A Low Overhead and Consistent Flash Translation Layer for Embedded Devices Utilizing Serial NOR Flash

by

Scott Ronald Fazackerley

B.Sc. Hons., The University of British Columbia, 2008
M.Sc., The University of British Columbia, 2010

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

THE COLLEGE OF GRADUATE STUDIES

(Interdisciplinary Studies - Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Okanagan)

July 2016

© Scott Ronald Fazackerley, 2016

The undersigned certify that they have read, and recommend to the College of Graduate Studies for acceptance, a thesis entitled: A Low Overhead and Consistent Flash Translation Layer for Embedded Devices Utilizing Serial NOR Flash submitted by Scott Ronald Fazackerley in partial fulfillment of the requirements of the degree of Doctor of Philosophy

Supervisor, Professor (please print name and faculty/school above the line)

Supervisory Committee Member, Professor (please print name and faculty/school above the line)

Supervisory Committee Member, Professor (please print name and faculty/school above the line)

University Examiner, Professor (please print name and faculty/school above the line)

External Examiner, Professor (please print name and faculty/school above the line)

(Date Submitted to Grad Studies)

Additional Committee Members include:

(please print name and faculty/school above the line)

(please print name and faculty/school above the line)

Abstract

In today's world, embedded devices are playing an ever increasing and important role in daily life. Recent interest has focused on how data from embedded devices can be harnessed. Whether it is complex environmental parameters or the average temperature in our homes, devices store and process data. Due to the vast amount of data that is generated by devices, the ability to process data on device is beneficial as it reduces the amount of data that must be transferred off the device. Flash memory is the most commonly found storage medium on embedded devices but presents challenges for developers. Flash memory is managed through the use of a flash translation layer (*FTL*) to address the physical limitations of memory. No FTL is currently available for the smallest of devices due to resource limitations. This thesis examines the features that are currently used in FTLs and highlights their shortcomings for use with resource constrained devices. In response to these challenges, this work introduces an FTL architecture with a minimal RAM footprint that is robust and fault-tolerant for resource constrained embedded systems. Utilizing attributes of the Adesto serial NOR Dataflash memory, a unique flash translation layer for use with serial NOR flash has been developed. Key features focus on minimizing data transfer between host and flash while maintaining persistent address translation. In addition to a low overhead and robust flash translation layer, the FTL contains a deterministic garbage collection and wear levelling strategy. This work introduces the technique of masked overwriting for NOR flash, which demonstrates the use page overwriting for modification of specific data in place. This technique offers savings in terms of write times, page erases and complexity in managing data pages, resulting in a novel FTL strategy suitable for resource constrained devices.

Preface

The following items have already appeared in print:

- The results in Section 4 are joint work with my supervisor Dr. Ramon Lawrence and Wade Penson and appear in the paper "Write Improvement Strategies for Serial NOR Dataflash Memory" [SFL16] and were published in the proceeding of the 29th annual IEEE Canadian Conference on Electrical and Computer Engineering.
- The basis for the algorithms shown in Chapter 5 were published with my supervisor Dr. Ramon Lawrence in the paper "A Flash Resident File System for Embedded Sensor Networks" [FL11] and were published in the proceeding of the 24th annual IEEE Canadian Conference on Electrical and Computer Engineering.

Table of Contents

Abstract . . . iii

Preface . . . iv

Table of Contents . . . v

List of Tables . . . viii

List of Figures . . . ix

List of Symbols and Abbreviations . . . xi

Dedication . . . xiii

Chapter 1: Introduction . . . 1

Chapter 2: Storage Architectures . . . 8
- 2.1 Embedded Devices . . . 9
  - 2.1.1 Wireless Sensor Networks . . . 10
- 2.2 Memory Stores . . . 11
  - 2.2.1 EEPROM . . . 17
  - 2.2.2 Flash . . . 18
  - 2.2.3 Other Storage Technologies . . . 30

Chapter 3: Data Persistence Strategies . . . 33
- 3.1 Data Management Strategies . . . 33
- 3.2 Flash Aware File Systems . . . 34
  - 3.2.1 Log Based File Systems . . . 35
  - 3.2.2 Logging Based File Systems for Flash . . . 39
- 3.3 Flash Translation Layers . . . 42
  - 3.3.1 FTL Taxonomies . . . 43

3.4 FTL Schemes . . . 49
3.4.1 Page Level FTL Schemes . . . 50
3.4.2 Page Based Logging Schemes . . . 52
3.4.3 Block Based Logging Schemes . . . 57
3.4.4 Block Set FTL Schemes . . . 61
3.4.5 State Based FTL Schemes . . . 62
3.4.6 FTL Improvement Schemes . . . 65
3.4.7 Suitability of FTL Schemes for Serial NOR Memories 72
3.5 Open Research . . . 73

Chapter 4: Write Strategies for Serial NOR Flash . . . 76
4.1 Write Strategies for Improved Performance with Serial NOR Dataflash . . . 78
4.1.1 Write Operations for Serial NOR Flash . . . 79
4.2 Hypothesis about Rewrites . . . 82
4.3 Experimental Results . . . 83
4.4 Use Cases . . . 88
4.4.1 Data Logging . . . 88
4.4.2 Bit Vectors . . . 89
4.5 Comments on Overwriting . . . 90

Chapter 5: Flash Translation Layer for Serial NOR Flash . . 91
5.1 Serial NOR Dataflash . . . 91
5.2 The Flash Resident FTL . . . 100
5.2.1 A Fully Associative Mapping Strategy . . . 101
5.2.2 Read and Write Operations for FlaReFTL . . . 101
5.2.3 Architectural Overview . . . 105
5.2.4 Address Translation . . . 109
5.3 Consistency and Recovery with Zero-Overhead Logging . . . 120
5.3.1 Keystoning . . . 120
5.3.2 Record Modification . . . 122
5.3.3 Consistency and Recovery . . . 126
5.3.4 Operation Cost Summary . . . 129
5.3.5 In-Place Writes . . . 131
5.4 Frontier Advance Wear Levelling and Garbage Collection . . 132
5.5 Conclusions . . . 142

Chapter 6: Conclusion . . . 144
6.1 Conclusions and Future Work . . . 144
6.2 Summary of Contributions . . . 144

6.3 Future Work 145

Bibliography 147

Index 163

Appendix 165

Appendix A: FTL Algorithms 166
A.1 FTL Schemes 166
A.1.1 Block Based Logging Schemes 166
A.1.2 State Based FTLs 177
A.2 Comparison of FTLs 184
Appendix B: The FlaReFTL Interface 191
B.1 FTL Instances 191
B.1.1 System Control Functions 191
B.1.2 Page Management Functions 192
B.1.3 Read Functions 192
B.1.4 Write Functions 193
B.1.5 Buffer Management Functions 194

List of Tables

Table 2.1 A Comparison of Memory Stores for Embedded Systems. 32

Table 3.1 A Comparison of Flash Aware File Systems. 41

Table 4.1 Write State Truth Tables for a NOR Memory Cell. . . 81
Table 4.2 Least Squares Coefficients for Writing Techniques . . . 85
Table 4.3 Timing Comparison of Overwriting and Erase-Before-Write Operations 87
Table 4.4 Comparison between raw SD storage and serial NOR Dataflash (SNDF) using masked overwrite strategy. . . 89

Table 5.1 Load, Store and Data Transfer Costs for FlaReFTL Operations 129
Table 5.2 Comparison of Load and Store Costs for FlaReFTL Write Operations 132

Table A.1 Summary of FTL Schemes for Flash Memory 190

List of Figures

Figure 2.1 The Structure of a MOSFET 12
Figure 2.2 Conduction Channel Formation in a p-type MOSFET 12
Figure 2.3 Static RAM Memory Cell 14
Figure 2.4 Floating Gate MOSFET Configurations 15
Figure 2.5 Impact on Charge on a Floating Gate MOSFET . . . 16
Figure 2.6 Memory Cell Alignment for NOR and NAND Flash Structure 19
Figure 2.7 Read circuit for NOR-architecture flash memories . . 20
Figure 2.8 Read circuit for NAND-architecture flash memories . 21
Figure 2.9 Flash Memory Functional Block Diagram 25
Figure 2.10 Flash Page Organization 26
Figure 2.11 Architecture of a NAND Flash Memory 27
Figure 2.12 Dual Buffer Serial Dataflash 27

Figure 3.1 Rotating Magnetic Hard Disk Drive 35
Figure 3.2 A Log Structured File System 37
Figure 3.3 Sprite-LFS Space Management Strategies 38
Figure 3.4 Page Mapping Scheme for Flash Address Translation 44
Figure 3.5 Block Mapping Scheme for Flash Address Translation 47
Figure 3.6 Hybrid Mapping Scheme for Flash Address Translation 49
Figure 3.7 The Mistubishi FTL Scheme 53
Figure 3.8 The ANAND and FMAX FTL Scheme 56
Figure 3.9 The BAST FTL Scheme 59
Figure 3.10 The STAFF State Machine 64
Figure 3.11 PORCE Block Validation 66

Figure 4.1 Memory Cell alignment for NOR and NAND Flash . 79
Figure 4.2 Overwriting strategies for data movement from SRAM buffer to flash page for Serial NOR Dataflash 80
Figure 4.3 Memory Cell Write Transition States for Serial NOR Dataflash 81

Figure 4.4 A timing comparison of overwriting techniques for serial NOR Dataflash. 85
Figure 4.5 Distribution of Single Byte Write Times with Masked Overwritting. 86

