device replacement

flag ($F_{DR}$) is set as high (1). After migrating all valid data in the old device, $F_{DR}$ becomes low (0) and the technique checks the hybrid RAID flag, ($F_{HB}$). If it is high then the device always stays at the *read only* state, otherwise its state changes to *disable*.

When a given device is in the *read only* state, the controller only permits read instructions to this device. Any other life-consuming operations—specifically write and update—are redirected to a hot spare device. A read operation on the old device can be served from the old device itself without rebuilding and migrating, and therefore does not introduce a performance overhead.

**Hot-Swapping Triggering Threshold**

The BER of flash memory-based storage grows proportionally to the number of erasures that storage has performed, as discussed in Chapter 2. The effects of BER are reduced by introducing *wear levelling*, which evenly spreads the wear (and BER) across the memory. SSDs basically use ECC—which are usually implemented in hardware—to correct bit errors. Even after applying ECCs, there might be some error bits left in the memory, which are called the Uncorrectable Bit Error Rate (UBER). A device reaches its endurance limit when the UBER reaches an unacceptable level. [8, 11].

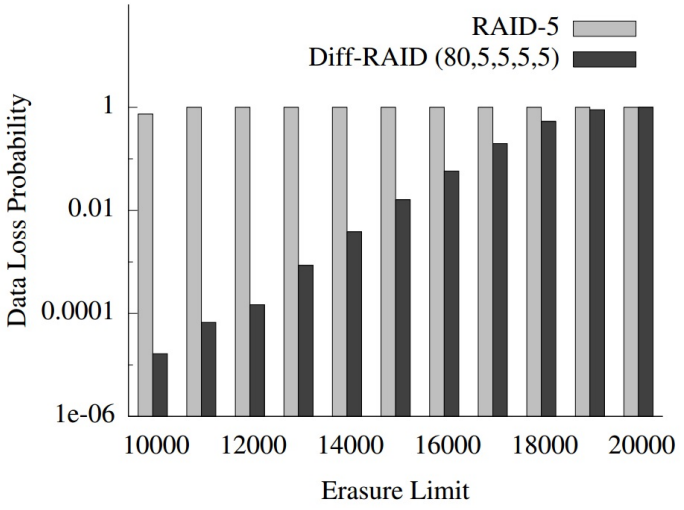



Figure 6.4: Comparison of erasure limits of SSD RAIDs with a 4 bit ECC taken from reference [8]

The reliability techniques prolong the lifetime of the storage by providing additional redundancy, as shown in Figure 6.4. It is clear that the life of SSDs with Diff-RAID is extended a few 1000s of cycles without a high ratio of data loss probability. The ratio of the data loss probability is considered as the metric by which to trigger the hot-swapping technique, which is also proportional to the number of erasures performed. A level of 0.0001 and under can be tolerable for the application within the scope of the research [8]. Therefore, it is defined as the triggering threshold for hot-swapping. There is an obvious trade-off between the system reliability and the lifetime. Starting hot-swapping at an earlier point will reduce the data loss probability, whilst at the same time reducing the average lifespan of the corresponding device.

The hot-swapping trigger threshold is also important for system performance. A late triggering point would result in worse system performance as the BER/UBER rate also proportionally rises. To correct these errors, either ECC or RAID reconstruction would need to be performed. However, the performance overhead of the correction of bit errors by ECCs can be ignored as they are rapidly handled at hardware level [89]. Moreover, the ratio of the UBER which triggers the expensive RAID reconstruction are relatively low in their overall effect on system performance.

**Completion Time**

The completion time of on-line rebuilding is important in terms of the reliability of the system. This is because if an additional device failure happens while the rebuilding is underway, it leads to non-recoverable data loss. To reduce the completion time, an idle time detection approach is commonly used, triggering the reconstruction process when idle or low density usage patterns are detected or predicted.

The presented technique reduces this probability of simultaneous failure in multiple devices as it starts the data migration before the given device fails. Because of this, the device replacement time is not as strict as it is in other on-line RAID reconstruction mechanisms. However, there will generally be some data which is not updated during

the data migration process (as it has not been accessed) and will have to be migrated before next device replacement operation is underway. To address this, a cold data migration operation is introduced where not only incoming write operations to the old device triggers the data migration, but also the cold data in the old device are partially migrated to the new device based on an idle-time detection approach. Each cold data page migration process costs the sum of the read, write, and transfer latencies of the data. This technique is discussed in the following section in detail.

## 6.4 Coordinated Data Migration

Cold data migration or on-line reconstruction operations are usually triggered when there is an idle time or low density workload pattern detected. The garbage collector is also based on an idle time detection technique, and is a common cleaning approach [33]. The possibility exists that the idle time period is not sufficient for completing both garbage collection and migration operations when the device replacement process has to be completed within a certain period. Therefore, when the migration operation overlaps with an ongoing garbage collection process in the target device, the performance of the system degrades.

In the first case of Figure 6.5, an idle time-triggered on-line reconstruction technique is illustrated on a magnetic disk RAID array. It is clear that to minimise the performance overhead of rebuilding, the device reconstruction tasks (DRTs) are triggered when an idle time period is detected. However, the existing technique does not work well with SSD RAID mechanisms where garbage collection is also based on idle time detection as it slow downs the I/O performance and increases the completion time of the device replacement, as shown in the second case of Figure 6.5. This is because the controller is not aware of internal operations in flash memory-based storage.

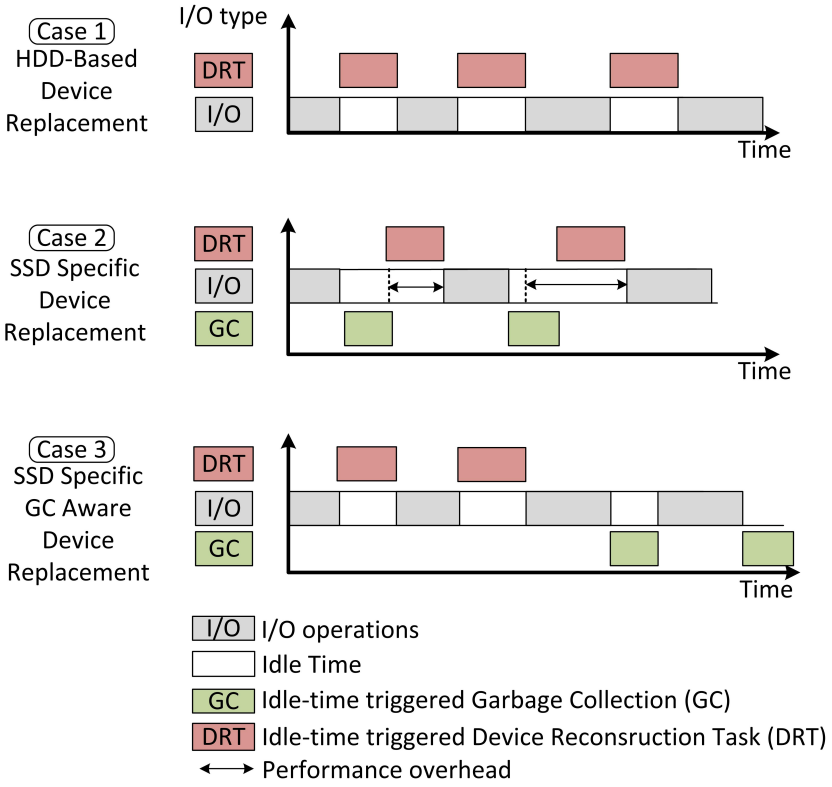



Figure 6.5: A comparison of device replacement techniques

Whenever the technique detects an idle time period, DRTs are inserted in the I/O queue of the corresponding memory. Since the device replacement time must be as short as possible to avoid non-recoverable data lose in case of additional device failure, a higher priority is given to the DRTs over system I/O instructions. However, flash memory-based storage requires regular background cleaning operations, which consume relatively long periods of time. Also, when GCs are initiated then they block all incoming requests to the memory component until they complete. It is obvious that the completion and response time of DRTs, and therefore I/Os, would be affected by an uncoordinated cleaning processes.

A coordinated data migration process is introduced to interleave these overlapping requests. It monitors the ongoing garbage collection processes in the array with the help of the Global FTL. If garbage collection and device replacement functions want

to access the target device during the same idle time period, the technique reschedules them depending on their priority levels to guarantee system response time.

Generally, idle time-based cleaning mechanisms do not initiate a cleaning process due to lack of free space, as in threshold-based mechanisms. Therefore, they have a lower priority compared to that of threshold-based mechanisms and thus shifting cleaning operations to later time periods may not incur the same starvation of free space as in threshold-based cleaning policies.

If the target device is running out of free space, and consequently has to immediately garbage collect, then GC takes a higher priority; otherwise, the DRT maintains a higher priority. Case 3 in Figure 6.5 shows that the technique can considerably enhance the performance of the system by avoiding potential conflicts as no I/O block is inadvertently delayed.

**Idle Time Detection**

Since improving the efficiency of idle time detection is not in the scope of this study, a basic idle time detection function—which is also used in the baseline reliability mechanism in reference [87]— is adopted. To detect an idle time in the storage, the last access time of the corresponding memory is stored in a register. If there is not any request assigned to the memory after a predefined time period this is considered as an idle time and idle time triggered techniques are invoked. If an instruction arrives before this time period elapses, then the system does not invoke idle time operations and serves the request.

## 6.5 Cost-Effective Parity Redistribution

Unlike traditional RAID levels, SSD-based RAID mechanisms such as Diff-RAID [8] and flash-RAID [11] allow a parity redistribution operation during a device replacement process. These mechanisms require strict control over the parity distribution per-

centages of the devices to maintain the ageing ratio between devices at specific levels. Any unmanaged changes in the percentages affects their reliability mechanisms.

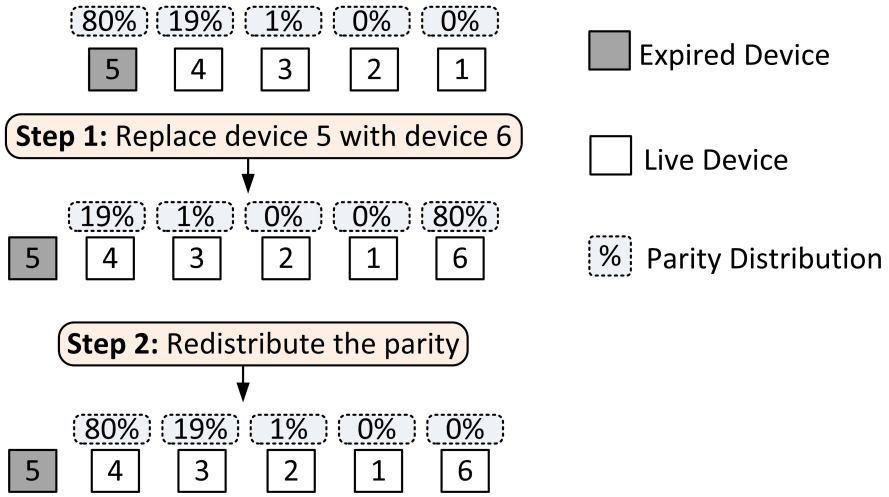



Figure 6.6: Parity redistribution of Diff-RAID

The parity redistribution method in Diff-RAID has two main steps, as illustrated in Figure 6.6. First, the old device (5) is replaced with the new one (6). Since the replacement process migrates (or reconstructs) the data on the new device, all parity data stored on the old device is also migrated. However, this leads to undesired parity distribution percentages in the system—for instance, in this example, (19%, 1%, 0%, 0%, 80%). To restore the parity ratios to the desired levels of (80%, 19%, 1%, 0%, 0%), Diff-RAID redistributes the parity data on the new device (6) to the other devices in the array. However, this operation has two main bottlenecks:

1. It increases the write amplification factor and replacement time by performing extra parity movement to the new device.

2. The parity redistribution step requires additional data movement between the new device and other devices in the array, which further slow downs the performance of the system.

In order to address these problems, a cost-effective parity redistribution technique is presented. The main idea behind this technique is to replace a device without requiring additional data movement among devices by considering the default parity assignment percentages. The technique redistributes the parity data of the old device during the execution of actual writes (hot data migration) or during a cold data migration period along with the proactive hot-swapping.

**Parity redistribution with hot data migration**

The cost-effective parity redistribution technique is compared with the that of Diff-RAID under conditions of hot data migration by considering a number of scenarios. These scenarios differ from each other with regards to the status of target stripe (full or partial stripe) and the location of the actual and the parity data in the stripe.

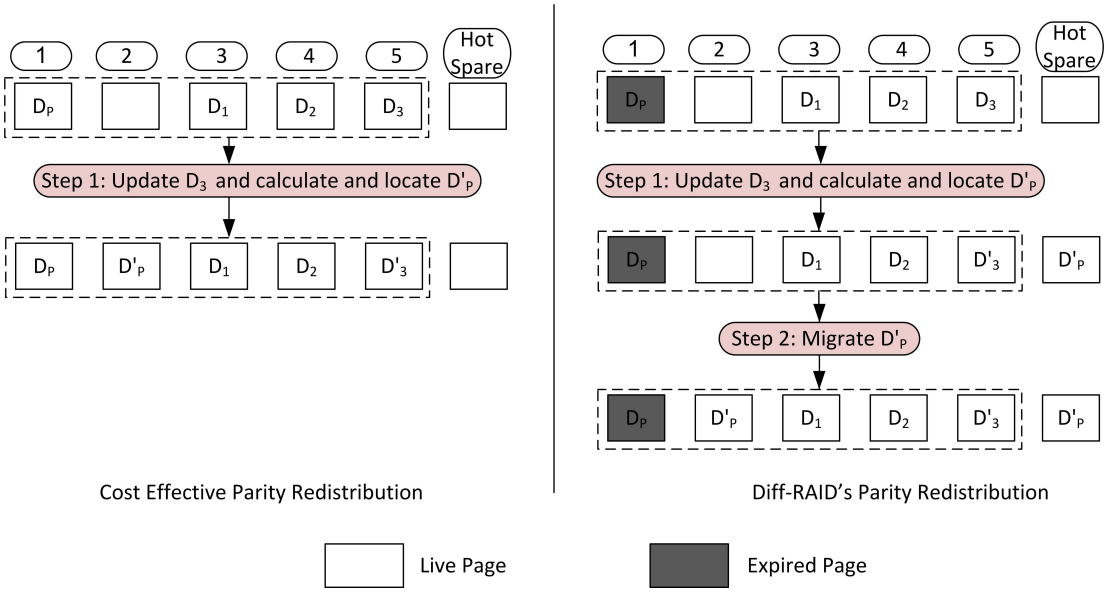



Figure 6.7: Comparison of parity redistribution for a partial stripe

In the first scenario, given an actual write ($D_3$ in Figure 6.7) targeting the old device, the technique does not directly relocate the updated parity ($P'_D$) to the hot spare device. Instead, it determines a target device (device 2) for parity data based on the initially assigned parity distribution ratios. If the target device does not include a member of

the logical stripe, then updated parity ($D'_P$) can locate the target device without any performance overhead, as illustrated in Figure 6.7. However, the same scenario would result in an extra write operation and an additional stripe unit migration with Diff-RAID's parity redistribution technique, as shown in Figure 6.7. This increases both the performance overhead and the write amplification of the system.

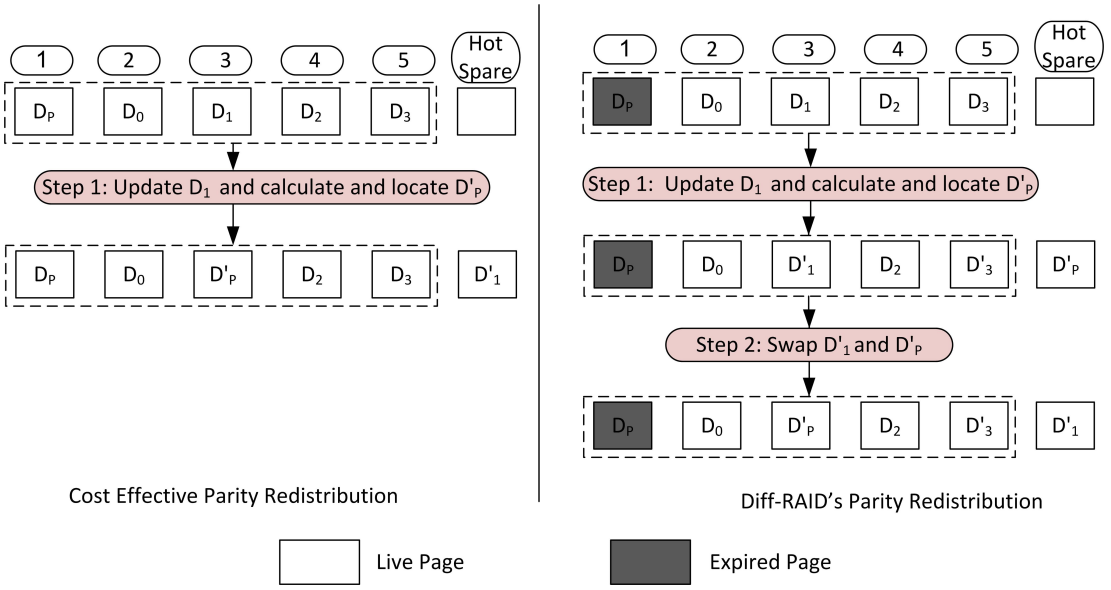



Figure 6.8: Comparison of parity redistribution for a full stripe, Case 1

Secondly, in case of an update operation on data whose parity is located on the old device, the technique performs only a swap operation between the updated parity and the data block, and so does not incur a performance overhead similar to the previous case, as shown in Figure 6.8. A similar scenario results in an additional write ($D'_P$) to the hot spare device in Step 1 and an additional stripe unit swap in Step 2.