Figure 5.1 Serial NOR Dataflash Data Read and Write Timing Sequences 93
Figure 5.2 Serial NOR Dataflash Write Operations 95
Figure 5.3 Serial NOR Dataflash Read Operations 96
Figure 5.4 FlaReFTL Data Movement During Read Operations . 102
Figure 5.5 FlaReFTL Data Movement During Write Operations 103
Figure 5.6 FlaReFTL Core Architecture 105
Figure 5.7 Memory Allocation Overview 106
Figure 5.8 The Management of Page State with Bit Vectors . . . 108
Figure 5.9 FlaReFTL Address Resolution Using Master Translation Table 111
Figure 5.10 FlaReFTL Address Resolution Using Secondary Translation Table 113
Figure 5.11 FlaReFTL Updating Master Translation Table 115
Figure 5.12 FlaReFTL Updating Secondary Translation Table . . 119
Figure 5.13 FlaReFTL Updating Master Translation Table 122
Figure 5.14 Updating Records in Data Page with MTTP 123
Figure 5.15 Updating Records in Data Page with STTP 124
Figure 5.16 Allocation of a Logical Page 125
Figure 5.17 Recovery in the Event of a Failure 127
Figure 5.18 Frontier Advance Wear Levelling and Garbage Collection Operations 135
Figure 5.19 Write Amplification Overhead During FAWL 137
Figure 5.20 System Resource Overhead with Respect to Write Frontier Extent Size 139
Figure 5.21 Results of the Wear Levelling Policy 142

Figure A.1 The FAST FTL Scheme 168
Figure A.2 The EAST FTL Scheme 172
Figure A.3 The STAFF FTL Scheme for In-Place Operations . . 179
Figure A.4 The STAFF FTL Scheme for Out-of-Place Operations 180

List of Symbols and Abbreviations

Abbreviation	Description	Definition
CHE	Channel Hot Electron	22
CS	Chip Select	92
EBW	Erase-Before-Write	33
EEPROM	Electrically Erasable Programmable Read Only Memory	18
FAFS	Frontier Advance Wear Leveling	132
FTL	Flash Translation Layer	34
FN	Fowler Nordheim	17
GC	Garbage Collection	42
LPN	Logical Page Number	46
LBN	Logical Block Number	46
MISO	Master In Slave Out	92
MLC	Multi-Level-Cell	24
MOSFET	Metal-Oxide Semiconductor Field Effect Transistor	13
MOSI	Master Out Slave In	92
OOB	Out Of Bounds	25
PBN	Physical Block Number	26
POR	Power Off Recovery	42
PPN	Physical Page Number	47
PPP	Partial-Page Programming	26
SCLK	Serial Clock	92
SLC	Single-Level-Cell	24
SOC	System on Chip	10
SPI	Serial Peripheral Interface	92
WL	Wear Leveling	29
WSN	Wireless Sensor Network	10

Acknowledgements

I would like to thank and acknowledge Dr. Ramon Lawrence for his ongoing motivation, support, vision and for the opportunity to study and learn under his supervision. The value of Dr. Lawrences ongoing guidance, feedback and friendship is immeasurable. His belief in you as a person only makes you want to excel to a higher level. Without his passion for research and belief in a vision, this work would not have been possible.

I would like to thank my wife Amy, and son Liam for their ongoing support, patience and perseverance. Most of all, I am forever grateful for their belief in me. I would also like to thank my family, friends and colleagues for being patient and the encouragement and support in my choosing to take this path.

I wish to extend my gratitude to committee members Dr. Yves Lucet, Dr. Jason Loeppky and Dr. Ramon Lawrence for the patience and guidance. I would also like to thank Dr. Craig Nichol for words of encouragement and support and gaining the understanding of done.

I would like to acknowledge the support of Dr. Ramon Lawrence, the Natural Sciences and Engineering Research Council of Canada (NSERC), and the University of British Columbia College of Graduate Studies for financial support which contributed to the success of this work.

Dedication

For Amy and Liam. Without your ongoing support, love, patience and understanding this journey would not have been possible.

Chapter 1

Introduction

> The more that you read, the more things you will know. The more that you learn, the more places you'll go.
>
> Dr. Seuss (1904-1991)

In today's world, embedded devices are playing an ever increasing and important role in daily life. An individual interacts with hundreds of devices in a single day, which all generate data. Recent interest has focused how this data can be harnessed and what underlying intelligence is contained within it. Whether it is simple environmental parameters such as the average temperature in our homes or complex data such as real time location and situational data [RR12], devices need to be able to store and process data. The emergence of the Internet of Things (IoT) has been gaining ground and momentum. This concept involves enabling common embedded devices such as wireless sensor networks and mobile computing platforms to be able to interact and intercommunicate with each other [GIMA10]. It is anticipated that by the end of this decade there will be between 30 and 50 billion devices participating in the IoT [Wit13]. One of the key aspects of the Internet of Things is the sensing and sharing of data and environmental parameters automatically in numerous domains [AIM10].

Due to the vast amount of data that is generated by devices, the ability to process data on device is beneficial as it reduces the amount of data that must be transferred off the device. IoT vendors such as Cisco anticipate the direct sharing of data between devices, driving the need for local storage and processing [Eva01]. Devices such as the Telos, Btnode, and MicaZ [BPC+07] platforms have been previously used as research and development platforms but the Arduino [Sev14] family of devices has driven low cost development and exploration. These devices are typically small 8-bit devices [ASSC02] with power, persistent storage and run-time memory constraints [DNH04] with less than 2 Kbytes of SRAM. While larger devices such as 32-bit ARM architectures are appearing in the embedded market space, 8-bit processors

are still dominant due to low cost, low pin count and complexity [Mur15]. While more complex processors are required in the realm of arithmetically complex operations, data collection and logging applications are still well suited to the 8-bit processor which is being driven by a growing number of IoT systems utilizing this technology [Mur15]. The 8-bit architecture is still the most commonly used device today accounting for almost 40% of all microcontroller device sales in 2014 [TBHR15]. In all cases, energy availability and secondary storage are key factors for consideration.

While rotating magnetic disk storage has prevailed for years in general purpose computing, it is unsuitable for use with embedded devices due to high power consumption and low mechanical robustness. Older solid state technologies exist, but until the advent of NAND and NOR flash memory, capacities were not large enough to be of any significant use for large volume data storage. Flash memory operates under fundamentally different principles than that of disk based storage. This presents significant challenges for accessing devices. The **F**lash **T**ranslation **L**ayer (*FTL*) functions as an interface to allow flash memory to be accessed in a similar fashion to disk based storage. While various schemes have been developed for flash memory, almost all of the work has focused on NAND flash. Unfortunately, NAND flash is not suitable for use with all embedded devices. NOR flash is still the most popular flash memory for use with embedded devices.

With the introduction of low-cost, micro-processing hardware and sensors, wide spread data collection is becoming more common for real-world use. In today's marketplace, embedded devices and wireless sensor networks require persistent storage due to the end user's desire for increased functionality and increased memory capacities balanced against its ever falling cost. There are a variety of reasons why data persistence is becoming an issue of critical concern. With the increased mobility of devices, the transmission of data is not always possible. Data must be cached for future dissemination in some form of transient, temporary storage. In other cases, data may be stored for backup and local processing, communicating only small subsets of data until further requests are made [Giu13]. Additionally, on-device processing is orders of magnitude less expensive that transmission [PK00].

With the different capacities and types of memories available on devices, one of the key challenges faced by users is how to handle the large amounts of data in an efficient fashion so that data persistence is achieved with the required degree of reliability and robustness. Traditionally, data has been stored in a raw format in some form of non-volatile memory. Data visualisation and analysis has not been possible on device due to how data is stored physically on the device, limited by both architectural and performance

constraints. For users to visualise and analyse data, sample points must be moved off the device. This may prove to be infeasible in cases where devices are connected as part of a wireless sensor network; large amounts of data being transmitted in a wireless sensor network can lead to premature failures of the network and loss of data. In general, it has been found that local processing of data can be orders of magnitude more efficient than transmitting data across the network. Unfortunately, challenges exist with local processing. This is primarily due to how data is stored on embedded devices.

Until recently, data has been stored in small scale non-volatile storage such as EEPROM as presented in Section 2.2.1, but this technology is limited to very small data storage, typically in the order of 1 to 100 Kb, and has significant power performance constraints. New sensors and hardware as well as increased demand for sensed data has increased the need for higher sampling rates and volumes. In response, new strategies for storing large amounts of data on small devices has been explored. These technologies utilize flash memory, which is a non-volatile storage medium that exhibits similar characteristics to other non-volatile file storage mediums but introduces significant challenges due to its physical structure and layout. Two architectures of flash have emerged which are categorized as NOR and NAND flash as presented in Section 2.2.2. While both share the same core architecture for the base memory cell, they differ significantly in the implementation with each having their own unique costs and benefits. Regardless of the type of flash memory, the technology offers some unique challenges that can limit the lifespan of the device as well as how a system interacts with it. The raw device is not a direct replacement strategy for current technologies such as EEPROM or rotating magnetic storage. As a result, new algorithms and strategies need to be developed in order to utilize and manage flash memory correctly such that the devices will have extended longevity and use in the field.

Traditionally, embedded systems have stored data in a raw or custom format where a single failure in the system can lead to an extensive loss of data. Unless considerable effort is put into development of the system, results will have uncertain data consistency and limited recoverability guarantees. This is due to the fact that the efforts are often time consuming and cost prohibitive. Additionally, it is often difficult to verify the correctness of the method, leaving developers with a system that appears to function correctly under certain conditions, but without absolute guarantees. Additionally, it requires the developer to have a high degree of technical understanding of the underlying technology in order to deal with the architectural constraints such as erase-before-write requirements, block level erasures and

wear considerations.

While NAND devices are commonly found in devices of all sizes and are well supported through extensive research on Flash Translation Layer *FTL* and wear levelling strategies, there are significant barriers for the use of this device on power and pin constrained embedded devices. Application assistance is also available through the use of flash specific file systems which are designed to work in conjunction with an operating system on a general purpose computing platform but are infeasible to use on smaller devices due to lack of operating system support, advanced memory management units and memory requirements. NAND devices also offer significant challenges due to their large pin count which causes issues for small pin count embedded processors. Additionally, NAND devices require the use of error correcting codes to manage bit level errors that commonly occur due to the design of the NAND device.

NOR devices are still the preferred choice for embedded systems [ZSI11] even through the capacities are lower that those offered by NAND devices. Low capacity devices can reasonably be used as a replacement for EEPROM devices as some devices offer byte addressability but require the implementation of a different software interface. Unlike NAND flash, the architecture of the device is more robust in terms of data correction and does not require the use of error correcting codes. This simplifies the overall implementation, but this feature does impact the overall performance of the device compared to NAND flash. NOR flash still has the same architectural limitations as NAND flash [MCO08] as well as having high pin counts. Serial NOR flash memories offer significant performance advantages over parallel NAND flash as they do not need error correcting codes. Serial NOR also has a smaller footprint and pin count than parallel NOR, making it more suitable for small embedded applications. While previous works have extensively examined the performance and optimisation of flash translation layers for NAND flash, none have examined the performance of these systems on serial NOR flash.

One significant limitation with all flash memory is the erase-before-write constraint. As a result of the physical limitation that a page must be set to the erase state before being written, data cannot be modified in place as it is done with rotating magnetic media. Systems utilizing flash memory as a drop in replacement for other memory stores are often architecturally limited from erasing the target data in-place. To accommodate this, systems will read the target data, modify and then rewrite the data to a free area of memory that has previously been erased. The challenge is to keep track of the physical addresses of the data as they are rewritten in such a fashion that the application is unaware of any changes. Research has focused on

using a Flash Translation Layer *FTL* (Section 3.4) to swizzle the physical address as data is moved and map it to a logical address that remains unchanged from the application's perspective. While this allows current systems that are designed to work with rotating magnetic media to seamlessly inter-operate with flash devices, it presents limitations in terms of efficient operation. Another common approach is to use a flash aware file system that incorporates the logical to physical translation directly into the file system as well as possibly exploiting other attributes of flash memory to gain an advantage over non-flash aware file systems utilizing an FTL.

Numerous open research problems exist in dealing with the architectural limitations of flash memory. One key area deals with the management and storage of the flash translations layer. Depending on the design of the FTL, if the complete table can fit in RAM, translations can occur very quickly as seen with page level mapping schemes (Section 3.3.1). Unfortunately, for most systems it is infeasible to store the complete table in RAM due to the ever increasing flash memory sizes. If the translation table can fit completely into memory, the issue of FTL persistence must still be addressed. If the system encounters a reset condition, the translation table would be lost and data would be orphaned. In order to become fault tolerant, the FTL must have some degree of persistence and recoverability. One strategy is to periodically flush the existing FTL to flash memory in order to guarantee durability but this can introduce a significant overhead in the system depending on the size of the translation table and as such is generally considered to be infeasible. As an alternate strategy, cache models for mapping tables have been investigated both at the page level where only part of the total mapping table is stored in RAM based on temporal and spatial frequency. Block level mapping (Section 3.3.1) utilizes the same strategies but with an increased level of granularity offering a trade-off between FTL flushing and addressable block size in memory as it generally requires pages within a block to be written to fixed addresses which can introduce challenges with non-uniforming writes. A third strategy is the hybrid page/block method (Section 3.3.1) that combines aspects of the page level FTL and block level FTL. It provides block level granularity for blocks of data but allows flexible data placement within the block, thus removing the issue associated with non-uniform writes.

A similar strategy to the hybrid scheme is the log-page or log-block scheme (Section 3.4.3). It is often categorized incorrectly as a hybrid FTL but is fundamentally different in terms of operation. While it maintains two different mapping granularities, its design is related more to a log based file system [RO92] than to a hybrid FTL and is intended to improve flexibility and performance under different write conditions. Data that is to be written

to the system is placed in a log structure in flash in a sequential fashion. Once a certain occupancy threshold is reached the log is flushed which allows logically aligned data elements that are currently in the log to be updated at the same time with target data that is currently in flash. The log is maintained as a small page mapped structure mapped limited size area that is reserved for frequently accessed data; as data in the log ages, it will eventually be moved out of the log to an area of flash that utilizes the block level mapping.

Regardless of the strategy used for managing access to flash memory, key issues exist. It is critical to ensure robust data persistence; that is having a storage solution that will ensure the consistency of data regardless of faults. Decisions as to the type of FTL, where to store the FTL and how frequently to commit the FTL are key open issues that dictate the suitability of a solution to a given problem. The primary goal of the majority of works is to reduce the number of erase operations, leading to higher write performance and indirectly, a longer service life. Other issues exist in terms of fault tolerance and recoverability, which most system do not address. Additionally, previous works have examined key issues with flash memory storage systems and proposed numerous solutions targeted for NAND flash. These solutions will not work with embedded devices that utilize serial NOR flash as their primary storage medium due to the architectural differences.

This work presents a novel flash translation layer for serial NOR flash that offers tunable consistency, a low memory footprint and a high degree of robustness allowing it to be suitable for use on constrained devices. The research has produced a fault tolerant flash translation layer for use with serial flash memories specially designed for resource constrained 8-bit systems. The work also introduces an overwriting strategy for NOR flash that can be used to reduce the cost of specific classes of writes as well as reducing the number of erase and garbage collection operations. The work simulates and tests on device an innovative FTL that provides good performance with a minimal host SRAM footprint as well as offering consistency guarantees. The system utilizes unique attributes of the Adesto (formerly Atmel [Atm04]) *Rapid-S* serial NOR Dataflash memory to provide a unique strategy for minimizing erase/write costs.

The key contributions of this work are a serial NOR Dataflash FTL with the features:

- Minimal SRAM memory footprint flash translation layer for NOR Dataflash
- A fault tolerant and robust flash translation layer

– A consistent and recoverable data management system

– A deterministic, low overhead garbage collection mechanism

– A low overhead wear levelling algorithm

– Efficient buffer management through the use of direct reads

– A general write-in-place writes for energy and device conservation.

Key basic components of FlaReFTL have been examined under simulation in [FL11] as the basis for a file system for embedded devices. This work expands and completes the migration to a stand alone system for use in memory constrained devices.

This thesis examines the current requirements for data persistence solutions in embedded devices and the performance and architectural differences. Chapter 2 presents a background on storage technologies, concentrating on NOR and NAND flash. Chapter 3 examines current data persistence strategies at both the file system and flash translation layer through the development of log based storage. Chapter 3 concludes with a summary of the limitations with current data persistence strategies specifically in regards to their suitability with resource constrained devices as well as highlighting open research questions. Write strategies for serial NOR Dataflash are presented in Chapter 4, which examines overwriting techniques to improve write performance and reduce erasures for specific write patterns. Chapter 5 presents a robust and consistent FTL designed for resources constrained devices with a minimal SRAM footprint. The FTL exploits overwriting techniques to reduce specific overhead operations. The work closes with Chapter 6, which summarizes the contributions of this work and concludes with a discussion of future research directions.

Chapter 2

Storage Architectures

> Give them the third best to go on with; the second best comes too late, the best never comes.
>
> — Robert Alexander Watson-Watt (1892 - 1973)

With the advancement and pervasiveness of mobile embedded devices in our daily lives and environment, we have become more interested in harnessing the power of these devices to capture and understand information about the surrounding environment. Whether it is simple environmental parameters such as the average hourly temperature in our homes or complex data such as real time location and situational data [RR12], devices need to be able to store and process data. Mobile embedded devices with high resolution sensors and the desire to share the data, have rapidly driven the need for larger capacity data stores. Additionally, a new paradigm called the Internet of Things (IOT) has been gaining momentum in research communities. This concept involves enabling common embedded devices such as wireless sensor networks and mobile computing platforms (such as smart phones) to be able to interact and intercommunicate with each other [GIMA10]. One of the key aspects of the Internet of Things is the sensing and sharing of data and environmental parameters automatically in numerous domains [AIM10]. Understanding what can be done with data only drives forward the need for improved data persistence strategies.

In terms of storage options, early adopted solid state technologies such as EEPROM, while suitable for small sets (both in terms of size and data rates), are limited in terms of capacity, speed, and power required. As a result, EEPROM's suitability as a data persistence device in today's environment has fallen due to our ever increasing need for capacity and device lifetime (through maximum power management). While suitable in terms of capacity, rotating magnetic media fails to offer suitable performance in terms of power consumption and physical robustness.

The emergence of solid state flash memory from Toshiba in the 1980's [PH12, p.6.14-4], presents new opportunities for improved data persistence strategies due to significantly larger storage capacity compared to EEPROM, long term storage stability, acceptable bandwidth and latency (particularly when matched with data I/O speeds of processors) as well as suitable power draw requirements. The evolution of flash occurred in two phases [PH12, p.6.14-4]. The first device to enter the market was NOR flash in 1984 with NAND flash following in 1989. The first commercial use of NOR and NAND flash was in digital cameras [PH12, p. 6.14-4]. NOR flash was adopted quickly in embedded applications due to the fact that some devices had the ability to be written at the byte level (with the precondition that the memory was already erased) and hence became a suitable replacement for EEPROM memory [SCKS08]. It maintained market dominance over NAND flash until 2005 by measure of market revenue [AA11]. The adoption of NAND flash did not move at the same rate due to the significantly more complex write patterns due to the block structure of the device [SCKS08]. While NAND flash has found numerous applications such as mobile handsets and primary storage solutions, NOR flash is still the primary choice for use in embedded applications [ZSI11] primary due to its low read latencies and data integrity.

2.1 Embedded Devices

Embedded devices are pervasive due to their low cost, low power requirements and small size, both in terms of physical footprint and memory. This makes them desirable for numerous applications where the end cost is a critical parameter. A microcontroller forms the computing core of an embedded device with the core, typically being an 8 or 16-bit device. A microcontroller differs significantly from a microprocessor, which forms the foundation of general purpose computing. A microprocessor contains only a processing core and lacks memory, peripheral devices, and timing circuity. A microcontroller contains all components on a single die; it only requires software to be added to make the device functional [Hea02, p.11].