In the last case, if a parity needs to be redistributed in the case of a full stripe write then the technique requires two steps, as illustrated in Figure 6.9. This is because to ensure that a single device does not include multiple members of a logical stripe, the destinations for all stripe elements are updated accordingly. In the first step, the cost-effective technique updates the stripe unit ($D_3$) and calculates the new parity ($D'_P$)

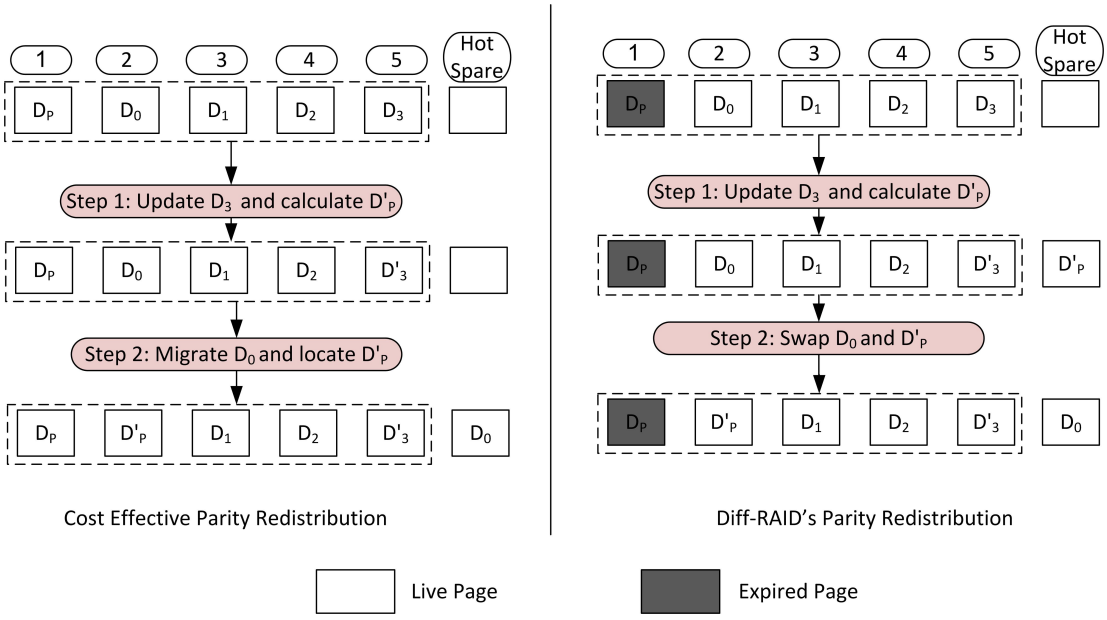



Figure 6.9: Comparison of parity redistribution for a full stripe, Case 2

but does not directly locate it onto the spare device, as in the case of Diff-RAID. It defines the target device and, if there is already a member of the logical stripe ($D_0$) present, then it is migrated to the hot spare device. After this operation is completed, the updated parity ($D'_P$) can then be located to the target device. By comparison with Diff-RAID, the new mechanism does not perform with improved performance, but the write amplification is improved in this scenario.

**Parity redistribution with cold data migration**

To redistribute the parity data in a cold data migration period, the technique first determines the type of the data (actual write or parity) by inspecting the dynamic stripe mapping table. If it is actual data, then the technique directly relocates it to the new device. Otherwise, the same procedures as hot data migration are followed. The only difference between them is that if a swap operation is required between a parity and a data block, it does not incur a performance overhead as it is triggered during an idle time period.

As a result, the technique directly migrates the parity data of the old devices to the correct target devices with the proactive hot-swapping schema, and therefore it reduces the write amplification factor and improves the I/O response time. Also, it satisfies the parity assignment ratios with minimal parity movement and therefore does not inadvertently contribute to increased ageing.

## 6.6 Semi-Hybrid RAID

Proactive hot-swapping initiates data migration from an ageing device to a new device before endurance limits become critical. Once the replacement process is complete, there are two copies of valid pages: one in the old device, and one in the new device, as a result of the coordinated data migration described in Section 6.4 as metadata has not been updated to invalidate old pages. This is a typical RAID 1 redundancy model, and may be exploited to improve read performance. In order to exploit this improved read performance further, in this section a semi-hybrid RAID architecture is presented. Here, when a device replacement is started, the technique configures old and new devices in a manner similar to RAID 1 which improves read performance and the data reliability of the system by keeping a copy of original data in a different device. In doing so, the old device is not immediately discarded and is instead retained for a period of time in order to service some (non-ageing) read requests. The architecture is referred to semi-hybrid RAID as it is effectively imposing a RAID 1 configuration over a device and its replacement in a manner that is transparent to the SSD RAID architecture of the system.

This gives rise to a more complicated hardware and metadata architecture over time as the array is used, as each device may be shadowed by a single older device that is used to service read requests; however, the benefit is that a higher throughput of read requests can be serviced, particularly in the presence of the idle time detection.

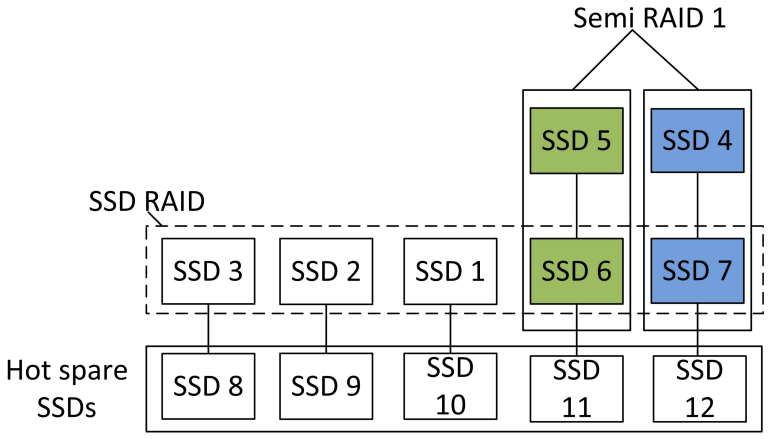



Figure 6.10: Semi-hybrid RAID after the second replacement

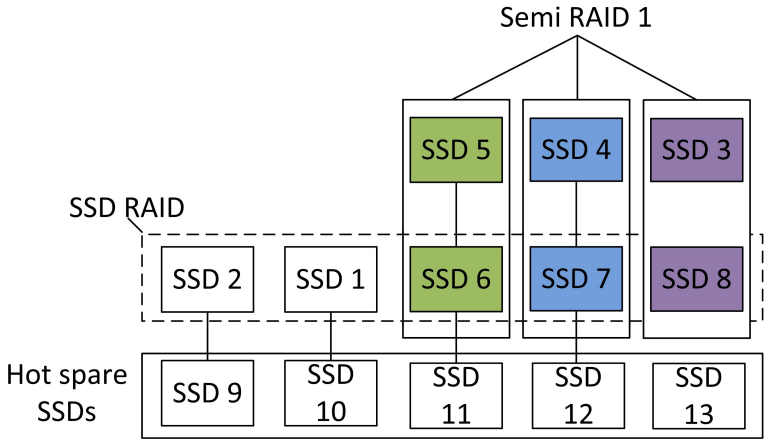



Figure 6.11: Semi-hybrid RAID after the third replacement

Figure 6.10 and Figure 6.11 illustrate an example of the semi-hybrid RAID system after the second and third device replacement processes, respectively. These diagrams have two different RAID mechanisms superimposed and co-existing. The SSD RAID system consists of devices 3, 2, 1, 6, and 7 after the second replacement. The semi-RAID 1 mechanism consists of the old devices, 5 and 4, which in this case have been replaced, shadowing the replacement devices 6 and 7. When, for instance, a replacement operation is initiated for device 3, and it is to be replaced by device 8, it is considered removed from the SSD RAID array and device 8 is imported into the array in its place, as shown in Figure 6.11. Devices 3 and 8 then form an individual semi-RAID 1

configuration. In this semi-hybrid RAID model, it is therefore possible—depending on the age of the SSD RAID array—that each individual device in the array is shadowed by its own RAID 1 partner old device.

The old device in a semi-RAID 1 array is then used to handle some of the incoming read requests. When data is updated, it is always written to the new device in the semi-RAID 1 array and the metadata is updated to reflect the location of that new data in the array; however, before this data is updated, the version stored in the old device is still valid to be read from. Therefore, when a read request comes in, the semi-hybrid RAID array technique checks to see if the old device retains a current valid copy; this incurs only a very minimal performance overhead as the metadata status table is stored in local NvSRAM. If the old device retains a valid copy of the data it may be read, otherwise the new device is read.

This concurrent architecture has several benefits. In particular, it increases the available idle time for each of the idle time detection techniques as data can be read from an old device whilst a new device is garbage collecting, or cold data migrating.

## 6.7 Summary

In this chapter, techniques were presented to enhance the real time and performance capabilities of device replacement mechanism for SSD RAID arrays. The presented novel techniques aim to improve the efficiency of the replacement process via flash aware management operations for hard real-time applications.

The proposed proactive hot swapping technique predicts and initiates swapping in advance. Thus maintaining system availability and reducing performance overhead in case of on-line reconstruction of the RAID array. This eliminates non-deterministic behaviour and improved performance for hard real-time embedded systems.

To improve system performance further during the replacement process, a new garbage collection aware data migration technique was proposed. This technique prevents possible collision between data migration of device replacement and garbage collection when both are triggered based on idle time detection methods.

Contrary to the existing techniques, a new cost effective parity redistribution technique was proposed, which offers improved system write capability and performance. Parity redistributions of existing techniques are expensive due to the additional data migration operations to achieve desired parity distribution ratios. In addition, the proposed technique reduces the parity redistribution steps, which further enhances the system write capability and performance.

Finally, a new semi hybrid RAID technique was proposed which improves the read performance of the system after the completion of the device replacement. As the proactive mechanism does not allow the old devices to be expired, therefore devices are reconfigured at RAID level to improve read performance of the system.

In conclusion, the proposed on-line device replacement framework eliminates non-deterministic device replacement behaviour in SSD RAID. In addition, due to the on-line real-time replacement, it indirectly increases the system availability. In addition, it provides greater system performance in comparison to existing mechanisms.

# Chapter 7

# Testbed and Results

In this chapter, the implementation of a testbed for a reliable and real-time SSD-based storage system is presented to investigate the effectiveness of the mechanisms presented in Chapters 4, 5 and 6. Also, the results of experiments are presented to support the use of mechanisms presented in these chapters, along with a number of synthetic and realistic (captured) traces. For accurate timing of disk requests, the NAND flash SSD simulator is designed as an extension of the Microsoft SSD simulator [37] which is derived from Disk Simulator (DiskSim) [90], a well known, validated and trace-driven disk simulator.

## 7.1 Introduction

The presented enhancement mechanisms are mapped in software architecture to build a real-time and reliable SSD storage system. These mechanisms are based on the SSD RAID mechanism which itself is based on unique parity distribution across the SSD array, unlike RAID 4 and RAID 5.

The contribution of this chapter can be summarised as follows: an architecture is presented that offers (1) deterministic response times for I/O using novel mechanisms,

regardless of workload characteristics; (2) high performance by exploiting the holistic view of the storage system; (3) a user-configurable software architecture that allows real-time operations by the host system.

The architecture is suitable for building a real-time and reliable SSD-based storage systems. It is evaluated under various synthetic and realistic workloads that include different ratios of random and sequential writes. The experimental results show a reduced WCET, improved performance and reliability compared to existing systems regardless of workload characteristics whilst maintaining the reliability mechanisms.

The chapter is structured as follows: in Section 7.2, general information regarding the DiskSim simulator is presented. Section 7.3 presents a software-based experimental platform for a real-time and reliable storage system. Section 7.4 presents simulation parameters and workload characteristics. The experimental results are presented in Section 7.5. Finally, the chapter is summarized in Section 7.6.

## 7.2 DiskSim Simulator

DiskSim is a trace-driven, accurate and efficient disk system simulator which has been widely used in number of research projects studying storage subsystem architectures [90]. It is a highly-configurable simulator and was written in C. It not only simulates disks, but also includes many secondary system components such as controllers, buses, device drivers, request schedulers, and disk cache components. It supports evaluation for both realistic and internally generated synthetic workloads. Its accuracy is widely evaluated against various hard disk drives from different manufacturers, such as Atlas III, and 10K from Quantum, Cheetah and Barracuda disk drives from Seagate, and Ultrastar 18 ES from IBM [90]. The performance behaviour of a system is modelled rather than involving actual write or read operations for I/O in the simulator.

Although it was designed to simulate HDDs, an SSD extension has been developed by a group at Microsoft Research to simulate an idealized SSD that is parametrised by NAND flash characteristics such as read, write, erase latency, number of chips, blocks and pages, reserved number of free blocks, and so on [37]. It has been used extensively in the literature to study different flash memory-related mechanisms [91–94].

## 7.3 Experimental Platform

Many embedded applications use microcontrollers for the implementation of physical systems but these are not suitable for high-integrity systems due to their performance issues caused by huge interconnectivity. Nowadays, use of FPGAs is a growing trend in high-integrity systems which can seamlessly meet the demands of high reliability, high performance and guaranteed timing behaviour [95]. FPGAs exploit design concurrency to increase throughput. In addition, on-chip integration of computing modules within a single FPGA make the scheme more reliable and performance efficient.

FPGA based designs are less expensive and do not require expensive fabrication as compared to Application Specific Integrated Circuits (ASICs). The performance of the FPGA based designs is nearly equal to ASIC [96]. The achieved performance helps real-time applications to execute in a deterministic way. A number of instructions need to be executed before a microcontroller can access data from memory. However, an FPGA already retains a memory with a direct access bus which improves its throughput significantly. There are various types of FPGAs—Flash, SRAM, and Anti-fuse. Both the flash and anti-fuse based FPGAs can only accommodate small to medium size designs. On the other hand, SRAM based FPGAs are available in large logic density and can accommodate large designs within a single FPGA but are vulnerable to Single Event Upsets [97–99].

To utilize the benefits of the FPGA based system architectures outlined above and keep consistent with the reliability enhancement mechanism [87], the presented mechanisms were adapted on an SRAM based FPGA.

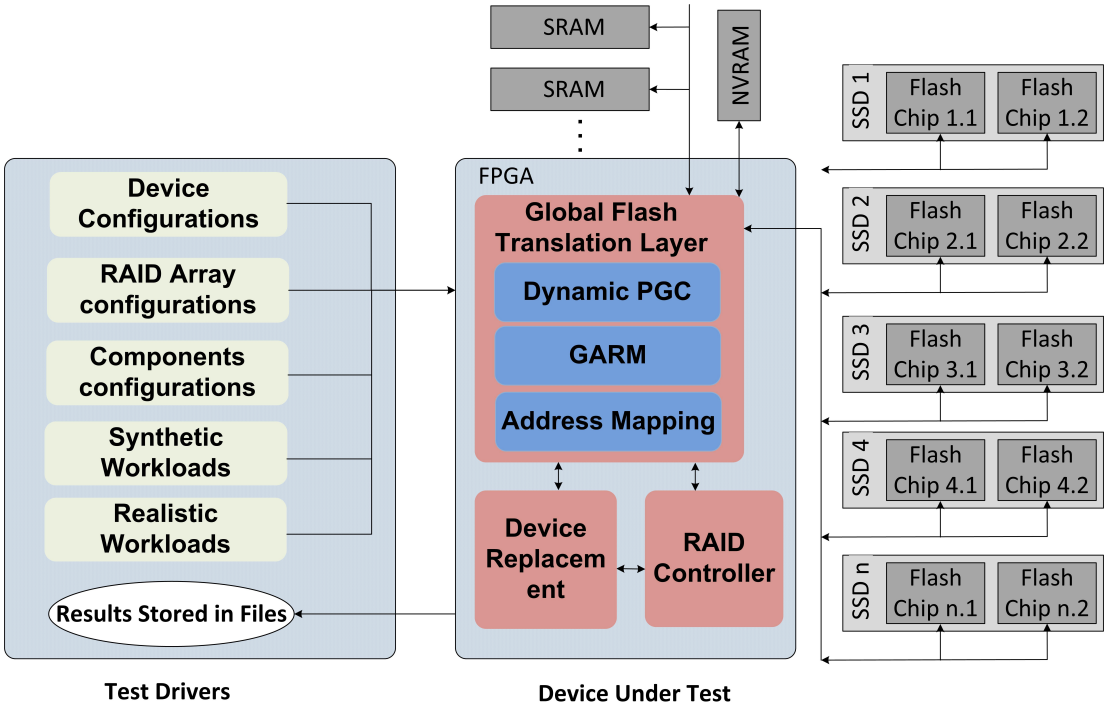



Figure 7.1: Experimental platform for a real-time and reliable SSD storage system

The experimental platform of the system and its internal communication paths are illustrated as a block diagram in Figure 7.1. It consists of the solid state devices, memory components to store metadata information (SRAM and NvSRAM), test drivers, and the FPGA-based management component. The FPGA-based management component consists of three main blocks.