A recent trend has seen the growth of microprocessor based embedded systems [Hea02] specifically with the introduction of the low power ARM architecture [PH12, p.2.20-4]. Patterson and Hennessy [PH12, p.2.20-4] note that one of the first embedded computers to use the ARM processor was the Apple Newton, a revolutionary personal digital assistant. While the device failed in the market place, Patterson and Hennessy note that because of

Apple's choice of processor, subsequent vendors were confident in making a similar choice. The ARM core is now found in numerous commercial embedded devices but most notably in communications devices. In 2008, it was estimated that over 3 billion ARM cores shipped [PH12]. Building on the ARM core which lacks peripheral devices, core memory and a timing system for embedded applications, designers are now producing systems-on-chip (SOC) which allow the combination of a microprocessor core and other devices fabricated onto a single silicon substrate and package. While suitable for applications where physical size, cost and power consumption play less of a design role, embedded microprocessors have yet to displace the 8/16-bit microprocessor for low cost and power limited resources.

In addition to the growth of wireless consumer devices, there has been significant interest in using low power and low cost devices to form an Internet of Things [GIMA10, AIM10] where embedded devices can interconnect, interact and share data. One of the key components in this paradigm is the wireless sensor network [GIMA10] which is the key backbone for data collection and sharing.

2.1.1 Wireless Sensor Networks

Wireless sensor networks are an efficient way to gather sensed data from a large physical area without the need for hard wired infrastructure [ASSC02]. A wireless sensor network consists of a series of wireless sensing devices that have the ability to measure parameters regarding their physical environment. Devices are typically small 8-bit devices [ASSC02] that are constrained in terms of power, persistent storage and run time memory [DNH04]. Numerous hardware platform choices are available with the majority of research being conducted on the Telos, Btnode and MicaZ platforms [BPC+07]. Devices intercommunicate using a wireless link [ASSC02]. Although there are many different communication paradigms, IEEE 802.15.4 [BPC+07, IEE07] has emerged as the dominant choice. It allows for low data rate, and low power communications between devices. These devices have been extensively used for data collection in military, environmental, agricultural and industrial applications [ASSC02, MCP+02, BTB04, SOP+04, CCS+07, WCS+07, MFM+08, PE08, BCDV09, GSS09, LSS+09, Riq09, MGZ+09, RGA+09, GPSZ10, FL10, JRO+11, RUJ+11, FCTL12]. In these applications, it is critical to be able to store data locally on device to prevent data loss due to the inconsistent nature of wireless networks. While they collect vast amounts of information, only a small subset is of interest and usually pertains to a specific transitory or periodic event in the sampling space. Recent

work suggests that transmitting data over the wireless link is the largest component of a device's energy budget and that the lifetime of a network can be increased substantially by minimizing the amount of data being transferred [RSPS02, CL10]. Mather et al. [MDGS06] suggest that locally processing of data utilizing NAND flash is two orders of magnitude more efficient in terms of energy usage. In a study of transmission costs, Pottie and Kaiser [PK00] demonstrate that the amount of energy to transmit 1 kilobyte over a link of 100 meters is equivalent to a processor executing 3 million instructions at 100 MIPS/Watt. As a result, research is focusing on developing improved data handling mechanisms in an effort to reduce network load [PK00, ASSC02, CL10, HSW+00, FL11].

The management of data can be handled by a custom application [FL10] or by one of the over 37 different operating systems available for wireless sensing devices [DTV09]. In the existing systems, local data storage has not been a focus. Sensed results are pushed unprocessed back to the sink or a collection point. With the increased availability of low cost flash memory ($0.003 per kilobyte), it is now possible for nodes to maintain results locally. Recent work has investigated how to efficiently store data in flash [GT05b]. To augment the existing operating systems, other work has investigated using flash specific operating systems to abstract the physical storage away from the application while taking advantage of the architecture of the flash memory [DNH04, GT05a, MDC+09].

2.2 Memory Stores

The operating nature of embedded systems is inconsistent due to power faults, programming errors or other factors. Devices require persistent memory stores to withstand these events. Rotating magnetic storage [PH12, p.22] is not suitable for embedded systems due to numerous factors such as physical size, robustness, interface complexity, bandwidth, and power consumption [DZ11][IFV11, p.219]. As a result, embedded designers have turned their attention to alternative re-writable, non-volatile memory technologies.

One of the earliest options used for embedded systems was the battery backed up SRAM which is based on the SRAM memory cell.

Definition 2.1. The *memory cell* is the basic storage element and stores 1-bit of information.

Definition 2.2. *Programming* is the act of writing data to or changing the state of a memory cell.

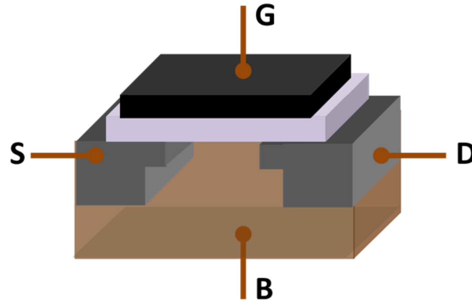
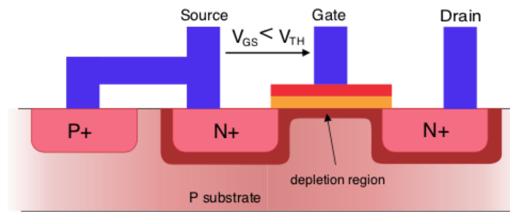
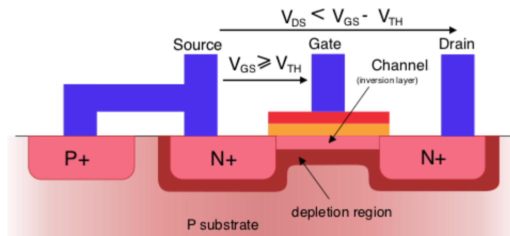


Figure 2.1: The Structure of a MOSFET. Reproduced from MOSFET Structure by Brews Ohare, available under the Creative Commons Attribution-Share Alike 3.0 Unported at `https://commons.wikimedia.org/wiki/File:MOSFET_Structure.png`



(a) p-type Enhancement-type MOSFET



(b) Channel Formation for p-type MOSFET

Figure 2.2: Conduction Channel Formation in a p-type MOSFET. Reproduced from MOSFET Functioning Body by Sketerpot (derivative work), from original work by Olivier Deleage and Peter Scott, available under the Creative Commons Attribution-Share Alike 3.0 Unported at `https://commons.wikimedia.org/wiki/File:MOSFET_functioning.svg`

SRAM is based on the metal-oxide semiconductor field effect transistor and is used in almost every modern computing device utilizing CMOS technology. The MOSFET is a power efficient transistor device that can allow current to flow through the device based on a control signal. While available in two different formats, memory cells are constructed using an enhancement-type MOSFET, which when not powered will have a very high resistance and thus not conduct current when in the off state [BN92, p.231]. Figure 2.1 shows the typical construction of a p-type enhancement-type MOSFET. The device is constructed of a p-type semiconductor substrate with two n-doped regions that form the drain (D) which is the entry point for conventional current and the source (S) which is the exit point for conventional current. Over top of the p-type substrate and the source and drain is placed a high dielectric insulating layer on which the control gate (CG) is placed. When no voltage is present on the gate relative to the source (Figure 2.2a), the p-type substrate between the source and drain is impervious to current flow. Once a bias voltage is placed on the control gate relative to the voltage at the source gate (Figure 2.2b), an electric field is created between the p-type substrate and the control gate. The presence of the electric field creates a depletion zone beneath the control gate and insulating layer in the p-type substrate. The size of the depletion zone is dependent on the voltage potential present at the control gate and once the voltage reaches a prescribed threshold, the depletion zone will form a conduction channel between the source and drain. While acting like an electronically controlled switch, the MOSFET is essentially an analogue device such that the voltage placed on the control gate will regulate the flow of current between the source and drain of the device. While there is a minimum threshold voltage required to create the conduction channel, once it has been established applying a greater voltage will increase the size of the channel and allow more current to flow. In SRAM, a flip-flop memory cell is constructed with a group of MOSFETs such that a bit of data can be stored in the structure (Figure 2.2). The memory cell requires two control lines in order to read or write data, typically presented in a two dimensional matrix (row and column) format. The word (or address) (WL) line is used to couple a given cell to a bit line (BL) which is used to sense the state of the cell. This is accomplished by connecting the word line to the control gate of the output MOSFETs and the bit line to the source of the same MOSFET. If a logic '1' is present in the cell, current will flow to the bit line, inducing a voltage that can then be read [Sta13, pp. 162-163]. While a power efficient structure, the major drawback to the structure is that power is constantly required for data persistence. As soon as power is lost, the data stored in

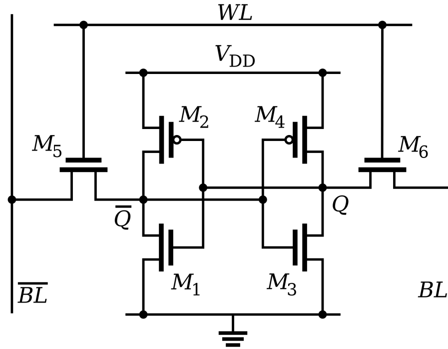


Figure 2.3: Static RAM Memory Cell

the system will be lost.

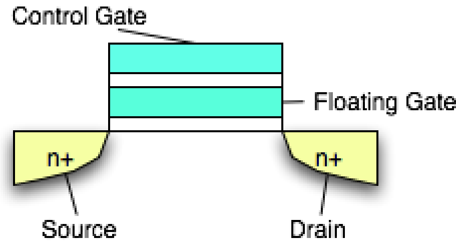
To address the issue of data persistence for systems that require long term stability with uncertain power, SRAM can be continuously powered by either an external or internal battery enabling the state to be retained even while the host processor is off. This technology is still in use today in computer BIOS and is available as a discrete memory component for use with embedded systems and is refereed to as battery backed-up SRAM or NVSRAM (non-volatile SRAM). While interactions with battery backed-up SRAM are straightforward and transparent, numerous drawbacks exist with the technology, most notable being the fact that if the battery were to fail, data would be lost. Additional challenges limit its implementation in embedded systems due to the high cost compared to other technologies.

While generally suitable for applications where power conservation or size was not an issue, battery backed-up SRAM is not suitable for most embedded computing applications. As a result, research investigated utilizing a class of memory called NVRAM, enabled by a technology called the floating gate MOSFET which has a similar architecture to the MOSFET used in SRAM. Unlike the MOSFET, the floating gate MOSFET can encode a persistent state for a long period of time without the requirement that it be continually powered.

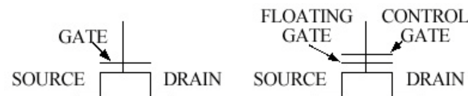
Definition 2.3. *NVRAM* is a memory device that exhibits read and write characteristics similar to other RAM devices, but maintains a degree of persistence with respect to the power state of the device. If the power is removed from an NVRAM device, the data that is stored in the device will not be immediately lost as it is with SRAM.

While a broad category, NVRAM generally includes electronically erasable

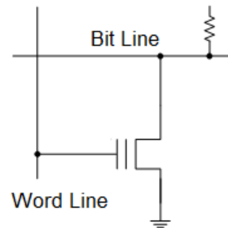
and re-writable devices also known as electrically erasable and programmable read only memory (*EEPROM*). While a misnomer, as the devices can be erased and re-written like RAM, the name has persisted; Stalling points out that historically the memories have been mostly read-only [Sta13, p.164].



(a) Configuration of the Floating Gate MOSFET



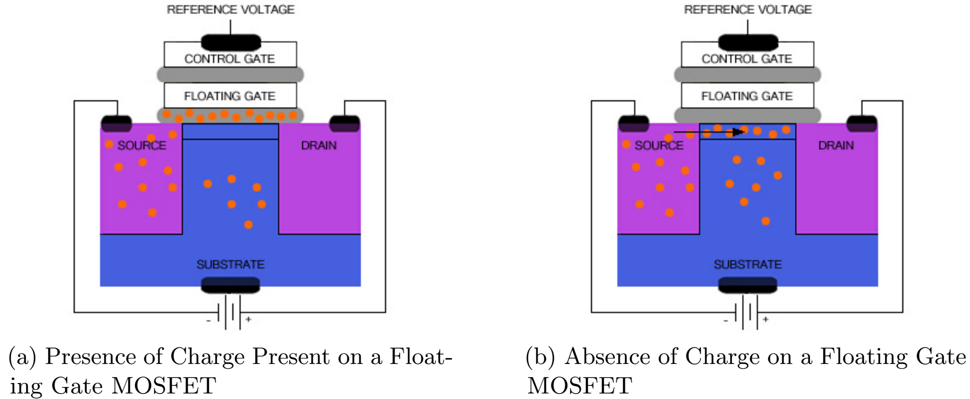
(b) Comparison of the MOSFET and Floating Gate MOSFET Schematic



(c) Floating Gate MOSFET Memory Cell

Figure 2.4: Floating Gate MOSFET Configurations

The floating gate MOSFET was first proposed by Kahng and Sze [KS67] as a device suitable to form a memory cell in 1967. The device is very similar in construction to the MOSFET but introduces a fourth gate into the architecture called the Floating Gate *(FG)* (Figure 2.4a). The floating gate is located between the control gate and the channel substrate. It is electrically isolated from all other parts of the circuit and acts as a barrier between the control gate and the channel substrate. The floating gate MOSFET is schematically different from the MOSFET with an additional line being



(a) Presence of Charge Present on a Floating Gate MOSFET

(b) Absence of Charge on a Floating Gate MOSFET

Figure 2.5: Impact on Charge on a Floating Gate MOSFET

added to the image between the control gate and substrate (Figure 2.4b). Due to the property of the floating gate being electrically isolated from other components of the device, it has the unique ability to hold charge for long periods of time (years to 10's of years) [BCMV03] but will overtime dissipate charge and loose the data encoded in the device [SCKS08].

When a charge is present on the floating gate, it will block the electric field from the control gate which is normally capacitively coupled to the channel substrate preventing the formation of the conduction channel [BCMV03]. Unfortunately, it is not a binary effect; the charge level present on the floating gate modifies the threshold voltage needed to be induced on the control gate in order to form a conduction channel [BCMV03]. This characteristic is used to form the basic element of NVRAM memory cell. Consider the circuit in Figure 2.4c. If charge is present in the float gate (Figure 2.5a) when the word line is activated to read from the cell, the bit line will not see a current flow which is interpreted as a logic '0' [BCMV03, CMMS08]. This is due to the fact that the electric field from the control gate to the channel substrate is being shielded by the floating gate. If no charge is present when a voltage is induced at the control gate (Figure 2.5b), a conduction channel will be formed allowing for current to flow which is interpreted as a logic '1'[BCMV03, CMMS08]. While the readability of the device is similar to other memories, technical challenges exist with writes and erases on the device as charges need to be introduced or removed from the electrically isolated floating gate [BCMV03, CMMS08, MMR08a, MMR08b] which is the fundamental difference between the non-volatile technologies utilizing

floating gate MOSFETs and other technologies. Additionally, different configurations of the floating gate MOSFET utilize different sensing methods for determining current flow. In Section 2.2.1 and 2.2.2 the different methods will be introduced as they relate to the specific architecture.

With the floating gate MOSFET architecture, erasure is accomplished using *Fowler Nordheim* (FN) tunnelling [BCMV03, CMMS08] which involves applying a strong electric field between the source and control gate. This generates a quantum mechanical tunnel through the oxide insulation layer through which the electrons trapped in the floating gate can be removed [BCMV03]. As Bez et al. [BCMV03] note, the advantage of this method is that large tunnelling currents can be applied without destroying the dielectric properties of the insulator. The induced voltage difference needs to be on the order of 18 V [CMMS08] which can not be typically generated from the low supply voltages normally used with memory devices. The voltage levels are generated with on-chip charge pumps, which represents a significant source of power consumption in the overall operation of any floating gate MOSFET device.

One significant limitation with all floating gate MOSFET devices is the accumulation of charge over time in the electrically isolated floating gate. With continual erase cycles over time, charge builds up on the floating gate affecting the forward threshold voltage required at the gate, which impacts the devices ability to conduct current. As a result, the device will eventually become stuck in a non-conducting state.

2.2.1 EEPROM

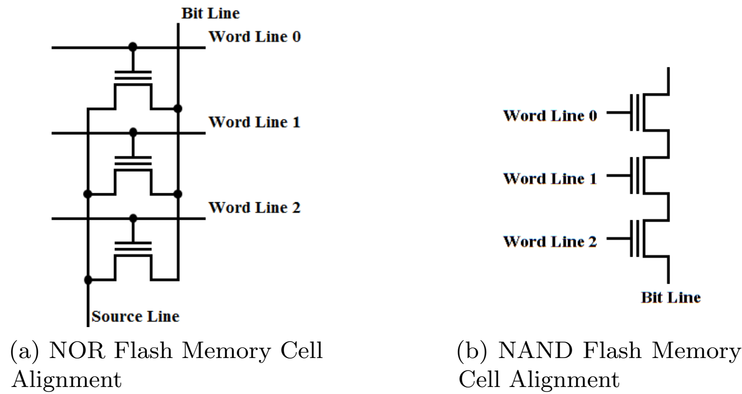
The first true non-volatile readable and programmable memory was developed by George Perlegos at Intel in 1978 [Ros02]. Unlike battery backed memories, this technology required no internal or external battery to maintain state and was called an *Electrically Erasable Programmable ROM* and was based on the floating gate MOSFET. While it still had limitations in terms of capacity due to the architecture and high voltage requirements for erasing [Ros02], it was a significant step in data persistence technologies. A microprocessor could have a small attached non-volatile storage that had no mechanical parts, low power consumption compared to previous technologies and eliminated the risk of accidental battery failure leading to unwanted data loss. This technology became so pervasive that it ushered in a new class of microprocessors during the early 1990's that allowed quick in-circuit reprogramming. This was a significant step forward as up until this point processor memories required erasure through exposure to ultraviolet radiation

which required external hardware [DB07] and took a lengthy period of time. Two distinct types of electrically erasable and programmable ROM's are available [DB07]. The first is *EEPROM* which groups memory cells into blocks for erasures. The second is E^2PROM, which is byte erasable. With both types of memories, writes must be preceded with an erase operation leading to a two step process to update a value in memory. The target location must first be erased before it can be written which is common to all floating gate MOSFET based memories. EEPROM uses FN tunnelling for both the reading and writing process [BCMV03].

2.2.2 Flash

Flash memory is a type of electronically erasable and reprogrammable read only memory *EEPROM* [DB07] that was invented by Dr. Fuji Masuoka in 1980 [KRKC11]. It is considered to be a low power consumer compared to other EEPROM technologies and is shock resistant, small in footprint and has a data persistence period greater than 10 years [Atm04, Inc08]. It offers increased capacity and speed in addition to lower energy usage [MDGS06, ZYLK+05] when compared to other types of solid state devices making it particularly attractive for energy constrained devices such as wireless sensor nodes and embedded microprocessor systems. Similar to EEPROM technologies, it uses the floating gate MOSFET as the fundamental element for a memory cell.