The first block in the management part is referred to as a Global FTL as it takes a holistic view of the whole array, rather than a view of a specific SSD device. It is mainly responsible for the real-time garbage collection functionality and address mapping of the storage system. Two novel real-time garbage collection mechanisms were developed, including Dynamic PGC and GARM, which were presented in Chapters Chapter 4 and Chapter 5, respectively. The Dynamic PGC includes two main areas of

functionality including the dynamic garbage collection to enhance the reliability and pre-emptive mode controller to improve real-time efficiency and performance of the system. The GARM has three main areas of functionality including GC-aware data locating and serialised GC to improve WCET and performance of the system, while on-line parity migration improves the system reliability.

Global FTL also includes address mapping management. The system is based on page-level striping, where each logical address is mapped to a physical address location. SRAM is used for storing address mapping tables for each individual device in the array. Moreover, to reduce the performance overhead of metadata operations, the Global FTL stores the status (valid, invalid, or free) of all pages in the array into NvSRAM, with an individual partition for each of the devices in the array. Global FTL also manages statistical information for each device in the array, including the number of erasures performed, the number of free blocks in each device, and so on.

The second block of the device under test — labelled Device Replacement — contains the proactive hot-swapping mechanism, the coordinated data migration mechanism, and the parity redistribution mechanism as presented in Chapter 6. It monitors the Global FTL in order to make decisions regarding the activation or deactivation of these functions. According to the outputs of these functions, the RAID controller effectively relocates both the incoming requests and the partial replacement tasks across the array. Also, it may suspend or reschedule these operations according to their priority levels to avoid possible overlaps.

The third block—the RAID controller—provides the primary RAID functionality via parity distribution and the dynamic stripe mapping function. The parity distribution manages the parity ratio of each device according to initially assigned parity percentages. A basic counter is used to manage the parity distribution ratios. For example, given parity percentages of (80,19,1,0,0) in the instance of five devices in the array, first 80% of parity data of $p$ new writes (where $p$ represents the length of the period) are

located to the $n^{th}$ device, while the next 19% and 1% of the parity data of new writes are located on the $(n-1)^{th}$ and $(n-2)^{th}$ devices, respectively. Software implementation of the mechanism is depicted in Appendix A.5.

The RAID controller also manages the dynamic page allocation process. Traditional RAID levels, such as RAID 4 and RAID 5, linearly map the actual data and parity to the devices. However, unlike traditional RAID levels, the dynamic stripe mapping function keeps track of links between the logical addresses and the corresponding physical locations of the stripe units, which were intentionally created to provide an uneven parity distribution. This component also plays an important role when a page needs to be dynamically located in a GC-free device, as explained in Section 5.2.

To analyse the presented techniques under a number of different scenarios, a test driver block is presented in the experimental platform, as illustrated in Figure 7.1. It mainly consists of two types of input. First, configuration files to set up the system including the device, array and component configurations. The device and array configurations enable us to set parameters relating to the flash memory and the RAID array, respectively. The main parameters and their default values will be given in the following section. On the other hand, the component configuration provides an opportunity to enable/disable various features of the presented techniques or for the user to set initial parameters of the techniques, such as the value of the period to adjust dynamic threshold levels of the pre-emptive GC mechanism.

Moreover, the test drivers give the opportunity to test the device under a number of synthetic and realistic traces. More details regarding the parameters of the synthetic workload and characteristics of the realistic traces will be given in the following section. Simulation results are gathered into a file to be analysed for each experiment.

Table 7.1: Default flash array parameters

| **Parameter** | Value |
|---|---|
| Reserved free blocks | 15% |
| Minimum free blocks | 5% |
| Flash chip elements | 1-2 |
| Blocks per plane | 2048 |
| Pages per block | 64 |
| Page size | 4 kB |
| Page read latency | 0.025 ms |
| Page write latency | 0.200 ms |
| Block erase latency | 1.5 ms |
| Redundancy Schema | Taken From [11] |
| Stripe unit size | 4 kB |

Table 7.2: Default parameters of synthetic traces

| **Parameter** | Value |
|---|---|
| Request Size () | 4 kB |
| Inter-arrival Time | 3 ms |
| Probability of read access | 0.2 |
| Probability of sequential access | 0.2 |

Table 7.3: Characteristics of realistic workloads

| **Workload** | **Read %** | **Avr. Inter-arrival time (ms)** | **Request Size (kB)** |
|---|---|---|---|
| Postmark | 30 | 0.92 | ~4 |
| Boot | 92 | 3.32 | ~4 |
| Financial | 8 | 3.77 | ~4 |

## 7.4 Simulator and Workloads

The default parameters of the SSD array are given in table Table 7.1. The default parameter file for an SSD is presented in Appendix A.1. The experimental SSD array consists of five initial devices. Also additional spare devices are initiated when it is required. The default storage system topology is given in Appendix A.3 Since the

reliability mechanism is based on page-level striping, the stripe unit size is set with a page size of 4 kB. Other default parameters of the array are given in Appendix A.2.

Experiments were conducted using a mixture of realistic and synthetic workloads in order to analyse the WCET, performance and reliability of the proposed mechanisms. Usage characteristic parameters required for the synthetic traces were set with default values as shown in table Table 7.2. For synthetic workloads, some parameters were varied including the probability of sequential access, inter-arrival time of requests, and the sizes of the requests. The synthetic trace runs until the most aged devices reaches its predefined endurance limit for the ageing experiments. For the performance experiments the mechanisms are tested with average of 100,000 synthetic I/Os.

For realistic traces, two types of captured traces for embedded systems were used to analyse the WCET and performance of the mechanisms. They are captured by the Flashmoon tool [100] that enables the monitoring of embedded NAND flash memory I/O requests. Also, a well-known, random write-dominant server trace (financial [84]) was chosen to conduct ageing tests.

Characteristic features of the realistic traces are given in Table 7.3. Overall they contain a small request size, which also makes the reliability mechanism work close to its optimum reliability level. In particular, Postmark is a benchmark which aims to generate the behaviour of a file system. It is a write-dominant workload. On the other hand, boot benchmark is captured I/O operations to NAND flash memory during the kernel boot process. As one might expect, it is read-dominant and instructions are relatively light in terms of inter-arrival rate. Financial trace is a captured server trace and includes a high enough number of write requests (nearly 4 million) [84].

For performance evaluation, the entire flash-based storage was first filled with valid data before conducting performance experiments in order to perform a fair performance and WCET analyses of the mechanisms.

## 7.5 Simulation results

In these experiments, the data reliability and performance of an SSD-based storage system is evaluated after adapting the presented mechanisms including the Dynamic PGC, GARM and device replacement techniques presented in Chapters 4, 5, and 6, respectively. Each time, one of these was evaluated by disabling the others to independently measure real improvements for each of the mechanisms. The comparison is drawn between the proposed mechanism and state-of-art studies such as PGC [17], the reliability mechanisms [8, 11], or a standard flash RAID without enabling any techniques with the parameters presented in Section 7.4.

This section can be categorized as follows. Reliability and performance evaluation of the dynamic pre-emptive GC is presented in Section 7.5.1. Section 7.5.2 presents the performance, WCET, and reliability evaluations of the GARM. Finally, improvements of the device replacement techniques are presented in Section 7.5.3.

### 7.5.1 Dynamic Pre-emptive Garbage Collection Mechanism

The reliability and performance analysis of the dynamic pre-emptive GC is evaluated. The parameters by which the SSD array is defined are illustrated in Table 7.1. $P_H$ factor ($h$) is set at 2% and $P_M^T$ is set at 1% of the total number of blocks. The default values of the synthetic traces used in the experiments are given in Table 7.2. To create a variety of workload scenarios with synthetic traces, an exponential and normal distribution is used for varying request sizes and inter-arrival times of requests. The main part of the software implementation of the presented techniques can be found in appendices Appendix A.6, Appendix A.7 and Appendix A.8.

**Reliability Analysis**

Two techniques are evaluated and compared: PGC-RAID, the pre-emptive GC [17] with the reliability mechanism of [11] which is considered as a base case; and Dynamic

PGC, the dynamic pre-emptive GC mechanism for SSD-RAID systems, as described in Section 4.5. Also, the optimal ratios for the age distribution percentages of the SSDs in an array for an optimum reliability level which is calculated based on the ageing formula in reference [8] with initially assigned parity ratios of 80%,19%,1%,0%,0% for 5 SSDs. It is only drawn for comparison purposes.

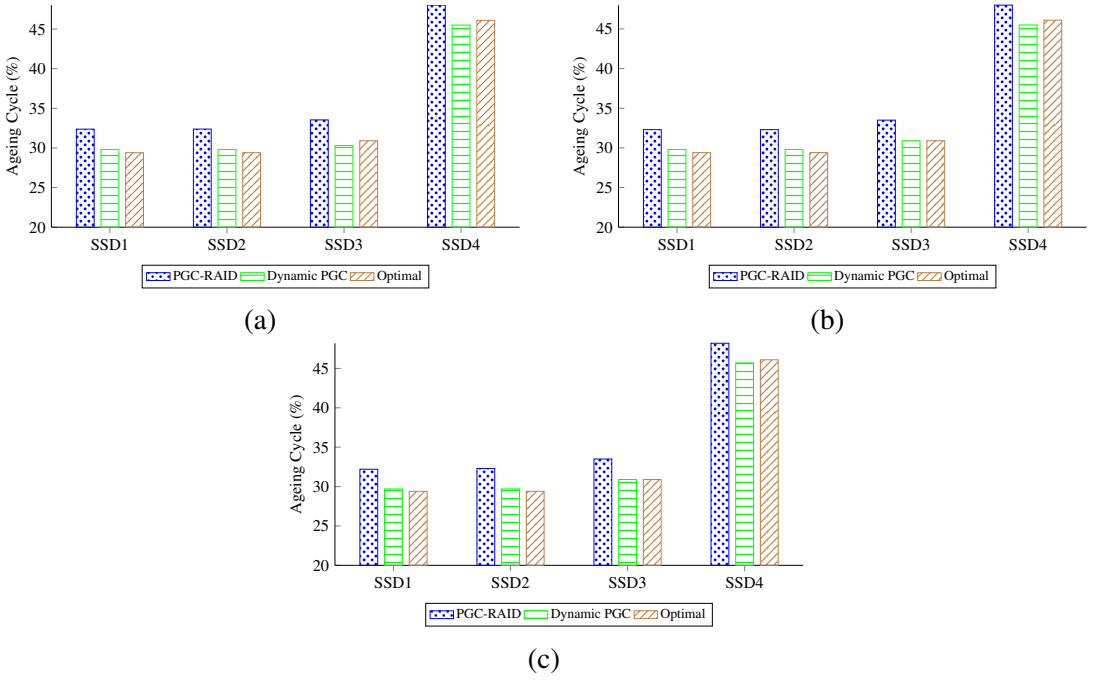



Figure 7.2: SSD age distributions by varying probability of read access. Probability of read accesses: a (0.2), b (0.4), c (0.6)

Figure 7.2a, b and c illustrate the ageing characteristics of each mechanism with varying read access probabilities; these cause write access probabilities to vary also as the probability of an access is the sum of both. It can be clearly seen that Dynamic PGC shows improvement in ageing distributions of devices compared to PGC-RAID for all cases. For example, the age distribution of SSD 3 is 7% closer to its optimum ageing level. As the probability of read access increases, Dynamic PGC performs slightly better in terms of tending towards optimum levels.

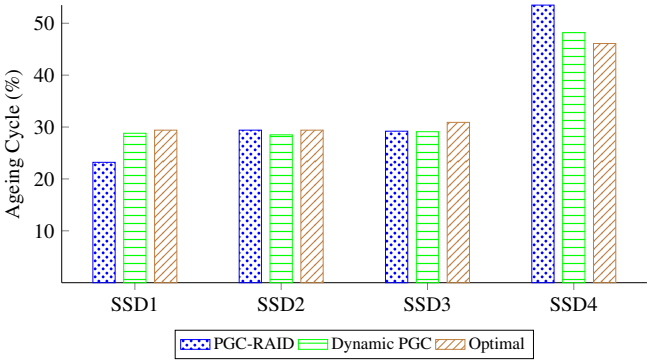



Figure 7.3: SSD age distributions with the financial trace

Furthermore, the techniques were evaluated and compared with a random write-dominant realistic trace. Dynamic PGC essentially eliminates the deviation in the age distribution levels of the reliability mechanism when coordinating $P_L^T$ and $P_H^T$ as illustrated in Figure 7.3. The ageing ratio of SSD 4 is much closer to the optimum level compared to PGC-RAID (by nearly 10%). Also, the ageing ratio of SSD 1 is 19% closer to its optimum level. This is because the Dynamic PGC has raised thresholds for SSD 1 while lowering them for SSD 5 when the deviation in age distribution was detected. SSD 5 does not appear on Figure 7.3 because it is the most aged and reference device to calculate the ageing cycles of other devices and its ageing cycles are equal to 100% for all experiments.

Garbage collection directly affects device lifetime due to erase operations. To observe its effect over the lifetime of the storage, it is compared with existing techniques using constant thresholds for pre-emption. Threshold 2 ($Th_2$) is set with a soft level of 5% and a hard level of 3%, and threshold 3 ($Th_3$) is set with a soft level of 8% and a hard level of 5%. Only optimal GC triggering threshold levels for PGC are considered for the experiments [17]. A higher threshold level than $Th_3$ will increase the number of erasures which negatively affects the lifetime of the flash memory. Also a lower threshold than $Th_1$ will increase probability of entering normal GC state where response time becomes non-deterministic. Figure 7.4 illustrates that the Dynamic PGC

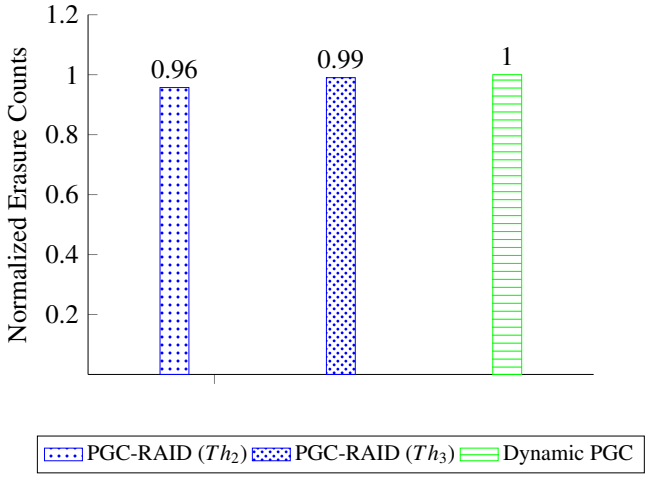



Figure 7.4: Normalized total number of erasures performed in the array with the financial trace

has a slightly higher erase count than PGC-RAID. It has 4% more erasure operations than PGC-RAID with threshold 2, and no tangible difference with threshold 3. While the threshold levels increase for PGC-RAID, the erasure count continues to rise.

Also, it was found that reducing the period of the Dynamic PGC for calculating thresholds helps ensure that the age distribution tends towards optimum levels, improves the age distributions further towards the optimum level but has a detrimental effect on device lifetime.

Overall, the results demonstrate that the Dynamic PGC improves the age distributions of devices (which is necessary for the reliability mechanism to be effective) along with offering upper bounds for I/O response, with minimal impact on device lifespan with the help of dynamic garbage collection component of the proposed mechanism.

**Downtime and Performance Analyses**

The real-time efficiency of the mechanisms was evaluated using a number of synthetic traces by varying the probability of read access as 0.2, 0.3, and 0.4. Overall, average inter-arrival time was measured as 1.2 milliseconds (ms) for the experiment. Also, the threshold for the $GCE$ function is defined as 15% of total pages in a block.

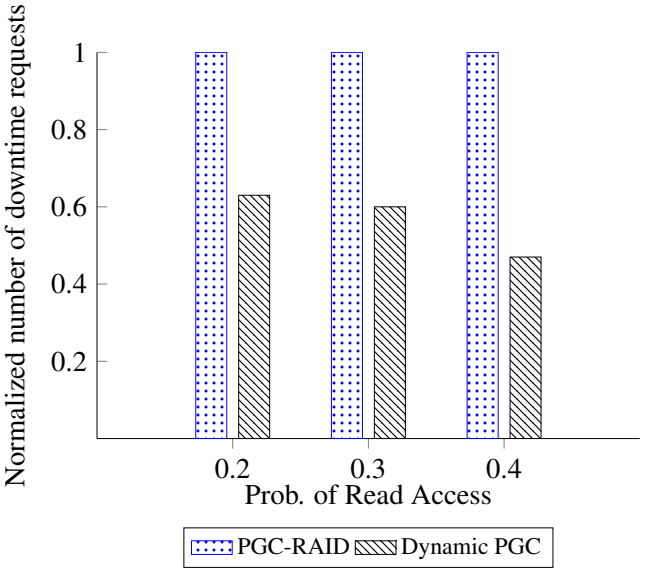



Figure 7.5: Comparison of number of requests arrived during downtime period

Figure 7.5 shows the amount of I/O requests (arriving during downtime) of each mechanism with varying probability of read ratio of I/O requests. For a short probability of read (0.2), the Dynamic PGC shows improvement in a number of downtime requests by 40% over PGC-RAID. As the probability of read access increases, the Dynamic PGC exhibits further improvement.

Moreover, the performance characteristics of each mechanism were analysed for the same traces of Figure 7.5. The Dynamic PGC shows improved average response times compared to PGC-RAID in all cases. However, the mechanism only shows a limited performance improvement of up to 3% compared to PGC-RAID as illustrated in Figure 7.6 due to the characteristics of the workload.

Overall, simulation results demonstrate that the Dynamic PGC not only provides real-time access guarantees, but also maintains the reliability mechanism. It also reduces the downtime of the array during erase operations, thereby causing fewer write operations to be postponed compared to PGC mechanism.

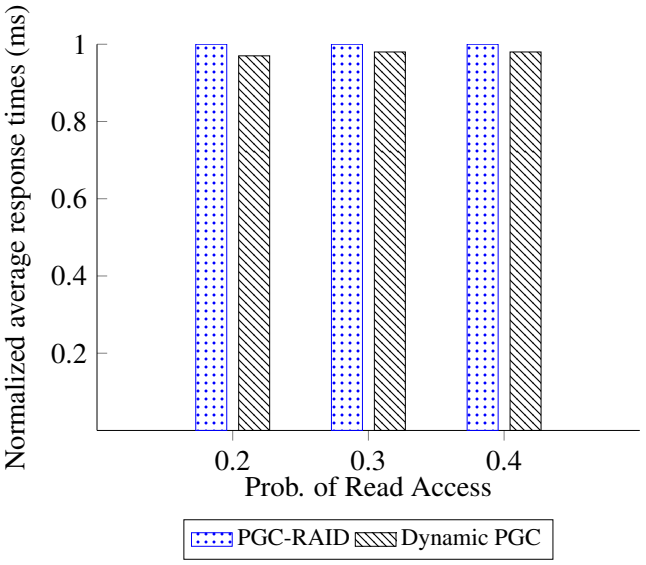



Figure 7.6: Comparison of average response times

Although the Dynamic PGC offers real-time access guarantees with minimal changes to the age distribution mechanism, the mechanism still suffers from a long maximum response time. Also, its performance enhancement is limited compared to the existing PGC mechanism (which showed only up to 6% improvement). To address these limitations, GARM was presented, and is analysed considering a number of different workloads in the following section.

### 7.5.2 Garbage Collection-aware RAID Mechanism

In this section the performance, WCET and reliability of the GARM were evaluated with number of synthetic and realistic traces.

A number of different GC algorithms were examined including a non-GARM, which was described in Section 5.2 and is considered to be the baseline implementation; GARM, which is presented in Section 5.2; and naive pre-emptive GC [17]. All these techniques are adapted with the reliability mechanism of reference [11]. The non-GARM and PGC mechanism uses the normal reactive GC scheme for cleaning,

where GC is only initiated according to the internal status of the flash chip in the array, and is not coordinated. However, the GARM mechanism considers a holistic view of the storage when initiating/postponing a cleaning operation.

A mixture of realistic and synthetic workloads were used to examine the presented technique. For synthetic workloads, some parameters are varied, including the probability of sequential access, inter-arrival time of requests, and the size of the request. The entire memory was filled with valid data before conducting performance experiments in order to perform fair performance analyses of the mechanisms.

**Performance Analysis**

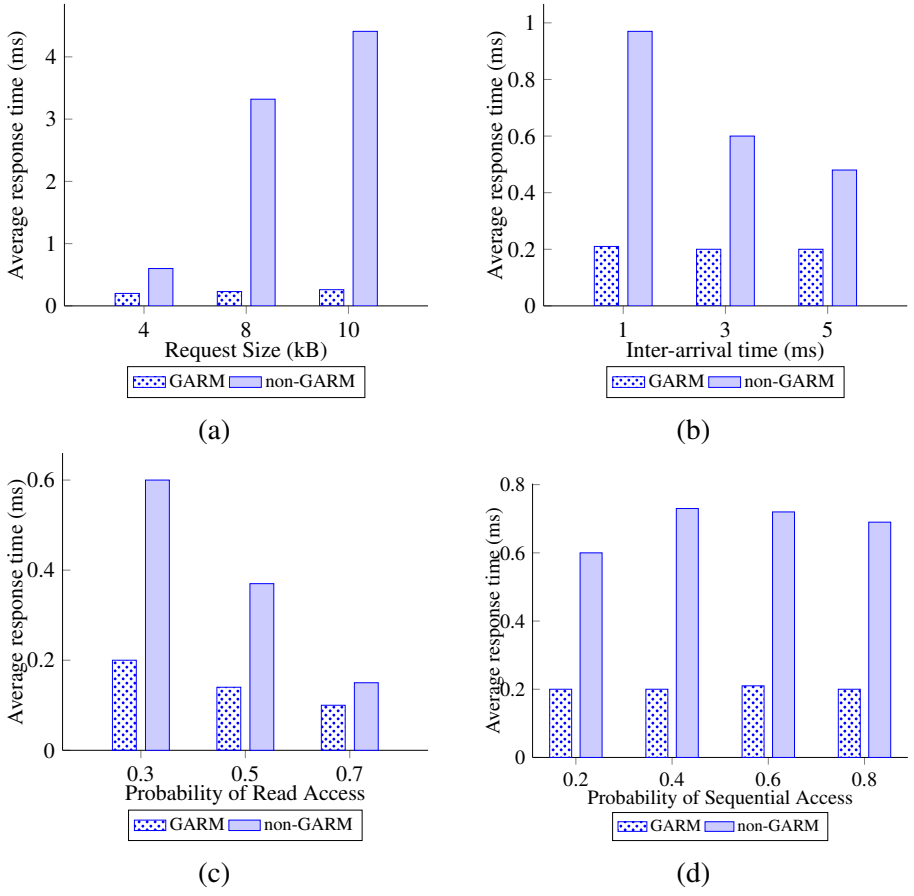



Figure 7.7: Performance improvements of GARM for synthetic workloads. Average response times are depicted with different parameters of synthetic workloads. (a) Request size. (b) Inter-arrival time. (c) Read ratio (d) Sequentiality.

**Synthetic workloads:**

The non-GARM and the GARM schemes were examined under a number of synthetic workloads.

First, evaluation was started on the performance of the GARM under various synthetic workloads. Incoming requests are not aware of ongoing cleaning operations in the baseline GC schema and thus they have to wait until GC finishes although the mechanism dynamically relocates them across the available elements. Figure 7.7 illustrates the performance improvements of the GARM schema under synthetic workloads.

**Request size:**

In this experiment, the request size was varied as 4, 8, and 10 kB. These values were selected because the reliability mechanism achieves maximum efficiency with partial stripe writes, which enables the creation of the desired age imbalance among the devices. Figure 7.7a shows the improvements in performance of the GARM schema for different request sizes. For a small request size (4 kB), response time is improved by a factor of three. As the request size increases, further improvements are observed. For a large request size (10 kB), the response time significantly decreases. This is because the mechanism is not affected by an increase in the number of cleaning processes.

**I/O arrival rate:**

A similar enhancement was observed with respect to varying the arrival rate of I/O requests. The inter-arrival time was varied between 1 and 5 ms in the experiments conducted. In Figure 7.7b demonstrates that GARM is minimally affected by changes in high arrival rates. However, the system response times for the baseline substantially increase with respect to the request arrival rate.

**Read Ratio:**

Writing a page is expensive compared to a read operation in flash memory. Also, further delay is incurred by the cleaning process, which is proportionally related to the

number of write operations requested. Thus the mechanisms were evaluated with a number of traces by varying the read/write ratios in the traces. In Figure 7.7c, a significant improvement in average response time was observed with a number of traces by varying the read ratio. The mechanism provides a 50% better response time for a read-dominant trace (read ratio 0.7) compared to the baseline mechanism. It further improved the performance with a write-dominant workload by a factor of nearly three.

**Sequentiality:**

GC overhead increases due to random workloads as they produce more invalid pages due to additional parity update operations. The mechanism is evaluated by varying the sequentiality of requests. Figure 7.7d illustrates the enhancement in average response time. For example, for a 40% sequential workload the mechanism improves the system response time by a factor of more than three compared with the baseline mechanism. Also, it can be seen that performance levels of the mechanism remain constant for all levels of sequentiality.

It can be observed from the performance evaluation with synthetic workloads that the GARM improves the performance, regardless of workload characteristics. Also, it was noted that the mechanism shows better performance improvements with workloads that cause more cleaning operations (e.g., fewer small request sizes, fewer large inter-arrival times, more random access, and fewer read access).

**Realistic workloads:**

Performance evaluations of the non-GARM, PGC and GARM mechanisms are examined under captured embedded traces—postmark and boot. Characteristic features of these traces are given in Table 7.3.

Figure 7.8 shows the improvement of system performance for realistic workloads. For a write-dominant workload, (postmark), the average response time is improved by a factor of nearly 13 compared to the baseline system, as shown in Figure 7.8a. Also, an improvement was observed in average response time by 18% compared to the PGC

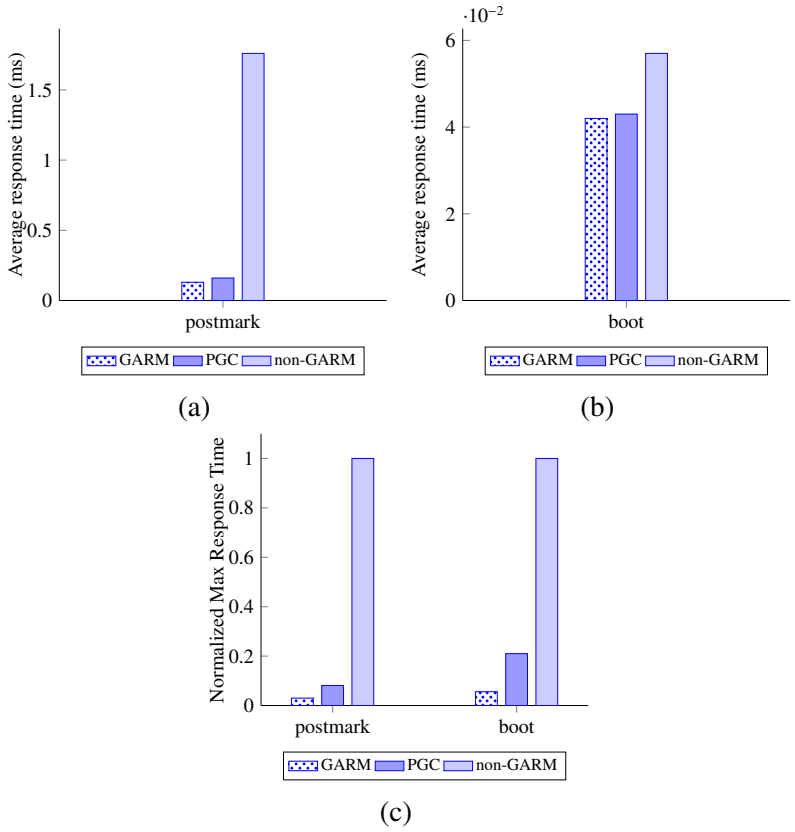



Figure 7.8: Performance improvements of GARM for realistic embedded system workloads. (a-b) Average response time. (c) Maximum response time.

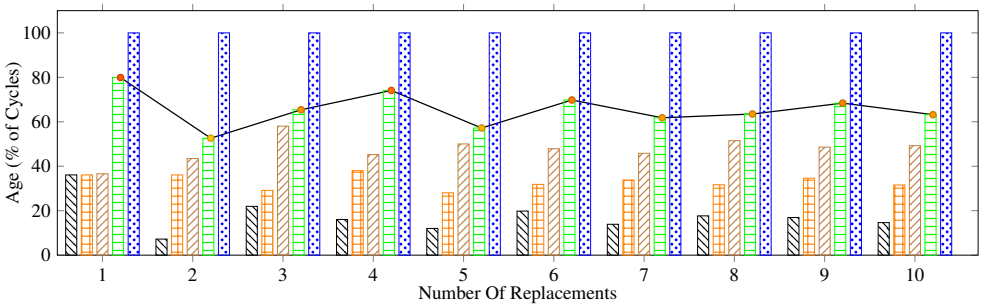
mechanism. For a read-dominant workload (boot), the mechanism enhances in system performance by 26% and 2% compared to the baseline and the PGC mechanisms, respectively as shown in Figure 7.8b.

Moreover, maximum response times of the architectures were evaluated. The GARM shows a 61% and 73% better maximum response time compared to the PGC mechanism for postmark and boot workloads, respectively, as illustrated in Figure 7.8c. It can be seen that response times are significantly improved compared to the baseline mechanism for both workloads because the mechanism eliminates the long delay arising from the cleaning process by dynamically disturbing data stripe units.

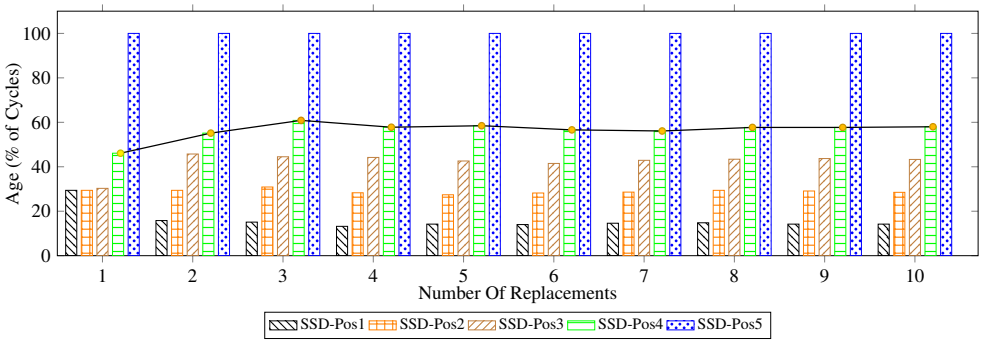
**Reliability Analyses**

Possible negative effects of the GARM mechanism on the parity distribution ratios were discussed in Section 5.2.7. To overcome these limitations, an on-line parity migration was presented.

In this section, age distribution ratios of the GARM mechanism are analysed by enabling/disabling the OPM feature presented in Section 5.2.7. The reliability mechanism has strict control over the lifespan of each element in the array. The proposed reliability mechanism applies an (80%, 19%, 1%, 0%, 0%) parity configuration for first two device replacements and then applies a (80%, 5%, 5%, 5%, 5%) parity configuration onwards in the case of a five element SSD array.




(a) SSD age distributions of GARM disabling the OPM




(b) SSD age distributions of GARM enabling the OPM

Figure 7.9: The changes in age distributions of SSDs with a GARM by enabling and disabling the OPM

A number of ageing tests were conducted with a write-dominant synthetic trace by using the Microsoft SSD simulator [37]. The default values of the parameters used for the synthetic trace are given in Table 7.2.

The results show that when the most aged device reaches its normalized endurance limit, it contains 26% less parity data than its desired level, as shown in the case when on-line parity migration is disabled in Figure 7.9a. Most of these parities are migrated to the second most aged device in the array, which is 42.5 % more aged than its optimum ageing level due to the dynamic data allocation of the GARM.

To observe the effects of degenerate parity distribution throughout other device replacement iterations, the expected ageing ratio for each of the devices were calculated for 10 iterations considering ageing ratios at the first device replacement point, as depicted in Figure 7.9a. The figures demonstrate that second most aged device has exceeded its critical ageing ratio by 60% several times at replacement points 1, 3, 4, 6 and 9. Also, during the first device replacement point, the age of the second device was observed to be 80%, such that the probably of data loss is significantly higher [8]. Moreover, the ages of second most aged devices fluctuate with GARM while it is stable after the third device replacement with respect to the default reliability mechanism which is illustrated with solid lines on figures.

The same experiment was conducted enabling the feature of OPM. It is clear that the OPM almost completely eliminates the negative effects of GARM by redistributing parity data, as depicted in Figure 7.9b. It recovers the average parity distribution ratios back to their optimum levels with a minimal performance overhead. The results show that the OPM keeps the default parity distribution percentages of the SSDs in the array and thus maintains the reliability enhancement of the mechanism.