Flash is available in two distinct types: *NOR* and *NAND* configurations. While both share common characteristics, the physical implementation of each type is very different and offers different performance benefits. NOR flash was first commercialized by Intel in 1988 [KRKC11] and initially offered read and write units of one byte. The cell size for NOR flash is 2.5x larger than NAND [SCKS08] due to additional circuitry required for reading the state of cells. It is considered to be suitable and well designed for applications that require random access to data but due to its larger cell size, NOR was not initially adopted as a suitable candidate for read/write storage [DZ11]. The evolution of NOR flash is now increasing its suitability as a storage candidate. It is now commonly available in page oriented fashion similar to NAND flash [Ade15, Atm04, Inc08]. NOR flash is the most popular format for embedded devices [ZSI11], being found in millions of embedded devices. NOR flash is typically less dense and less energy efficient but is available in either a parallel or serial access format [MDGS06]. The majority of research with NOR flash has focused on parallel versions which have traditionally been used for code images and not run-time data storage. It offers other



(a) NOR Flash Memory Cell Alignment

(b) NAND Flash Memory Cell Alignment

Figure 2.6: Memory Cell Alignment for NOR and NAND Flash Structure

advantages in terms of device organization as it may be byte, word or page orientated.

NAND flash is the most commonly found format [ZBT09], and is characterized by fast access for sequential byte access and high density and durability. However, it exhibits high latency in start up and can be prone to bit errors due to the physical cell interconnects and architecture leading to increased operational complexity. Data is only accessible in a page format [MDGS06, ZYLK+05] which limits the types of read and write actions. NAND flash is typically accessed in a parallel fashion, requiring a high pin count commitment from the host processor which makes it unsuitable for small, low pin count devices.

At the most basic level, flash memory consists of memory units called *cells* that will encode a bit or bits of information depending on the fabrication technique used. Due to the electronic design of flash memory, an erase cell will have the state of a logic '1'. When programmed, the cell value will be set to a logic '0' [MCO08, p.29]. The cells are connected together in matrix using the wording line for addressing and the bit line for sensing, allowing the encoding of data as a byte. The most significant difference between the architecture of NOR and NAND flash is how the state of a cell is determined [BCMV03][MCO08, p.30] as shown in Figure 2.6. With the NOR architecture (Figure 2.6a) each element in the matrix has its control gate connected to the word line and the bit line connected to the drain [BCMV03, MMR08b]. This allows the matrix to address a single element in the memory array without disturbing any other element. NAND flash shares a similar configuration in terms of the word line which is used to

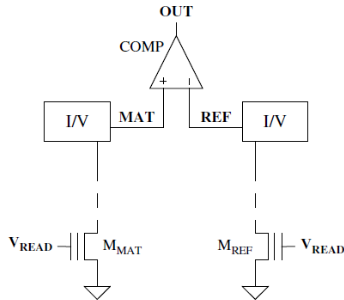


Figure 2.7: Block scheme of a circuit used to compare two currents, used for reading for NOR flash. Reproduced from Memories in Wireless Systems, Nonvolatile Memories: NOR vs. NAND Architectures, 2008, p. 31, Crippa, L., Micheloni, R., Motta, I. and Sangalli, ©Springer-Verlag Berlin Heidelberg 2008 "With permission of Springer"

activate the control gate of the element or elements being read. The single largest difference in the architecture difference between NAND and NOR is how the bit line is connected. Unlike in NOR memory, the source and drains of the memory cells in NAND flash are linked together (Figure 2.6b) in a daisy chain fashion where the source of one gate is connected to the drain of the next [BCMV03, MMR08a].

The method by which the state of a NOR cell is determined is very different from the method by which the state of a NAND cell is determined and leads to a significant size difference between the two architectures. The NOR architecture uses a voltage and current comparator to measure the state of a cell (Figure 2.7) were M_{MAT} is the matrix cell being read and M_{REF} is a reference cell used for comparison. To read a cell, the corresponding word line for the cell being testing is activated. If there is no charge present on the floating gate, a current in the order of microamps will flow [BCMV03]. The output from the bit line is passed through a current to voltage converter [CMMS08] which converts the current flow into a corresponding voltage. This is done to simplify measurement techniques and results in a system that can use a well understood voltage comparator circuit known as a sense amplifier [MMR08b]. The value is measured against the output from the reference cell which will compare the two voltage levels using a differential comparison method and generate a voltage that will correspond to the logic level stored in the cell. On modern NOR flash memories, this technique is robust and fast with read times in the order of

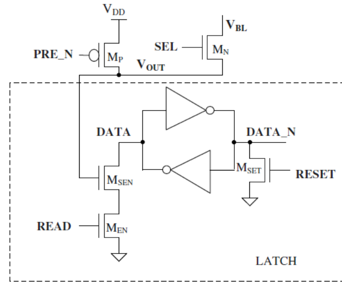


Figure 2.8: Read circuit for NAND-architecture flash memories. Reproduced from Memories in Wireless Systems, Nonvolatile Memories: NOR vs. NAND Architectures, 2008, p. 37, Crippa, L., Micheloni, R., Motta, I. and Sangalli, ©Springer-Verlag Berlin Heidelberg 2008 “With permission of Springer”

10 to 20 nanoseconds [CMMS08, p.33]

In the configuration of the NAND architecture where cells are connected in series [CMMS08] unlike NOR flash were one specific cell can be targeted. The value of the cell being read must be disambiguated from the remaining cells in the chain. In addition to correctly selecting the word line for a cell, each group of cells contain a switching device that will connect the group to the bit line being tested. The remaining cells not being read need to be biased so a sense current can flow through the target cell (Figure 2.8). This is accomplished by driving the word lines on the remaining cells in the group past the conduction threshold voltage such that each cell will conduct regardless of the charge level on the floating gate, thus effectively rendering the devices as pass through transistors [CMMS08, p.34][MMR08a, p.86].

A read in NAND flash is accomplished through the measurement of parasitic capacitance [MMR08a, p.87] and charge integration [MMR08a, p.89], unlike NOR flash were current is directly measured. In addition to biasing the other gates in the chain correctly, the bit line is pre-charged before the read which represents a very power intensive operation [MMR08a, p.87]. In order to evaluate the state of the target cell, the charged bit line is allowed to dissipate through the string of over biased cells to the cell under evaluation during what is called the *evaluation phase*. If there is a charge on the floating gate of the target cell, the charge on the bit line will remain constant whereas if there is no charge on the floating gate, the cell will absorb charge causing the bit line charge to decrease which will impact the bit line voltage with respect to time. After a fixed amount of time, the bit line voltage is measured and used to determine the state of the target

cell [MMR08a, pp.87-89][CMMS08, pp. 35-37]. This technique is more prone to errors and spurious effects as the current encountered with reads in NAND flash are very small (typically 200 nanoamps) [CMMS08, p.36]. Due to the very small currents used for read operations, NAND flash cannot use the differential comparator used in NOR flash. Once the evaluation phase has been completed, the voltage level is latched in volatile memory structure (Figure 4.1b), which can then be read out.

Both methods are not without there positive and negative attributes. While the NAND flash read configuration results in a smaller physical footprint, it offers some draw backs in terms of timing. In an effort to be cognizant with respect to peak current capacity, the bit line pre-charge is typically 2 to 6 microseconds [CMMS08, p.35] with the evaluation phase typically being between 5 and 10 microseconds [CMMS08, p.36].

While based on the same core memory cell, NAND and NOR flash are programmed using different techniques [MMR08a] that accounts for a fundamental difference in operation. With NAND flash, programming is accomplished using Fowler-Nordheim tunnelling which is a quantum-effect electron tunnel generated in the presence of a strong electric field [MMR08a]. A channel tunnel is created when an electric field is generated between the substrate and control gate to overcome the potential barrier of the insulating oxide. This allows electrons to pass into the floating gate [CMMS08, MMR08a]. The level of charge that can be induced onto the floating gate is proportional to the strength of the electric field generated. As a result, it is necessary to have a high voltage source to generate the strong electric field which improves programming performance [CMMS08] but presents a challenge in terms of available power. The shortcomings with the method is that the programming time is considerably longer than compared to the method used with NOR flash [Bri97, CMMS08] but overall realizes higher number of programmed bits per second due to the parallel architecture utilized. NAND flash memory will also suffer from high voltage induced tunnel oxide degradation which impacts the overall endurance of the device [CMMS08]. On the plus side, the technique is desirable when a large number of cells are required to be programmed as it uses very low currents (on the order of nanoamps) [CMMS08].

Unlike NAND flash, NOR flash is programmed using *Channel Hot Electron* (CHE) injection, which involves the generation of an electric field by a differential voltage between the source and drain [CMMS08]. The electric field provides enough energy to electrons through a collision mechanism [CMMS08] such that electrons can freely pass through the oxide layer [BCMV03]. A traversal electric field is then generated by the application of a voltage to the

control gate which allows electrons to migrate to the floating gate [CMMS08]. One advantage of the channel hot injection programming method is that it has a natural termination point due to the fact that as charge accumulates in the floating gate impacting its electric potential; additional charge will be less attracted and eventually programming will stop [CMMS08]. Compared to the per cell programming speed of NAND flash, channel hot injection is much faster [Bri97, CMMS08], but requires a much larger current. This can pose a limitation to system resources especially when programming a large number of cells in parallel. Design restrictions may limit the total current available and thereby limit the total number of cells that can be programmed in parallel.

While the programming methods for NAND and NOR flash are different, both architectures employ a technique called program and verify. It is used to verify that the correct charge has been induced onto the floating gate thus creating a specific threshold voltage such that a cell can be considered programmed [CMMS08]. If the target cells do not reach the correct threshold voltages, the cells will be successively reprogrammed until the voltages are correct. If the correct threshold voltage cannot be reached in a given number of attempts, the system will generate an error that the programming operation has failed.