Overall, the results show that the presented techniques outperform existing mechanisms in terms of average response time and maximum response time for various types of workloads. The architecture not only provides deterministic response times but also

maintains the reliability mechanism against simultaneous device failure. The on-line parity migration maintains the initially assigned parity ratios among the devices in the array and thus the reliability enhancement of the mechanism is not disturbed.

The GARM mechanism successfully addresses the root cause of non-deterministic behaviour of the flash memory, which is garbage collection. However, some problems regarding the flash RAID array still remain (e.g., the requirement for off-line device replacement), which makes this unsuitable for real-time applications where deterministic response time and continuous system availability are required. To address this, a number of presented on-line device replacement mechanisms are investigated in the following section.

### 7.5.3 On-line Device Replacement Techniques

Experiments were conducted only with synthetic traces (as published captured traces are not sufficient to age devices to the desired levels) to analyse performance, device replacement time, and write amplification. Usage characteristic parameters required for the traces were set with a request size of 4 kB, an inter-arrival time of 3 ms, and a probability of read access of 0.2 as default. The results of the performance evaluation, device replacement time, and write amplification experiments are presented as follows.

**Performance Analyses**

Two different performance evaluations are given. Firstly, system response times are measured during the replacement process to evaluate the efficiency of the proactive hot-swapping and coordinated data migration techniques. Secondly, the read performance of the system is evaluated with the semi-hybrid RAID once a device replacement is completed. Inter-arrival times of requests are varied over a normal distribution for three experiments, with average times of 2.0, 3.0, and 4.2 milliseconds (ms). A basic

idle time detection for garbage collection and device replacement tasks is employed, as presented in Section 6.4.

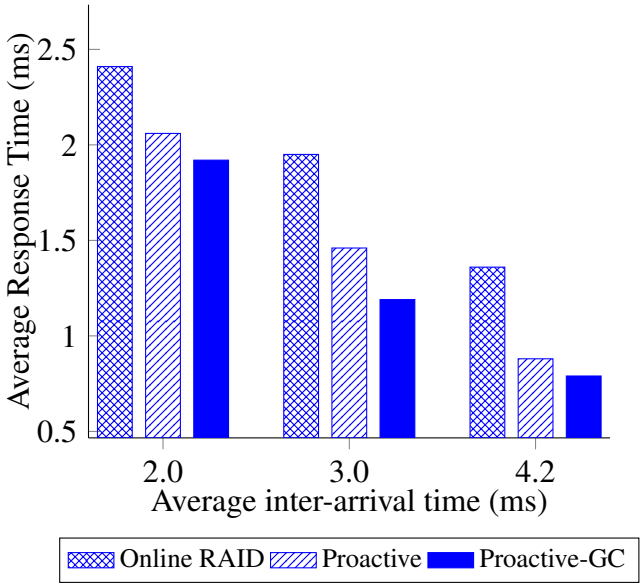



Figure 7.10: Average response time of traces by varying inter-arrival time

Figure 7.10 illustrates the performance characteristics of three techniques: On-line RAID— the on-line reconstruction of Section 6.4, Proactive—the proactive hot-swapping without garbage collection of Section 6.3, and Proactive-GC—the proactive hot-swapping with garbage collection of Section 6.4. For a short inter-arrival rate (2 ms), proactive hot-swapping exhibits an improvement in response time by 16% over basic on-line reconstruction. As the inter-arrival time increases, proactive hot-swapping exhibits further improvements in average response time. Moreover, further improvement in response time is exhibited by proactive hot-swapping with garbage collection. For an inter-arrival time of 3 ms, the average response time during the replacement process further improves by 19% over proactive hot-swapping without garbage collection. For longer inter-arrival times, improvement in average response time decreases because the idle time periods of the trace are long enough to reduce the performance overhead of overlaps between the garbage collection and partial device replacement tasks. For shorter inter-arrival times (2 ms), the proactive techniques do

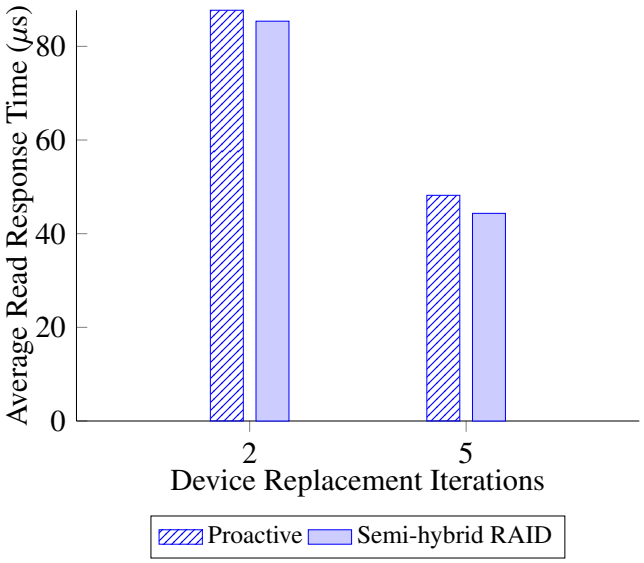



Figure 7.11: Average read response time of RAID schemas after different device replacement points

not exhibit significant differences due to the limited number of idle time periods in the workload.

Figure 7.11 shows a comparison of read performances for Proactive—proactive hot-swapping using the parity distribution of reference [11] and the Semi-hybrid RAID of Section 6.6 after two and five device replacements. The size of read operations was fixed at 8 kB and the probability of read access configured as 0.4. Results exhibit only marginally better performance in the semi-hybrid. However, the read performance of the semi-hybrid RAID can be further improved with a lower parity percentage on the most aged device. Typically the most aged device holds 80% parity and 20% data, and so the amount of data migration is low when device replacement is triggered. As the percentage of data migrated to the new devices increases, the possibility of incoming read requests hitting back-up devices in the semi-hybrid architecture rises, and so lesser parity percentages will improve the hit rate and consequently performance.

**Device Replacement Time Analyses**

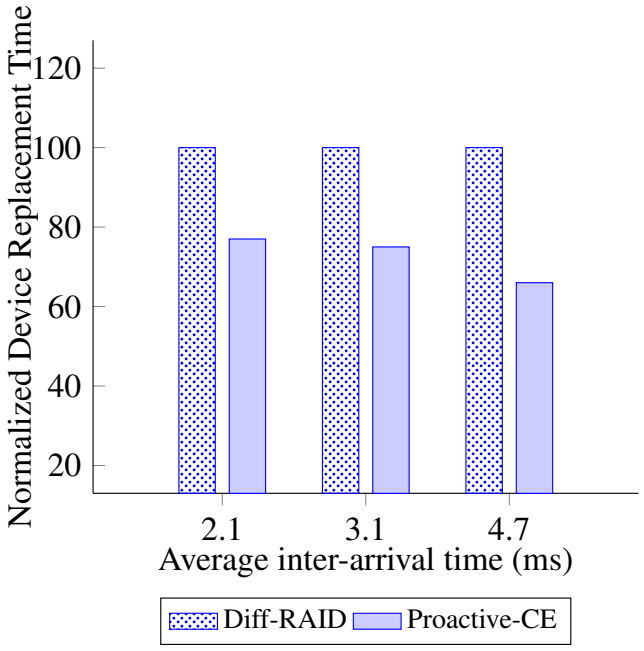



Figure 7.12: Device replacement times for parity redistribution

Figure 7.12 presents the results of comparing device replacement times for Proactive-CE—the on-line cost-effective parity distribution of Section 6.5 to that of Diff-RAID [8]. Default configuration parameters are used, and inter-arrival times varied. Results are normalized to enable direct comparisons. The cost-effective parity distribution technique exhibits a speed-up of 34% over Diff-RAID due to proactive hot-swapping. As the inter-arrival time increases so do the performance benefits due to co-ordination of idle time periods.

**Write Amplification Analyses**

Two different garbage collection triggering approaches are used to measure write amplification. The first is idle time-based, where cleaning takes place in the background. The second is threshold-based, where cleaning is triggered based on the amount of free space remaining. Both employ a greedy policy which selects the dirtiest blocks to clean.

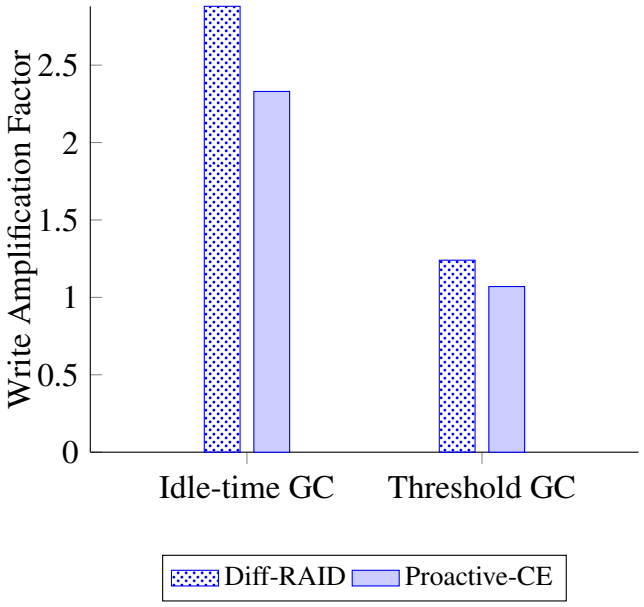



Figure 7.13: Write amplification (random write workloads)

Figure 7.13 presents the write amplification factors for both Diff-RAID and Proactive-CE—the cost-effective parity distribution with proactive hot-swapping of Section 6.5 with both garbage collection approaches.

Under both garbage collection approaches, the cost-effective parity distribution technique exhibits improved write amplification, with the idle time approach exhibiting the largest percentage improvement; Diff-RAID exhibits an amplification factor of nearly 3, whilst cost-effective parity distribution shows a factor of only 2.

Overall, simulation results demonstrate that non-deterministic behaviours of the device replacement in reliability enhancement mechanisms successfully eliminated via proactive device replacement technique that provides continues system availability while replacing a device. Also GC-aware data migration enhances I/O response time during device swapping. The results also indicate the on-line parity redistribution technique improves write capability and replacement time compared to other existing works. Moreover, the semi-hybrid RAID improves read performance after device replacement is complete.

## 7.6 Summary

In this chapter, implementation of the proposed real-time and reliable flash-based storage architecture was presented. This architecture incorporates all the proposed mechanisms and techniques presented throughout this thesis. First, the Dynamic PGC mechanism for SSD RAID was analysed and tested. The results showed that the proposed mechanism provides deterministic response time while ensuring non-disturbed parity ageing levels of the devices. Second, the proposed GC-aware RAID mechanism was tested. The results showed that it improves the WCET for the PGC with the help of proposed features of serialised GC and GC-aware read/write. In addition, the Global FTL component of the architecture further improves the performance of Dynamic PGC and GARM by benefiting holistic view of the whole storage array. Thirdly, on-line device replacement framework was analysed and tested. This on-line replacement mechanism offers SSD specific device replacement with deterministic response time which improve the system performance and write efficiency.

In comparison to the existing, proposed mechanisms/techniques are also quantitatively analysed. Dynamic Garbage Collection of the Dynamic PGC improves the system reliability in terms of the optimum aging ratios by up to 19% in comparison to the existing pre-emptive GC mechanism. In addition, the pre-emptive mode controller of the Dynamic PGC reduces down time of storage up to nearly 50%. The garbage collection aware RAID shows up to 73% better WCET as compared to the existing real-time GC mechanisms for a number of synthetic and realistic workloads. It also improves the system performance up to 50% and 26% compared to the baseline mechanism, where GC is not coordinated, for synthetic and realistic traces, respectively. Finally, the results on the device replacement demonstrate that the average I/O response time is improved by up to 39% while simultaneously offering system availability. In addition, new semi-hybrid RAID feature improves read performance by an average of 8%.

Contrary to the existing mechanisms, the proposed mechanisms and techniques showed promising results. This is the first ever SSD storage system which simultaneously meet the demands of real-time and reliability. In general, the architecture is suitable to all embedded applications but most suitable to meet the demands of real-time and reliable SSD storage systems.

# Chapter 8

# Conclusions and Discussions

In this thesis, the real-time support and performance concerns in SSD-based storage systems were investigated. The motivation behind this thesis was to investigate novel enhancements in NAND flash-based SSDs. The system reliability and the performance of flash-based devices has become a major challenge for hard real-time applications due to the non-deterministic behaviour of the management components. Unpredictable performance in the system can be partially solved by various proposed real-time GC schemas, but they do not present reasonable explanations for the research question which were proposed in Chapter 1. The real-time support and performance issues were addressed in this thesis with a number of novel mechanisms including real-time garbage collectors and on-line device replacement techniques.

## 8.1 Final Evaluation of Results

A number of research questions were shaped as result of the research challenges presented in Chapter 1. These questions considered as benchmarks by which to solve the stated issues of storage systems and thus these questions need to be answered to show that the outcomes of the thesis are assessed. The answers to the each of the orig-

inal questions is confirmed on the contributions made by this thesis, as presented in previous chapters. Here, the answers to the research questions are summarised:

**I. Can existing real-time garbage collection mechanisms be adapted to work with RAID techniques where the devices in the array are strictly controlled by the reliability mechanisms?**

The answer to this question is yes, but after a few modifications. The solution to this question is presented in Chapter 4 with a number of enhancements over the naive pre-emptive GC mechanism. Chapter 3 explained real-time support issues of flash memory and the limitations of the existing real-time techniques regarding reliability mechanisms. To experimentally observe these limitations, a number of experiments were conducted and are presented in Chapter 4. To address these limitations, Dynamic PGC was presented which minimise deviations of the age distribution ratios and show higher system performance in the reliability mechanisms.

**II. How can a real-time GC mechanism be developed without (or by minimally) disturbing the reliability mechanisms in the RAID array?**

This can be possible with a dynamic GC component of the Dynamic PGC mechanism which not only considers the remaining free space in the memory but also the current ageing level of devices in the array when triggering a cleaning process. It does not fully eliminate the deviation in the reliability mechanism, but it is significantly reduced.

**III. How can the downtime of flash-based RAID storage be improved further with the real-time GC mechanism?**

In Chapter 4, it is emphasised that the existing real-time mechanism has several deployment issues over RAID architecture. To address these limitations, the pre-

emptive mode controller component of the Dynamic PGC was introduced to improve the real-time efficiency of the PGC mechanism. It reduces the down time of the array during erase operations, thereby causing fewer write operations to be postponed compared to other published techniques and thus the performance of the system, is further improved.

**IV. Can the WCET of the real-time GC mechanisms be reduced further by utilizing the concurrency of parity-based RAID architecture?**

The answer is yes. The existing real-time mechanisms do not present an architecture that completely avoids I/O requests from being blocked by GC operation. They usually adopt a partial or pre-empted GC mechanism where the longest atomic operation involved in WCET calculation is as highlighted in Chapter 3. The mechanism, which is called GARM, provides reduced WCET compared to the existing mechanisms by benefiting from the concurrency of the architecture.

**V. How can unpredictable GC delays in flash-based RAID architectures be completely avoided without disturbing the reliability mechanisms?**

To completely avoid the long latency of the GC with RAID, a number of features are introduced into the GARM including GC-aware write and read and a serialised GC. The GC-aware write feature of the mechanism dynamically allocates incoming requests across the GC-free devices, and for read operations redundant data is utilized with the help of the serialised GC.

**VI. Can the existing HDD-based on-line device replacement techniques be fully adapted to flash-based mechanisms?**

The answer to this question is no. The main reason for this answer is due to the characteristics of device failure in an HDD are different to those of an SSD-based device. The failure due to wearing out of flash cells in an SSD is deterministic,

while an HDD failure will be non-deterministic. Therefore, SSD-specific on-line device replacement techniques are required.

**VII. How can continuous system availability be provided when a device replacement is required for the reliability mechanism?**

To provide continuous system availability for the reliability mechanisms, a proactive hot-swapping technique is proposed in Chapter 6. It does not allow the most aged device to be failed, but instead triggers on-line data migration when the device is nearing its endurance limit, thus maintaining system availability.

**VIII. How can the performance overhead of the device replacement techniques be reduced in SSD RAID?**

HDD-based device replacement techniques do not provide an efficient replacement technique for SSD-based RAID systems because they are not aware of the inherited features of flash memory, such as garbage collection. To improve the efficiency of the replacement process, two techniques were proposed. First, a coordinated data migration that avoids possible overlap between GC and data migration. Second, a semi-hybrid RAID configuration was presented to improve the system performance further once device replacement is completed.

**IX. How can the proposed mechanisms & systems be tested and validated?**

An FPGA-based system architecture was adapted in the software simulator. To build this environment, a number of design components were implemented including dynamic page allocation, uneven parity distribution, Global FTL, and so on. The experiments were conducted using both synthetic and realistic traces. To compare the results, a baseline scenario was also simulated.

Table 8.1: Comparison of presented techniques with existing real-time and reliability enhancement mechanisms

| Features | GFTL | RFTL | PGC | Diff RAID | Flash RAID [11] | Dynamic PGC [1] | GARM [1] |
|---|---|---|---|---|---|---|---|
| **WCET** | Low [2] | Low | Low | None | None | Low | Very Low |
| **Additional Memory for GC** [3] | Yes | Yes | No | No | No | No | No |
| **Online Device Replacement** | None | None | None | Not Supported | Not Supported | Yes | Yes |
| **Random I/O Performance** | Normal | High | High | High | High | High | Very High |
| **Sequential I/O Performance** | Normal | High | High | High | Low | High | High |
| **Protection** | No | No | No | Single Device Failure | Single Device Failure | Single Device Failure | Single Device Failure |
| **Age Distribution Protection** | No | No | No | No | No | Yes | Yes |
| **Min Device** | 1 | 1 | 1 | 3 | 4 | 4 | 4 |
| **Memory Overhead for Mapping** [4] | No | No | No | No | Yes | No | Yes |
| **Storage Overhead** | No | No | No | Parity | Parity + a spare device | Parity + a spare device | Parity + a spare device |
| **Write Amplification** [5] | None | None | None | High | High | Low | Low |

## 8.2 Review of Contributions

The literature review revealed that RAID-based reliability enhancement mechanisms have serious non-deterministic behaviour which limit their usage in hard real-time applications. Existing real-time GC mechanisms can effectively be used in single flash memory, but they are not fully adoptable with system designs where flash memory is used in a concurrent architecture such as RAID with strictly managed age distribution mechanisms. Moreover, existing GC techniques in flash RAID architecture generally focus on reliability and performance problems, ignoring the deterministic response time of the system.

[1] Here it is considered the complete product, with the on-line device replacement mechanism

[2] The terms low, normal and high refers rough comparison of the mechanisms for a given feature. Please see Chapter 7 and relevant papers for detailed experimental comparisons.

[3] This shows that additional memory blocks are required for guaranteed access time for I/Os, or not.

[4] It shows either mechanism requires additional memory for special mapping techniques such NvS-RAM memory used in the architecture.

[5] Only the write amplification factor of the device replacement process is considered.

In this thesis, the current real-time support and performance concerns in SSD-based RAID arrays were addressed effectively by introducing real-time garbage collectors mechanisms (Dynamic PGC and GARM) and on-line device replacement techniques for SSD RAID. The findings of this research were gathered in a form of final product as real-time and reliable flash-based storage device. Table 8.1 presents a comparison between the presented techniques (Dynamic PGC and GARM with the on-line device replacement support) with existing real-time and reliability enhancement mechanisms. The table clearly shows that the GARM outperforms the existing techniques in terms of WCET with higher performance and by maintaining the age distribution percentages. Moreover, PGC RAID could be seen as an alternative to the GARM when the memory overhead of the address mapping table is considered.

To achieve the research goal, the existing real-time solutions for an SSD array were first investigated. It was revealed that the existing techniques were not efficient for the architecture where flash-based devices are used in an array with a reliability enhancement technique for number of reasons. The first reason was that GC is triggered with a free rein across the array, which caused deviations in the strict age control mechanism of the reliability techniques. Second, the techniques were not capable of eliminating the performance overhead of the most expensive atomic operation (erase) in the cleaning process, and thus they did not provide a reduced WCET nor higher performance. Moreover, it was revealed that the reliability enhancement mechanisms host serious non-deterministic behaviours when a device replacement is required. These limitations were motivation for the development of novel real-time garbage collection mechanisms and device replacement techniques. The development of the mechanisms/techniques was followed by implementation in an FPGA-based SSD RAID architecture for evaluation. A detailed review of the thesis contributions is presented in the following sections.

### 8.2.1 Dynamic PGC Mechanism

The research work investigated existing real-time GC mechanisms. Since the PGC mechanism does not require additional memory blocks to provide a deterministic access guarantee, it was used in an SSD array with uneven parity redistribution ratios. To experimentally observe its effect on the reliability techniques, a number of experiments were conducted with a realistic trace. Based on the results, it was noted that the ratios were significantly disturbed by the uncoordinated GC mechanism. To address this limitation, a dynamic PGC mechanism was presented with two main components including Dynamic GC and pre-emptive mode controller.

**Dynamic GC**

The existing real-time GC techniques were only triggered based on remaining free space in corresponding memory. However, this creates a problem for the system where a strict age control mechanism is required. Since the PGC mechanism offers static GC and there is no optimal GC triggering threshold for the optimum age distribution of the reliability enhancement mechanisms, a dynamic GC technique was developed.

Dynamic GC dynamically adjusts the GC triggering thresholds by not only considering the remaining free space, but also the current and optimum age levels. This achieved a dynamically changing ageing speed for the corresponding memory and thus the deviation in age distribution was minimised in the array. Moreover, to increase the GC efficiency and avoid additional erase operations, a new feature of GC efficiency is proposed, which only permits cleaning of blocks with a certain number of invalid pages.

**Pre-emptive Mode Controller**

Continuous PGC may result in a lack of free space that may ultimately result in a system crash. To prevent this, the PGC mechanism offers number of states to define priority of GC over I/Os. These states are only changed depending on the amount of

free space left in the memory. Once the PGC is applied to a RAID array with an uneven parity distribution, it was found that uncoordinated PGC could result in an increase in the downtime period and reduced performance.

To address this problem, a pre-emptive mode controller was introduced. If the mechanism changes the GC state of a device in an array, it also checks other devices to trigger GC if their GC efficiency is relatively high. This encourages parallel cleaning in the array and thus reduces any downtime periods caused by cleaning and improves the system performance further.

### 8.2.2 GC-aware RAID Mechanism

Although the Dynamic PGC mechanism offers a deterministic response time by considering the reliability mechanisms, it was found that the mechanism was not fully efficient in several of its aspects. First, the improvement in WCET is limited, and there is no way to eliminate the overhead of the erase operation. Second, the mechanism considered the age distribution mechanism while triggering/postponing GC operations but the mechanism was not effectively reaching the target ratios, especially in the case of sequential write-dominant operations.

To benefit from concurrent system architecture further and to address these limitations, a GC-aware RAID mechanism was introduced with a number of new features: GC-aware Read/Write, a serialised GC, and an on-line parity migration.

**GC-aware Read/Write**

The mechanism dynamically locates incoming I/O requests to be non-blocked by an ongoing GC operation. Two separate features were presented for random and sequential GC-aware writes. Since a random write does not occupy all the devices in the array, chunks of the random write are dynamically located to GC-free devices. Thus, non-deterministic latency of GC is eliminated.

For sequential writes, a new forced random write feature was adopted, which is similar to that of the reliability mechanism. However, as opposed to the existing technique, each piece of partial write behaved as a new random write so as not to be blocked by a cleaning process.

For both random and sequential GC-aware read operations, the mechanism utilized existing parity data in the case of an active GC on a target device. The target device with a GC is considered as a failed device in the RAID mechanism, and a reconstruction mechanism was proposed to calculate the actual data.

If multiple GC processes are activated simultaneously on different devices of the array then the mechanism cannot provide a deterministic response time for I/Os, especially for read requests. This is because the parity techniques used in the reliability mechanism are only available to reconstruct a single device and data across multiple devices, so with an ongoing GC cannot be readable in a deterministic way. Therefore, a serialised GC feature was introduced.

**Serialised GC**

The serialised GC feature of the GARM was presented to permit single active garbage collector on the array at any given time. It was developed as the core feature of the mechanism to provide a deterministic access guarantee for I/Os. One of the bottlenecks of the mechanism is that a system failure may result from not producing enough free space, especially for the sequential-dominant traces. Although the research scope was defined to address random-dominant traces, to improve system dependability further and eliminate this bottleneck, a number of serialised GC states were defined. The states were defined as No-GC, serialised GC, prioritised GC and Normal GC.

**On-line Parity Migration**

One of the thesis research goals was to maintain the reliability mechanism while offering a real-time access guarantee to the system. The reliability mechanism is tightly coupled with the age distribution ratios of the array devices, which is directly related

to parity distribution ratios. It was observed that the GARM disturbed these ratios due to its GC-aware read/write features in the existence of an ongoing GC mechanism. To address this, a new on-line parity migration feature was incorporated to the mechanism. The OPM balances the parity distribution ratios towards the optimum levels by migrating parity between the devices with only a minor performance overhead. The GARM was achieved to maintain the parity distribution ratios required for the reliability enhancement with the help of the OPM feature.

### 8.2.3 On-line Device Replacement Techniques

After developing solutions to the non-deterministic behaviour of GC for flash memory-based devices, further investigations were carried out on the reliability mechanism. It was revealed that the reliability enhancement mechanisms have serious reliability issues for high integrity systems. Particularly during a device replacement process, the system needs to go off-line until the device reconstruction is completed, which significantly effects system availability and creates non-deterministic behaviour. The literature review indicates that existing on-line device replacement mechanisms are based on HDDs, which cannot be effectively adopted on SSD RAID mechanisms. To address this, a number of on-line device replacement techniques were designed for an SSD RAID framework.

**Proactive Hot-Swapping**

Once a flash memory exceeds its endurance limit then the data it stores cannot be considered reliable due to a high BER rate. To replace the device with a spare, an expensive reconstruction is necessary in parity-based RAID techniques. Moreover, the system has to stay off-line while replacing the expired device in the existing reliability enhancement mechanisms.

To address this bottleneck, a proactive hot-swapping technique was introduced. Since the failure of flash-based devices due to wearing out can be predictable, the mechanism starts data migration from an almost-expired device to the new one. Thus, an expensive data reconstruction process can be replaced with a proactive migration process whilst still providing continuous system availability.

**Coordinated Data Migration**

On-line rebuilding or cold data migration are usually initiated when there is an idle time or low density workload pattern detected. The default garbage collector of the reliability mechanism is also based on an idle time detection technique. This may result in performance overheads due to a possible overlap between the GC and the migration of data.

To overcome this, a coordinated data migration mechanism was introduced. It globally monitors the garbage collectors in the array via the Global FTL layer. If GC and device replacement functions attempt to access the target device in the same idle time period, the technique reschedules them depending on their priority levels to guarantee a response time.

**Cost-efficient Parity Redistribution**

Unlike other RAID techniques, the reliability enhancement mechanisms require parity redistribution during device replacement. It was noted that their device replacement techniques do not directly redistribute the parity data due to the reconstruction process. This increases the number of extra physical writes (write amplification) and the device replacement time.

To address this issue, a cost-efficient parity redistribution mechanism is presented which directly relocates the parity data from the most aged device to the other devices in the array according to target ratios. It improves device replacement time while reducing the write amplification effect.

**Semi-hybrid RAID**

To improve the system performance further using the concurrent architectural design of the system, a semi-hybrid RAID was presented. The proactive hot-swapping technique initiates data migration before the device reaches its endurance limit. After the migration process, there will be two valid copies of the same data on two independent devices, which is analogous to a RAID 1 architecture. The redundant data can be utilized to improve the read performance of the system. Semi-hybrid RAID configures the new and old device as RAID 1 whilst adding the new device to the default RAID configuration. Since the old device is not allowed to update its data, it can only serve non-modified data after replacement; hence the term semi-hybrid RAID.

### 8.2.4 Development Platform

In this thesis, a software-based test methodology was presented to develop a configurable and flexible debug environment for the design of a real-time and reliable SSD storage system. Each component of the design architecture was mapped into the well-known and validated Microsoft DiskSim simulator. The existing simulator was extended to meet the requirements of the system architecture. Global FTL with real time GC mechanisms, on-line device replacement techniques, dynamic address mapping, dynamic page allocation and the specific RAID controller with uneven parity distributions were implemented in the simulator.

To evaluate the system under a number of the different scenarios both synthetic and realistic traces were used. For synthetic workloads, some parameters were varied, including probability of sequential access, inter-arrival time of requests, sizes of the requests with distribution functions to produce traces similar to a realistic scenario. Moreover, a number of realistic traces were used from embedded and server applica-

tions. To conduct fair performance analyses, the entire memory was filled with valid data for the performance experiments.

This test environment was developed to validate the system against existing solutions in terms of data reliability for real-time applications and performance. By contrast, the novel flash based RAID storage architecture with real-time support presented in this thesis produced outstanding results.

This thesis mainly presents practical solutions for current real-time support, reliability and performance concerns in SSD RAID storage systems. On the other hand, a number of research points need further investigation, such as measuring the implementation cost of each of the presented techniques on a system-on-chip design. Investigation of these points would considerably lengthen the time required for any potential research work. Brief guidelines to such research are discussed in this section.

## 8.3 Future Work

### 8.3.1 Next target: Determining Real Estate Efficiency on FPGAs

For embedded system designs it is of considerable importance to determine implementation costs in terms of the number of gate counts they will consume. To measure this, the mechanism first needs to be mapped into a system-on-chip by converting them to Register Transfer Level (RTL). One of the ways to implement these RTL are by utilizing a synthesizable hardware description language such Verilog or VHDL. As a first step toward this aim, a simulation environment was implemented to analyse the RTL designs in order to investigate real-time support issues of flash-based mechanisms [101]. The simulation platform also allows for synthesizing the RTL design and for determining consumed hardware resources.

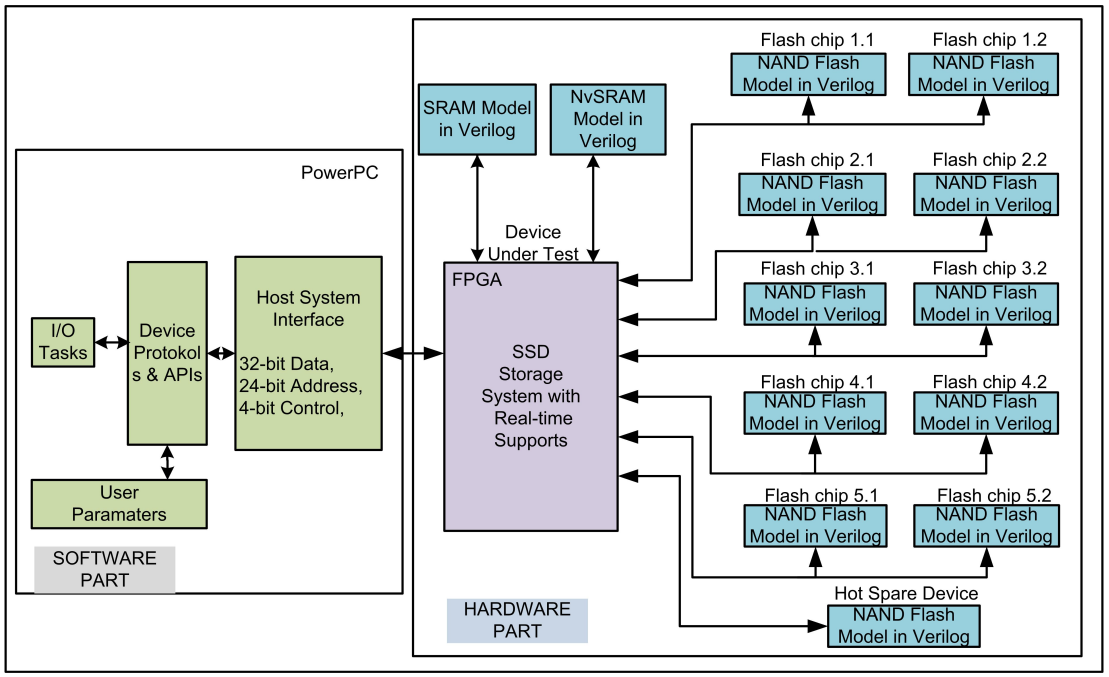



Figure 8.1: An FPGA-based SSD development platform

The proposed system architecture to determine real estate efficiency of the presented techniques is presented in Figure 8.1. It mainly consists of two parts: hardware and software. These techniques can be mapped into the FPGA to be synthesized. The FPGA communicates with the memories—for storing the actual data and the metadata— application layer through the software part. The software part helps to test the mechanism with different types of workload whilst eliminating complicated internal architecture via device protocols and Application Programming Interfaces (APIs). The main part of the architecture has already been developed, and initial simulations were conducted without real-time support techniques. The next task is to develop the Register Transfer Level (RTL) design of the presented mechanisms/techniques and to determine their cost efficiency in terms of the hardware resources they consume.

### 8.3.2 Improving Performance under Sequential Writes

Pure sequential writes lead to two main bottlenecks for the proposed mechanisms, as discussed in this thesis. First, the reliability mechanism is significantly affected as it reduces the wear imbalance among the elements of the array. Second, once a pure sequential write arrives at an array, its chunks occupy each device, which prevents the GARM mechanism from working properly. To address these problems, forced random feature was proposed. However, the forced random write has some drawbacks. First, it reduces the overall response time, thereby reducing concurrent access to the memory. Second, since each randomized write causes an additional parity write, the write amplification factor would significantly increase in the case of a sequential write-dominant workload. Since there is a trade-off between system reliability and performance, the forced random write mechanisms needs further investigation.

## 8.4 Final Remarks

The aim of this thesis was to propose a suitable flash based RAID storage architecture with real-time support. Although some existing real-time mechanisms were proposed to provide deterministic performance for a single flash memory unit, they were not fully compatible with an architecture where multiple flash memories are used in a concurrent architecture design such as RAID. Furthermore, the reliability enhancement mechanisms for SSD RAID have further non-deterministic behaviour, especially when a device replacement operation is performed. These were the motivations to explore different aspects of flash-based RAID storage systems.