Both NAND and NOR flash are erased in the same fashion. Due to the architectural interconnections of memory cells, they must be erased in blocks unlike other floating gate MOSFET memory technologies. This was primarily done to conserve space and lower production cost. The erase operation is performed using Fowler-Nordheim tunnelling [BCMV03, MMR08a, MMR08b] where a high voltage is placed across the oxide layer. This is accomplished by placing a high voltage erase pulse across the entire block [CMMS08]. This allows the formation of a Fowler-Nordheim tunnel between the gate and source allowing electrons to be removed from the floating gate. The removal of the charge buildup in the floating gate now allows for channel formation when the control gate is activated. One of the fundamental limitations of flash memory arises from the erase operations. Over time, the tunnel oxide will break down [CBCF94] leaving residual charge on the floating gate which will eventually leave the memory cell locked in one state. Another complication that arises from the charge build up is that as the device wears the erase time per cell increases leading to a decrease in performance over the lifetime of the device. In practise, memory devices utilize complex erase algorithms to minimize this effect [BCMV03, CMMS08]. Both NAND and NOR memories utilize an erase verification algorithm that is designed to ensure that block has been erased correctly, but due to

the architectural differences between NAND and NOR memories the erase algorithm involves different operations [CMMS08]. This leads to a significant difference in erase times between NAND and NOR memories with NOR flash erase times being three orders of magnitude greater that with NAND memories [CMMS08, DZ11] due to the increased complexity and steps of the erase algorithm. This erase time difference can limit the class of applications a specific memory may be used for [DZ11].

With flash memory there are fundamental challenges that limit the operation of the device and are due to the methods used for programming and erasure [Bri97, p.10-12]. Oxide stress and over erasing and programming can affect long term data retention. Over time, charge starts to build up in the oxide layer of the cell which causes both the required erase and program voltages to increase with the number of times the device is cycled [CMMS08]. This is known as tunnel oxide degradation [BCMV03, CCS+07]. In NOR flash, the oxide degradation creates a "erase threshold window closer" [BCMV03] where the voltage window between the erased and program states narrows. In practise, the internal controller accommodates for this which results in increasing programming and erase times as the device ages.

In an effort to increase the storage capacity per unit area of flash memory cells without having to shrink the size of the memory cell, a single cell can be used to encode more than one state of data. A cell that encodes only one bit of data is called a Single-Level-Cell *(SLC)* whereas a cell that stores more than one bit of information is called a Multi-Level-Cell *(MLC)* [DZ11]. A MLC increases the density of a cell by varying the charge level on the floating gate to represent different bit states. This significantly increases the complexity of the read circuitry as the system must now not only determine the presence or absence of a current flow but now must interpret what the value is to determine the value encoded by the cell [BCMV03]. With this technique, a cell can now encode up to 8 different states allowing for 3-bits of data to be stored [Lev08]. This technique allows MLC flash to have a lower cost per bit and higher density [Lev08, DZ11] but this is not without cost. MLC devices have considerable shorter lifespans due to oxide degradation [Lai08] as well as increased read times. Read and write times for SLC NAND flash can be on the order of three times faster than with MLC NAND flash as well as having a considerably lower bit error rate [JKJ+10]. As a result, SLC technologies have a longer service life, are more robust and faster, but have a higher cost per density when compared to MLC devices.

Erase operations are slower than other operations on the device and must occur in physical blocks or sectors, which can pose a memory management problem for small microprocessors. Each page in the device has a limited

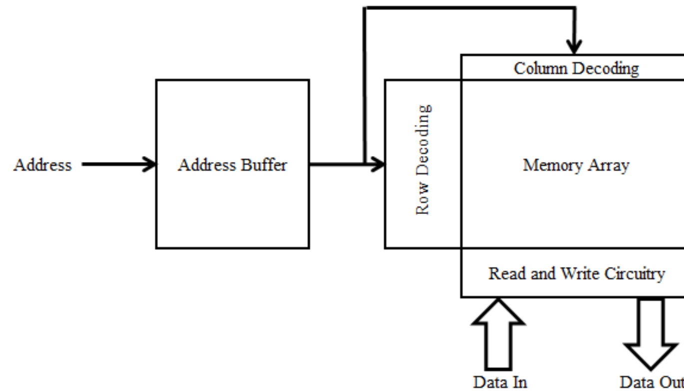


Figure 2.9: Flash Memory Functional Block Diagram [Des10]

number of erase cycles (typically 100,000) before the device starts to fail. A wear levelling algorithm [KNM95] can be used to amortize the cost of the write/erase cycle across all pages to extend the effective life of the device. Other devices maintain adjacent block erase/write restrictions which limits the difference in erasures (from a time perspective) forcing the need for periodic consecutive block erases [Atm04].

Flash memory groups cells into minimum read/write units called pages. Figure 2.9 shows the general organization layout of flash memory. Confusion is commonly generated by authors also referring to the minimum unit of readable or writable data as a *sector* due to the fact that some manufacturers use the term *sector* differently. Additional confusion is encountered as file systems will also use sector to refer to the minimum unit of transmutable data on rotating magnetic disk [Sta13]. In this discussion, the term *page* will be used explicitly to refer to the minimum transmutable unit of data in flash memory.

Definition 2.4. A *page* is a physical sequence of bytes in memory that share a common page address and is the smallest unit of data that can be written or read in the flash architecture. [CPP+09].

In addition to the main memory area of a page, which utilizes a 2^n base addressing scheme, a page contains limited additional memory which can be used to store meta-information or error correction codes. This area is called the out of bounds area (OOB) and is typically not used to store data. With some NAND flash devices, the OOB area have the ability to support partial rewrites, where the OOB area is broken into smaller subsections. This allows

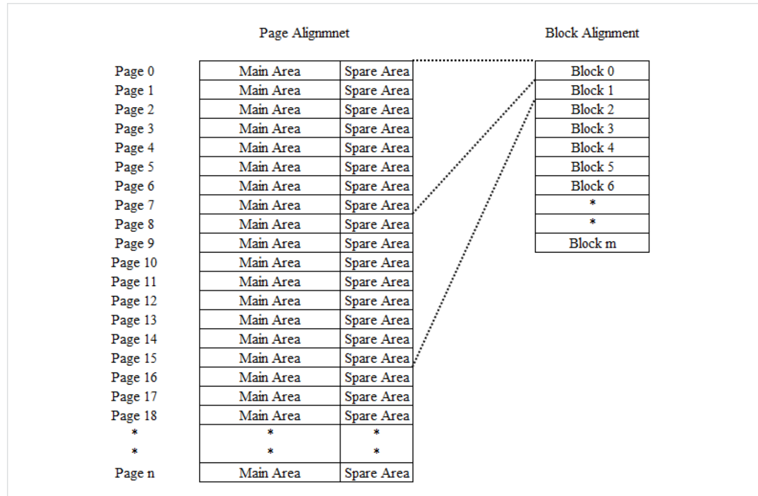


Figure 2.10: Flash Page Organization

for the OOB area for a single page to be re-written a limited number of times before having to be erased [Inc06]. The characteristic is called *Partial-Page Programming* (PPP).

Definition 2.5. *Partial-page programming* allows a single NAND flash page to accommodate a limited number of write operations to distinct sub-divisions of a page before the page is required to be erased.

Partial page programming does not allow previously written areas to be re-written, but allows sections of a page to be written in isolation. The number of sections is limited and this features is not available in all NAND flash devices [Inc06].

Pages are grouped into *blocks*. A block is the minimum erasable unit and can be identified with a physical block number. That is, if you wish to erase the contents of a specific single page, the entire block that the page is a member of must be erased.

Definition 2.6. A *block* is a contiguous set of pages in flash memory and is manufacturer dependent. A block is the minimum unit of erasable data in a flash device [CPP+09].

Depending on the specific implementation, a block many contain numerous pages and the size of a block (in terms of the number of pages contained within one block) depends on the device specifications set forth

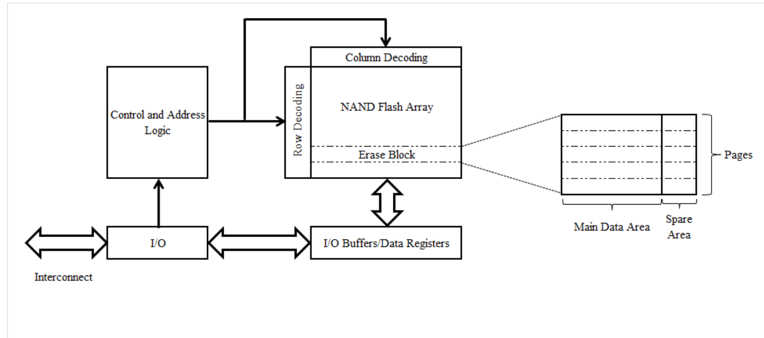


Figure 2.11: Architecture of a NAND Flash Memory [Des10, DZ11]

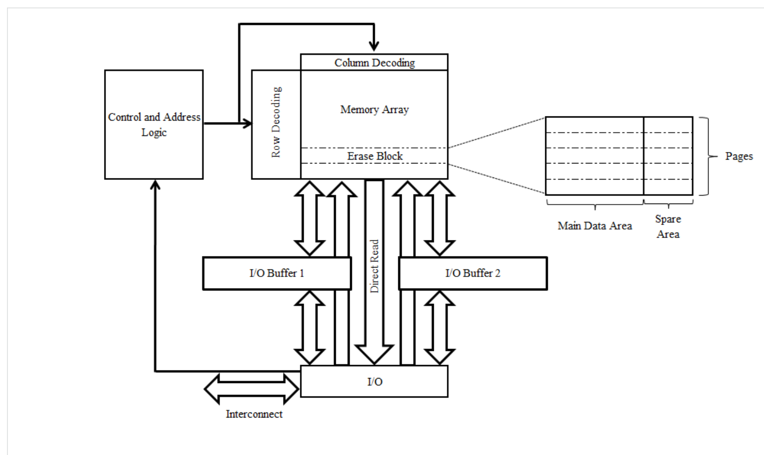


Figure 2.12: Dual Buffer Serial Dataflash

by the manufacturer. As a result, the size of the block is not universally consistent. There are notable exceptions to this rule. With the Atmel/Adesto Dataflash [Atm04, Ade15], the device can also be erased at a page level (for a significantly higher cost in terms of energy and time).