In this thesis, three novel mechanisms were introduced to address real-time support issues of SSD RAID systems. These mechanisms were developed and tested in a validated SSD simulator. The mechanisms consist of a number of features which

provide deterministic access guarantees to the storage and improve the system performance regardless of the workload characteristics. Moreover, they achieve increased system availability, improved efficiency of the device replacement in terms of the performance, replacement time and reduced write amplification.

To conclude, the research goals of the thesis— a reliable and real-time mechanism for SSDs to use on-board high integrity embedded systems—have been achieved. The experimental results show promising results for the proposed novel GC mechanisms and the device replacement techniques compared to existing studies in the literature. This research work is one in a group of studies that are intended to provide some enhancement to high integrity storage systems. It also creates a new perspective towards reliable storage systems for hard real-time applications.

# References


[1] J. Ganssle and M. Barr, *Embedded Systems Dictionary*. CMP Books, 2003.

[2] P. Bondyopadhyay, "Moore's Law Governs the Silicon Revolution," *Proceedings of the IEEE*, vol. 86, pp. 78–81, Jan 1998.

[3] X. Lin and M. Chan, "An Opposite Side Floating Gate FLASH Memory Scalable to 20 nm Length," in *IEEE International 2002 SOI Conference*, pp. 71–72, Oct 2002.

[4] R. Chen, Y. Wang, J. Hu, D. Liu, Z. Shao, and Y. Guan, "Unified Non-volatile Memory and NAND Flash Memory Architecture in Smartphones," in *2015 20th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 340–345, Jan 2015.

[5] M. Helm, J.-K. Park, A. Ghalam, J. Guo, C. wan Ha, C. Hu, H. Kim, K. Kavalipurapu, E. Lee, A. Mohammadzadeh, D. Nguyen, V. Patel, T. Pekny, B. Saiki, D. Song, J. Tsai, V. Viajedor, L. Vu, T. Wong, J. H. Yun, R. Ghodsi, A. d'Alessandro, D. Di Cicco, and V. Moschiano, "19.1 A 128Gb MLC NAND-Flash Device Using 16nm Planar Cell," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 326–327, Feb 2014.

[6] M. Durna, H. Urhan, O. Turhan, O. Kozal, and M. Gurun, "A New Generation On-Board Computer and Solid State Data Recorder suitable for SpaceWire Platforms," in *3rd International Conference on Recent Advances in Space Technologies*, pp. 429–432, IEEE, June 2007.

[7] M. Caramia, S. Di Carlo, M. Fabiano, and P. E. Prinetto, "Flash-memories in Space Applications: Trends and Challenges," in *IEEE 7th East-West Design & Test Symposium (EWDTS)*, pp. 429–432, 2009.

[8] M. Balakrishnan, A. Kadav, V. Prabhakaran, and D. Malkhi, "Differential RAID: Rethinking RAID for SSD Reliability," *ACM Transactions on Storage (TOS)*, vol. 6, pp. 4:1–4:22, July 2010.

[9] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L. R. Nevill, "Bit Error Rate in NAND Flash Memories," in *2008 IEEE International Reliability Physics Symposium*, pp. 9–19, April 2008.

[10] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: High-performance, Reliable Secondary Storage," *ACM Computer Survey*, vol. 26, pp. 145–185, June 1994.

[11] I. Mir and A. McEwan, "A Reliability Enhancement Mechanism for High-Assurance MLC Flash-Based Storage Systems," in *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, vol. 1, pp. 190–194, Aug 2011.

[12] NASA, "Lesson(s) Learned: MER Spirit Flash Memory Anomaly: NASA Lessons (August, 2004)." Available: `http://llis.nasa.gov/lesson/1483`. (Last accessed: 21.11.2016).

[13] C.-H. Chen, C.-T. Chen, and W.-T. Huang, "The Real-Time Compression Layer for Flash Memory in Mobile Multimedia Devices," *Mobile Networks and Applications*, vol. 13, no. 6, pp. 547–554, 2008.

[14] S. Madden, W. Hong, J. Hellerstein, and K. Stanek, "Tinydb: A declarative database for sensor networks (August, 2003)." Available: `http://telegraph.cs.berkeley.edu/tinydb/`. (Last accessed: 21.11.2016).

[15] L.-P. Chang, T.-W. Kuo, and S.-W. Lo, "Real-time Garbage Collection for Flash-memory Storage Systems of Real-time Embedded Systems," *ACM Transactions on Embedded Computing Systems*, vol. 3, pp. 837–863, Nov. 2004.

[16] S. Choudhuri and T. Givargis, "Deterministic Service Guarantees for NAND Flash Using Partial Block Cleaning," in *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '08, pp. 19–24, 2008.

[17] J. Lee, Y. Kim, G. Shipman, S. Oral, and J. Kim, "Preemptible I/O Scheduling of Garbage Collection for Solid State Drives," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, pp. 247–260, Feb 2013.

[18] Z. Qin, Y. Wang, D. Liu, and Z. Shao, "Real-Time Flash Translation Layer for NAND Flash Memory Storage Systems," in *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, pp. 35–44, April 2012.

[19] M. Holland, G. Gibson, and D. Siewiorek, "Fast, On-line Failure Recovery in Redundant Disk Arrays," in *The Twenty-Third International Symposium on Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers*, pp. 422–431, IEEE, June 1993.

[20] J. Y. B. Lee and J. C. S. Lui, "Automatic Recovery from Disk Failure in Continuous-Media Servers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, pp. 499–515, May 2002.

[21] L. Tian, D. Feng, H. Jiang, K. Zhou, L. Zeng, J. Chen, Z. Wang, and Z. Song, "PRO: A Popularity-based Multi-threaded Reconstruction Optimization for RAID-structured Storage Systems," in *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, pp. 277–290, USENIX Association, 2007.

[22] B. Boehm, "A Spiral Model of Software Development and Enhancement," *IEEE Computer Journal*, vol. 21, pp. 61–72, May 1988.

[23] Intel, "Understanding the Flash Translation Layer (FTL) Specification," tech. rep., Dec. 1998.

[24] D. Woodhouse, "JFFS: The Journalling Flash File System," in *Ottawa Linux Symposium*, vol. 2001, 2001. (Last accessed: 21.11.2016).

[25] A. One, "YAFFS: Yet Another Flash File System (2002)." Available: `http://www.yaffs.net/`. (Last accessed: 21.11.2016).

[26] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings," *ACM SIGPLAN Notices - ASPLOS 2009*, vol. 44, pp. 229–240, Mar. 2009.

[27] A. Ban, "Flash File System Optimized for Page-mode Flash Technologies," Aug. 10 1999. US Patent 5,937,425.

[28] Y. Wang, D. Liu, M. Wang, Z. Qin, Z. Shao, and Y. Guan, "RNFTL: A Reuse-aware NAND Flash Translation Layer for Flash Memory," *ACM SIGPLAN Notices - LCTES '10*, vol. 45, pp. 163–172, Apr. 2010.

[29] H. Cho, D. Shin, and Y. I. Eom, "KAST: K-Associative Sector Translation for NAND flash memory in real-time systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pp. 507–512, European Design and Automation Association, 2009.

[30] Z. Qin, Y. Wang, D. Liu, Z. Shao, and Y. Guan, "MNFTL: An Efficient Flash Translation Layer for MLC NAND Flash Memory Storage Systems," in *Proceedings of the 48th Design Automation Conference*, DAC 2011, pp. 17–22, ACM, 2011.

[31] S. J. Kwon and T.-S. Chung, "An Efficient and Advanced Space-management Technique for Flash Memory Using Reallocation Blocks," *IEEE Transaction on Consumer Electronics*, vol. 54, pp. 631–638, May 2008.

[32] J. Lee, A. Kim, M. Park, J. Choi, D. Lee, and S. H. Noh, "Real-time Flash Memory Storage with Janus-FTL," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pp. 1799–1806, 2012.

[33] E. Gal and S. Toledo, "Algorithms and Data Structures for Flash Memories," *ACM Computing Surveys (CSUR)*, vol. 37, pp. 138–163, June 2005.

[34] Y. Kim, J. Lee, S. Oral, D. Dillow, F. Wang, and G. Shipman, "Coordinating Garbage Collection for Arrays of Solid-State Drives," *IEEE Transactions on Computers*, vol. 63, pp. 888–901, April 2014.

[35] R. Jones, A. Hosking, and E. Moss, *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 2011.

[36] J. Liu, S. Chen, T. Wu, and H. Zhang, "A Novel Hot Data Identification Mechanism for NAND Flash Memory," *IEEE Transactions on Consumer Electronics*, vol. 61, pp. 463–469, November 2015.

[37] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," in *USENIX 2008 Annual Technical Conference*, ATC'08, pp. 57–70, USENIX Association, 2008.

[38] E. H. Nam, B. Kim, H. Eom, and S. L. Min, "Ozone (O3): An Out-of-Order Flash Memory Controller Architecture," *IEEE Transactions on Computers*, vol. 60, pp. 653–666, May 2011.

[39] S. Jose and C. Pradeep, "Design of a Multichannel NAND Flash Memory Controller for Efficient Utilization of Bandwidth in SSDs," in *2013 International Multi-Conference on Automation, Computing, Communication, Control and Compressed Sensing (iMac4s)*, pp. 235–239, IEEE, March 2013.

[40] B. He, J. Yu, and A. Zhou, "Improving Update-Intensive Workloads on Flash Disks through Exploiting Multi-Chip Parallelism," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, pp. 152–162, Jan 2015.

[41] E. Deal, "Trends in NAND Flash Memory Error Correction, Cyclic Design (June, 2009)." Available: `http://www.cyclicdesign.com/index.php/ecc-trends-in-nand-flash`. (Last accessed: 21.11.2016).

[42] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf, "Characterizing Flash Memory: Anomalies, Observations, and Applications," in *42nd Annual IEEE/ACM International Symposium on Microarchitecture, 2009. MICRO-42*, pp. 24–33, Dec 2009.

[43] K. Park, D.-H. Lee, Y. Woo, G. Lee, J.-H. Lee, and D.-H. Kim, "Reliability and Performance Enhancement Technique for SSD Array Storage System Using RAID Mechanism," in *9th International Symposium on Communications and Information Technology*, pp. 140–145, Sept 2009.

[44] B. Mao, H. Jiang, S. Wu, L. Tian, D. Feng, J. Chen, and L. Zeng, "HPDA: A Hybrid Parity-based Disk Array for Enhanced Performance and Reliability," *ACM Transactions on Storage*, vol. 8, pp. 4:1–4:20, Feb. 2012.

[45] S. Rizvi and T.-S. Chung, "Data Storage Framework on Flash Memory Based SSD RAID 0 for Performance Oriented Applications," in *The 2nd International Conference on Computer and Automation Engineering (ICCAE)*, vol. 1, pp. 126–128, Feb 2010.

[46] Y. Oh, J. Choi, D. Lee, and S. H. Noh, "Improving Performance and Lifetime of the SSD RAID-based Host Cache Through a Log-structured Approach," in *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '13, pp. 5:1–5:8, ACM, 2013.

[47] A. McEwan and M. Komsul, "On-Line Device Replacement Techniques for SSD RAID," in *2015 Euromicro Conference on Digital System Design (DSD)*, pp. 438–444, IEEE, Aug 2015.

[48] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives," *ACM SIGMETRICS Performance Evaluation Review - SIGMETRICS '09*, vol. 37, pp. 181–192, June 2009.

[49] J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang, "Efficient Identification of Hot Data for Flash Memory Storage Systems," *ACM Transactions on Storage*, vol. 2, pp. 22–40, Feb. 2006.

[50] S. Lee, D. Shin, Y.-J. Kim, and J. Kim, "LAST: Locality-aware Sector Translation for NAND Flash Memory-based Storage Systems," *ACM SIGOPS Operating Systems Review*, vol. 42, pp. 36–42, Oct. 2008.

[51] J. A. Stankovic, "Misconceptions about Real-time Computing: A Serious Problem for Next-generation Systems," *Computer*, vol. 21, pp. 10–19, Oct 1988.

[52] R. Oshana, *Software Engineering for Embedded Systems: Methods, Practical Techniques, and Applications*. Newnes, 2013.

[53] S. J. Kwon, A. Ranjitkar, Y.-B. Ko, and T.-S. Chung, "FTL Algorithms for NAND-type Flash Memories," *Design Automation for Embedded Systems*, vol. 15, no. 3, pp. 191–224, 2011.

[54] "Intel X25-E Extreme SATA Solid-State Drives, DataSheet (Jan, 2008)." Available: `ftp://download.intel.com/design/flash/NAND/extreme/extreme-sata-ssd-product-brief.pdf`. (Last accessed: 01.05.2016).

[55] "Samsung SSD White Paper (2013)." Available: `http://www.samsung.com/semiconductor/minisite/ssd/download/overview.html`. (Last accessed: 21.11.2016).

[56] Micron, "Nand Flash 101: An Introduction to NAND Flash and How to Design It In to Your Next Product," tech. rep., 2010.

[57] Y. Yong-Tae and S.-K. Jo, "Memory System and Wear-leveling Method Thereof based on Erasures and Error Correction Data," Jan. 5 2016. US Patent 9,229,805.

[58] A. Kosuge, J. Hashiba, T. Kawajiri, S. Hasegawa, T. Shidei, H. Ishikuro, T. Kuroda, and K. Takeuchi, "Inductively-powered Wireless Solid-state Drive (SSD) system with Merged Error Correction of High-speed Non-contact Data Links and NAND Flash Memory," in *2015 Symposium on VLSI Circuits (VLSI Circuits)*, pp. C128–C129, June 2015.

[59] E. Zhou, Y. Lu, N. Xiao, Y. Ou, Z. Chen, and X. Bao, "A theoretical analysis of lifespan impact on flash memory imposed by erasure code," in *2015 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pp. 361–362, Aug 2015.

[60] C. Kim, C. Park, S. Yoo, and S. Lee, "Extending Lifetime of Flash Memory Using Strong Error Correction Coding," *IEEE Transactions on Consumer Electronics*, vol. 61, pp. 206–214, May 2015.

[61] B. Chen, X. Zhang, and Z. Wang, "Error Correction for Multi-level NAND Flash Memory Using Reed-Solomon Codes," in *IEEE Workshop on Signal Processing Systems*, pp. 94–99, Oct 2008.

[62] M. Blaum, J. Brady, J. Bruck, and J. Menon, "EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures," *IEEE Transactions on Computers*, vol. 44, pp. 192–202, Feb 1995.

[63] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar, "Row-diagonal Parity for Double Disk Failure Correction," in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST'04, pp. 1–1, USENIX Association, 2004.

[64] S. Im and D. Shin, "Flash-Aware RAID Techniques for Dependable and High-Performance Flash Memory SSD," *IEEE Transactions on Computers*, vol. 60, no. 1, pp. 80–92, 2011.

[65] J.-W. Hsieh and M.-X. Liu, "Configurable Reliability Framework for SSD-RAID," in *Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pp. 1–6, IEEE, Aug 2014.

[66] J. Colgrove, J. Hayes, B. Hong, and E. Miller, "Adaptive RAID for an SSD Environment," Mar. 29 2012. US Patent App. 12/892,894.

[67] J. Kim, J. Lee, J. Choi, D. Lee, and S. Noh, "Efficient RAID Technique for Reliable SSD," Dec. 25 2014. US Patent App. 14/377,159.

[68] S. Chen and D. Towsley, "The Design and Evaluation of RAID 5 and Parity Striping Disk Array Architectures ," *Journal of Parallel and Distributed Computing*, vol. 17, no. 1-2, pp. 58 – 74, 1993.

[69] A. McEwan and I. Mir, "An Embedded FTL for SSD RAID," in *2015 Euromicro Conference on Digital System Design (DSD)*, pp. 575–582, IEEE, Aug 2015.

[70] L.-P. Chang and T.-W. Kuo, "An Efficient Management Scheme for Large-scale Flash-memory Storage Systems," in *Proceedings of the 2004 ACM Symposium on Applied Computing*, SAC '04, pp. 862–868, ACM, 2004.

[71] E. Stott and P. Cheung, "Improving FPGA Reliability with Wear-Levelling," in *2011 International Conference on Field Programmable Logic and Applications (FPL)*, pp. 323–328, Sept 2011.

[72] R. Micheloni, R. Ravasio, A. Marelli, E. Alice, V. Altieri, A. Bovino, L. Crippa, E. D. Martino, L. D'Onofrio, A. Gambardella, E. Grillea, G. Guerra, D. Kim, C. Missiroli, I. Motta, A. Prisco, G. Ragone, M. Romano, M. Sangalli, P. Sauro, M. Scotti, and S. Won, "A 4Gb 2b/cell NAND Flash Memory with Embedded 5b BCH ECC for 36MB/s System Read Throughput," in *2006 IEEE International Solid State Circuits Conference - Digest of Technical Papers*, pp. 497–506, Feb 2006.

[73] Q. Huang, S. Lin, and K. Abdel-Ghaffar, "Error-Correcting Codes for Flash Coding," *IEEE Transactions on Information Theory*, vol. 57, pp. 6097–6108, Sept 2011.

[74] Y.-H. Chang and T.-W. Kuo, "A Reliable MTD Design for MLC Flash-memory Storage Systems," in *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '10, pp. 179–188, ACM, 2010.

[75] T. Xie and Y. Sun, "PEARL: Performance, Energy, and Reliability Balanced Dynamic Data Redistribution for Next Generation Disk Arrays," in *IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems*, pp. 1–8, Sept 2008.

[76] S. Y. Park, E. Seo, J. Y. Shin, S. Maeng, and J. Lee, "Exploiting Internal Parallelism of Flash-based SSDs," *IEEE Computer Architecture Letters*, vol. 9, pp. 9–12, Jan 2010.

[77] K. Leung, W. Smith, and S. Wilheim, "Method and Apparatus for Analysis of User Traffic within a Predefined Area," Apr. 15 2014. US Patent 8,699,370.

[78] J. Bennett, "Memory System having Persistent Garbage Collection," Dec. 26 2012. EP Patent App. EP20,100,786,789.

[79] R. Goss and M. Gaertner, "Selection of Units for Garbage Collection in Flash Memory," Apr. 9 2013. US Patent 8,417,878.

[80] Y. Qin, D. Feng, J. Liu, W. Tong, and Z. Zhu, "DT-GC: Adaptive Garbage Collection with Dynamic Thresholds for SSDs," in *2014 International Conference on Cloud Computing and Big Data (CCBD)*, pp. 182–188, IEEE, Nov 2014.

[81] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write Amplification Analysis in Flash-based Solid State Drives," in *Proceedings of SYSTOR*

*2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pp. 10:1–10:9, ACM, 2009.

[82] C. Lu, G. A. Alvarez, and J. Wilkes, "Aqueduct: Online Data Migration with Performance Guarantees," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, p. 21, USENIX Association, 2002.

[83] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes, "Idleness is Not Sloth," in *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pp. 17–17, USENIX Association, 1995.

[84] "Storage - UMass Trace Repository (June, 2007)." Available: `http://traces.cs.umass.edu/index.php/Storage/Storage`.

[85] I. Mir and A. McEwan, "A High Performance Reconfigurable Flash Management Framework," in *2014 International Conference on Information Science, Electronics and Electrical Engineering (ISEEE)*, vol. 2, pp. 1216–1220, IEEE, April 2014.

[86] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda, "Characterization of Storage Workload Traces from Production Windows Servers," in *IEEE International Symposium on Workload Characterization*, pp. 119–128, Sept 2008.

[87] I. Mir, *Reliability Management Techniques in SSD Storage Systems*. PhD thesis, University of Leicester, 2013.

[88] A. McEwan and I. Mir, "Age Distribution Convergence Mechanisms for Flash Based File Systems," *Journal of Computers (JCP), Academy Publisher*, vol. 7, pp. 988–997, April 2012.

[89] S. Tanakamaru, S. Hosaka, K. Johguchi, H. Takishita, and K. Takeuchi, "Understanding the Relation Between the Performance and Reliability of NAND

Flash/SCM Hybrid Solid-State Drive," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. PP, no. 99, pp. 1–12, 2015.

[90] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger, "The Disksim Simulation Environment Version 4.0 Reference Manual (cmu-pdl-08-101)," *Parallel Data Laboratory*, p. 26, 2008.

[91] Y. Cai, Y. Luo, E. Haratsch, K. Mai, and O. Mutlu, "Data Retention in MLC NAND Flash Memory: Characterization, Optimization, and Recovery," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 551–563, Feb 2015.

[92] Y. Lu, J. Shu, J. Guo, S. Li, and O. Mutlu, "High-Performance and Lightweight Transaction Support in Flash-Based SSDs," *IEEE Transactions on Computers*, vol. 64, pp. 2819–2832, Oct 2015.

[93] Y. Winata, S. Kim, and I. Shin, "Enhancing Internal Parallelism of Solid-state Drives while Balancing Write Loads Across Dies," *Electronics Letters*, vol. 51, no. 24, pp. 1978–1980, 2015.

[94] C. Wang and S. Baskiyar, "Extending Flash Lifetime in Secondary Storage," *Microprocessors and Microsystems*, vol. 39, no. 3, pp. 167 – 180, 2015.

[95] J. Thomson, *High Integrity Systems and Safety Management in Hazardous Industries*. Butterworth-Heinemann, 2015.

[96] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, pp. 203–215, Feb 2007.

[97] N. Battezzati, L. Sterpone, and M. Violante, *Reconfigurable Field Programmable Gate Arrays for Mission-critical Applications*. Springer Science & Business Media, 2010.

[98] P. Adell and G. Allen, *Assessing and Mitigating Radiation Effects in Xilinx FPGAs*. Pasadena, CA: Jet Propulsion Laboratory, California Institute of Technology, 2008.

[99] F. L. Kastensmidt, G. Neuberger, L. Carro, and R. Reis, "Designing and Testing Fault-tolerant Techniques for SRAM-based FPGAs," in *Proceedings of the 1st Conference on Computing Frontiers*, pp. 419–432, ACM, 2004.

[100] P. Olivier, J. Boukhobza, and E. Senn, "Flashmon v2: Monitoring Raw NAND Flash Memory I/O Requests on Embedded Linux," *ACM SIGBED Review - Special Issue on the 3rd Embedded Operating System Workshop*, vol. 11, no. 1, pp. 38–43, 2014.

[101] M. Komsul, A. McEwan, and I. Mir, "An FPGA-based Development Platform for Real-time Solid State Devices," in *2014 International Conference on Information Science, Electronics and Electrical Engineering (ISEEE)*, vol. 2, pp. 1198–1203, IEEE, April 2014.

# Appendix A

# Development Platform Structure

## A.1 Default SSD configuration

```
ssdmodel_ssd SSD {
// this is a percentage of reserved total pages in the ssd
Reserve pages percentage = 15,

// min percentage of free blocks needed. if the free
// blocks drop below this, cleaning kicks in
Minimum free blocks percentage = 5,

// a simple read-modify-erase-write policy = 1 (no longer supported)
// this is out-of-place write policy= 2
Write policy = 2,

// random = 1 (not supp), greedy = 2, wear-aware = 3
Cleaning policy = 2,

//  number of planes in each flash package (element)
Planes per package = 1,

//  number of flash blocks in each plane
Blocks per plane = 2048,

//  how the blocks within an element are mapped on a plane
// simple concatenation = 1, plane-pair stripping = 2 (not tested),
// full stripping = 3
Plane block mapping = 3,
```

```
// copy-back enabled (1) or not (0)
Copy back = 0,

// how many parallel units are there?
// entire elem = 1, two dies = 2, four plane-pairs = 4
Number of parallel units = 1,

// we use diff allocation logic: chip//plane
// each gang = 0, each elem = 1, each plane = 2
Allocation pool logic = 1,

// elements are grouped into a gang
Elements per gang = 1,

// shared bus (1) or shared control (2) gang
Gang share = 1,

// when do we want to do the cleaning?
Cleaning in background = 1,

Flash chip elements = 2,
Page size = 4,
Pages per block = 64,

// Cchanging the no of blocks from 16184 to 16384
Blocks per element = 2048,

// Changing the Chip xfer latency from per sector to per byte
Chip xfer latency = 0.000005,

Page read latency = 0.025,
Page write latency = 0.200,
Block erase latency = 1.5,
}  // end of SSD spec
```

## A.2 SSD Array Configuration

```
disksim_logorg org0 { // Logical Organization of the SSD array
    Addressing mode = Dynamic,// dynamic stripe mapping is adapted
    Distribution scheme = Striped,
    Redundancy scheme = Flash-RAID,// the parity distribitions of the reliability mechanism
    Components= Whole,
    // devices in the array
    devices = [ ssd0 .. ssd4 ],

    Stripe unit  =  4, // stripe unit size kB
```

```
    RMW vs. reconstruct =  0.5,// this is a threshold ratio to calculate the new parity either using the old ↩
         data or new data.
    Parity stripe unit =  4,
}
```

## A.3 System Topology

```
instantiate [ ssd0 .. ssd4 ] as  SSD
instantiate [ bustop ]    as  BUSTOP
instantiate [ busHBA0 .. busHBA4 ]    as  BUSHBA
instantiate [ ctlrHBA0 .. ctlrHBA4 ]  as  CTLR0
instantiate [ driver0 ]         as  DRIVER0


// system topology
topology disksim_iodriver driver0 [
    disksim_bus bustop [

        disksim_ctlr ctlrHBA0 [
                disksim_bus busHBA0 [
                    ssdmodel_ssd ssd0 []
                                ]     // end of bus0
                            ],        // end of HBA0

        disksim_ctlr ctlrHBA1 [
            disksim_bus busHBA1 [
                    ssdmodel_ssd ssd1 []
                                ]     // end of bus1
                        ],        // end of HBA1

        disksim_ctlr ctlrHBA2 [
            disksim_bus busHBA2 [
                    ssdmodel_ssd ssd2 []
                                ]     // end of bus2
                        ],        // end of HBA2

        disksim_ctlr ctlrHBA3 [
            disksim_bus busHBA3 [
                    ssdmodel_ssd ssd3 []
                                ]     // end of bus3
                        ],        // end of HBA3

        disksim_ctlr ctlrHBA4 [
            disksim_bus busHBA4 [
                    ssdmodel_ssd ssd4 []
                                ]     // end of bus4
                        ]        // end of HBA4
```

```
            ]           // end of bustop
        ]               // end of driver0 (and system topology)
```

## A.4 Synthetic Workload Generator Configuration

```
disksim_synthgen { // generator 0
    Storage capacity per device  =  222822,
    devices = [ org0 ],
    Blocking factor =  4,
    Probability of sequential access =  0.2,
    Probability of local access =  0.0,
    Probability of read access =  0.2,
    Probability of time-critical request =  0.0,
    Probability of time-limited request =  0.0,
    Time-limited think times  = [ normal, 30.0, 100.0  ],
    General inter-arrival times  = [ uniform, 3.0, 3.0 ],
    Sequential inter-arrival times  = [ uniform, 3.0, 3.0  ],
    Local inter-arrival times  = [ exponential, 0.0, 0.0  ],
    Local distances  = [ normal, 0.0, 40000.0  ],
    Sizes  = [ exponential, 0.0, 3.0 ]
} // end of generator 0
] // end of generator list
```

## A.5 Uneven Parity Redistribution and Page Allocation

```
if((!(curr->flags & READ )) || (!(curr->flags & READ ) & ps[preblkno].first_dev==null)) {//make sure it is new ↩
      write

    if (counter<0.8*par){//80% of new parity to the second most aged device
            partsperstripe=n;//id of the most aged device for parity (initially 0)
            paritycounter[partsperstripe]++;
            if (counter<(0.8*par)/4){//define fist device of the actual data
                temp->devno = table[(n+1)].devno;
                                }
            else if ((0.8*par)/4<=counter & counter<(0.8*par)*2/4) {
                temp->devno = table[(n+2)].devno;
                                }
            else if ((0.8*par)*2/4<=counter & counter<(0.8*par)*3/4){
```

```
            temp->devno = table[(n+3)].devno;
                                }
            else if ((0.8*par)*2/4<=counter & counter<(0.8*par)*4/4){
            temp->devno = table[n+4].devno;
                                }
                        }
    else if (counter<0.99*par & counter>=0.8*par){ //19% of new parity to the second most aged device
            partsperstripe=n+1;
            paritycounter[partsperstripe]++;
            if ((0.8*par)<=counter & counter<(0.8*par)+((0.19*par)*1/4)){ //define fist device of the actual ↩
                  data
                temp->devno = table[(n)].devno;
                                }
            else if ((0.8*par)+((0.19*par)*1/4)<=counter & counter<(0.8*par)+((0.19*par)*2/4)){
                temp->devno = table[(n+2)].devno;
                                }
            else if ((0.8*par)+((0.19*par)*2/4)<=counter & counter<(0.8*par)+((0.19*par)*3/4)){
                temp->devno = table[(n+3)].devno;
                                }
            else if ((0.8*par)+((0.19*par)*3/4)<=counter & counter<(0.8*par)+((0.19*par)*4/4)){
                temp->devno = table[n+4].devno;
                                }
                        }
    else { //1% of new parity to the third most aged device
            partsperstripe=n+2;
            paritycounter[partsperstripe]++;
            if ((0.8*par)<=counter & counter<(0.8*par)+((0.19*par)*1/4)){ //define fist device of the actual ↩
                  data
                temp->devno = table[(n)].devno;
                                }
            else if ((0.8*par)+((0.19*par)*1/4)<=counter & counter<(0.8*par)+((0.19*par)*2/4)){
                temp->devno = table[(n+1)].devno;
                                }
            else if ((0.8*par)+((0.19*par)*2/4)<=counter & counter<(0.8*par)+((0.19*par)*3/4)){
                temp->devno = table[(n+3)].devno;
                                }
            else if ((0.8*par)+((0.19*par)*3/4)<=counter & counter<(0.8*par)+((0.19*par)*4/4)){
                temp->devno = table[n+4].devno;
                                }
                    }
}
```

## A.6 Dynamic Threshold Calculation

```
void dynamic_threshold(){
    unsigned int low;
    int i;
    float opt=0;
```

```
    ssd_t *s;
    for (i = 0; i < 5; i++)//dynamic threhold calculation for 5 array device
        {
            s = getssd (i);

    if (s->devno==0)
    {
        opt=0.29;//optimum age level
        s->dt = s->dt+(opt-s->agingratio);
    }
    else if (s->devno==1)
    {
        opt=0.29;//optimum age level
        s->dt = s->dt+(opt-s->agingratio);
    }
    else if (s->devno==2)
    {
        s->dt = s->dt+(opt-s->agingratio);
        opt=0.30;//optimum age level
    }
    else if (s->devno==3)
    {
        s->dt = s->dt+(opt-s->agingratio);
        opt=0.46;//optimum age level
    }
        else if (s->devno==4)//most aged device
    {
        s->dt = s->dt-(totalgap);
    }

totalgap+=s->agingratio-opt;
    if (s->dt<MT)//minimum boundary for threshold
    {
        s->dt=MT;
    }
        else if (s->dt>0.5)//max boundry for threshold
    {
        s->dt=0.5;
    }

}
```

## A.7 PGC Mode Controller

```
static void ssd_access_complete_element(ioreq_event *curr)// efter each access of the memory the mechanism ↩
      chechs either cleaning has started or not
{
```

```
    ssd_t *currdisk, *s1;
    int elem_num, i;
    ssd_element  *elem;
    ioreq_event *x;
    ssd_element_metadata *metadata ;
    plane_metadata *pm ;


    currdisk = getssd (curr->devno);
    elem_num = currdisk->timing_t->choose_element(currdisk->timing_t, curr->blkno);
    metadata = &(getssd (curr->devno)->elements[elem_num].metadata);
    ASSERT(elem_num == curr->ssd_elem_num);
    elem = &currdisk->elements[elem_num];
    pm = &metadata->plane_meta[0];

    // all the reqs are over
    if (ioqueue_get_reqoutstanding(elem->queue) == 0) {
        elem->media_busy = FALSE;
    }

    ssd_complete_parent(curr, currdisk);
    addtoextraq((event *) curr);
    ssd_activate_elem(currdisk, elem_num);

    if (pm->clean_in_progress) //if cleaning is invoked after an opeartion then the mechanism activate other ↩
          devices to check for cleaning
    {
        for (i = 0; i < n; i++)
        {
            s1 = getssd (i);
            if (currdisk->devno==s1->devno)// if it is current device skip it
            {
                i++;
                continue;
            }
            ssd_activate_elem(s1, elem_num);
        }
    }
}
```

# A.8 GC Efficiency Function

```
static double ssd_clean_blocks_greedy(int plane_num, int elem_num, ssd_t *s, int period) {

 for (i =0;  i <s->params.pages_per_block-1; i ++) {
 int j;
 usage_table *entry;
```

```
 // get the bucket of blocks with "i" valid pages
 entry = &(table[i]);

 // free all the blocks with "i" valid pages
 for (j = 0; j < entry->len; j ++) {
 int blk = entry->block[j];
 int block_life = metadata->block_usage[blk].rem_lifetime;


 // if the block is already dead, skip it
 if (block_life == 0) {
 continue;
 }

 // clean only those blocks that are sealed.
 if (ssd_can_clean_block(s, metadata, blk)) {
 continue;
}
 if (s->Current_Threshold>s->Medium_threshold)//// Make sure that current low priority threshold is not bigger ↩
      than medium threshold
 {
continue;
 }

if (s->params.pages_per_block*GCE_factor>i)//// Make sure that there are not many valid pages in the block
  {
  continue;
  }
 // finally here the block can be cleaned.
 // invoke cleaning
 cost += _ssd_clean_block_fully(blk, metadata->block_usage[blk].plane_num, elem_num, metadata, s);
 }
 }
 }
```