Data in flash memory cannot be directly read or written under normal circumstances. All data movement occurs through an on board SRAM buffer that is used to shadow a page of flash (Figure 2.11). In order to read data, the host requires that the flash memory move the desired page into the page buffer due to the page alignment configuration. Once the page has been moved successfully into the buffer, the host can request the data be transferred from the device. Conversely, when a page is to be updated the entire page must be copied into the buffer, updated and then written back into an erased page due to the erase before write requirement.

The constraint of data movement into and out of the buffer introduces latency and complexity into read and write operations. To address theses limitations, Atmel [Atm04] introduced a dual buffer NOR-based Dataflash (Figure 2.12) which introduces a second buffer into the architecture. The configuration reduces latency by allowing concurrent non-overlapping operations to occur. For example, a page from main memory can be transferred to one of the buffers while the host reads live data from the other. Pages in one buffer can also be written concurrently while reading data from the second buffer. The architecture also introduces a direct read option due to the NOR configuration. This unique operation allows a data stream to be directly read from the flash memory without disturbing the data held in the buffers which presents a significant performance advantage for this memory for read intensive applications. In a performance analysis comparing the cost in terms of read latency between buffered and direct reads, Fazackerley and Lawrence [FL11] demonstrate that even for the smallest readable data element direct reads will always out perform buffered reads. This is due to the setup and transfer time of moving data from a flash page to one of the internal buffers.

Limitations of Flash Memory

While flash memory is low cost and relatively robust, it presents some unique challenges: it has a finite erase/write count as well as not supporting in-place updates. Attempts to mitigate these issues are done through write normalization and wear levelling in an effort to evenly distribute erase and writes uniformly across the entire device. Use of flash memory without the use of wear levelling or other erase/write normalization strategies can

accelerate device failure [FL11]. While there are numerous strategies for flash management that present novel and efficient methods for storing the data, they all rely on extensive sensor processor resources in terms of RAM or EEPROM which limits their practical use. Additionally, the strategies are targeted for parallel NAND flash and will not transfer well in terms of performance to serial NOR flash.

As with any technology, there are trade-offs in terms of cost, performance and usability. As pages in flash memory degrade over a series of erases and has a finite lifetime they will eventually not be able to reliably encode data. At this point, the page is considered to have failed. It is up to the host system to track the consistency of data or erase cycles to plan for eventual failure. Some devices offer read back after write verification to check for consistency [Atm04]. To ensure that a device ages uniformly, efforts are made to write and erase pages uniformly across the device. In order to accomplish this, host systems utilize wear levelling algorithms (WL) that introduce additional overhead into device operations [KRKC11]. Typically the wear levelling algorithm is transparent to the user and ensures that data writes are spread evenly across the device.

In addition to wearability, flash memory suffers from asymmetric read and write costs. Typically, write times will be orders of magnitude longer than read times. From an implementation perspective, this is different from other storage media such as rotating media which generally is symmetric in terms of performance. Another significant difference from rotating media, is that a unit of data cannot be permuted in place. Before data can be written to flash memory, the target cell must be erased. This is referred to as *erase before write* constraint [CPP+09]. The erase costs (in terms of time and energy) are more expensive than other operations. Additionally, most architectures require that data be erased in blocks which introduces additional complexity from a management perspective [AB99] and limits its usability as a drop in replacement for other memory types.

Another effort to increase the density and performance of devices is the large block device but introduces further restrictions. A large block device may have page sizes in the order of 2 kB with 64 pages per block [Ele06]. With a large block device, pages within a single block are required to be programmed in sequential order [JKJ+10] to minimize the risk of write disturb errors of adjacent memory cells, they do not allow for random page programming [Ele06]. The implication of this is significant as it means that if page 20 of a 64 page block needs to be programmed, pages 0 through 19 need to be programmed in advance of page 20. While this technology does increase the capacity of the device, it does introduce challenges for utilizing

the technology on devices that have limited SRAM memory in terms of data management and buffering.

2.2.3 Other Storage Technologies

Recent investigations have made significant gains in newer memory technologies that address the limitations of previous generations of NVRAM. Ferroelectric RAM (*FRAM*) has started to gain some inroads for some applications due to its very high endurance (10^{10} to 10^{12} read/write cycles [Lim11, Ram12]), low power and fast write times. Unlike flash memory, it does not require a memory location to be erased before being written and is byte addressable. Unfortunately, device capacity is still relatively low (8 kB) [Lim11, Ram12] with a high cost per byte compared to flash memory technologies.

Another alternative NVRAM on the emerging horizon is the magnetoresistive RAM (*MRAM*). It has been speculated that MRAM is a promising replacement for Flash memory as a universal memory [Ake05]. Different from most other memory devices that encode binary information through charge levels, MRAM utilizes magnetic storage cells. One of its desirable features is that it has infinite endurance and will not fail through use like with EEPROM based technologies. Due to the physical architecture and methods of operation, MRAM can operate at high speeds, has symmetric read/write costs, a long data retention period and is a byte addressable [Tec15]. It appears to combine the best characteristics of SRAM and Flash memory technologies but has not seen a high adoption level. The largest drawback is that capacities are still low [Tec15] when compared to other viable candidates as well as having a high per bit cost for storage. Eventually, MRAM may find its place but until the capacity increases and cost decreases, its adoption rate will be low.

Other recent works have examined utilizing phase-change memory as a component of a non-volatile solution [KLCB08, PKCH10, PPP11]. This technology is different from other non-volatile storage media which encode information electrically or magnetically. It relies on a physical change in state of the memory matrix to encode information [SKF+10] leading to stable, long term storage. It is a byte addressable technology that does not require erase-before-write as with flash memory technologies [ea04, KK04]. PRAM's inherent stability, and high endurance [Inc05] would seem to make this emerging technology desirable for use as a replacement for flash memory [KLCB08]; claims have also been made that significant power savings can be made over flash memory technologies [Bar08] but in reality, currently available devices

show substantially higher current draws [Inc05] when compared to other technologies [Inc08, Atm04, Ele05]. To further confound this claim, devices demonstrate instability and propensity for data loss at moderately high temperatures and have a considerably smaller operating window than other devices [PRP$^+$04], limiting its use for harsh environments. PRAM's low density, write performance and cost [KLCB08] has lead to a slow adoption rate for use in embedded applications.

As a result of the limitations with other non-volatile memory technologies, flash memory has emerged as the predominant replacement for previous NVRAM technologies [SCKS08] as it offers a cost effective, relatively robust, flexible and energy efficient storage media for embedded systems [BCMV03] but still requires a storage strategy to ensure that data will be consistent.

Table 2.1 compares and contrasts the different memory options that are available as memory storage options for embedded systems. The remainder of this work focuses on the use of flash memory and supporting operations that allow for robust data storage with flash media. Section 3 will further expand on the limitations of flash memory in general purpose computing systems, the differences between flash memory and traditional storage media and translation strategies for flash memories.

Table 2.1: A Comparison of Memory Stores for Embedded Systems.

Memory Type	Pros	Cons	Cost/Byte	Read/Write Speeds
SRAM	Fast. Values stored as long as device is powered. Byte erasable. Erase-before-write not required.	Non-persistent. Low density. Cost.	High	Symmetric
NVRAM	Performance similar to SRAM. Byte erasable. Erase-before-write not required. Operates like SRAM.	Requires continual backup power to prevent loss of data. Cost.	High	Symmetric
EEPROM	Byte programmable. Low power consumption.	Erase-before-write required. Low density. Block erase required.	Medium	Asymmetric read/write
E^2PROM	Byte programmable. Byte erasable. Low power consumption.	Erase-before-write required. Low density	Medium	Asymmetric read/write
NOR Flash	Medium density. ECC not required.	Erase/write endurance. Requires flash management strategy due to erase-before-write constraints. Single bytes not erasable (erase blocks required).	Medium-low	Asymmetric read/write
NAND Flash	High density. Low current requirements.	Erase/write endurance. Requires flash management strategy due to erase-before-write constraints. Page addressable. Requires ECC. Suffers from write disturb. Single bytes not erasable (erase blocks required).	Low	Asymmetric read/write
FRAM	High endurance. Low power requirements. Fast write times. Erase-before-write not required. Device only requires power during read/write operations.	Low density. High cost. Destructive reads. Requires rewriting after reads.	High	Fast writes
MRAM	Infinite endurance. Long data retention period. Symmetric read/write speeds.	Low density. High cost.	High	Symmetric read/write
PRAM	High stability and endurance. Byte addressable. Erase-before-write not required.	Limited temperature operating window. High temperature exposure leads to data loss. Low density.	High	Asymmetric read/write

Chapter 3

Data Persistence Strategies

> The beginning of knowledge is the discovery of something we do not understand.
>
> Frank Herbert - Dune (1920 - 1986)

3.1 Data Management Strategies

Numerous challenges exist using a flash based memory strategy for data persistence in an embedded system.With flash memory, data cannot be mutated in-place. The data must be first copied out along with all other data in the same block at which point the block erased.

Definition 3.1. A block of pages that have been selected for erasure is called a *victim block*.

Once the block is erased, the page can be re-written to the same physical location along with the unchanged pages. This limitation is referred to as the erase before write constraint.

Definition 3.2. The limitation that a flash page must be erased before data can be re-written to the same physical location is called *erase-before-write*.

Second, flash memory has significant asymmetric read and write costs both in terms of the total time for the operation to complete and the amount of energy required for the operation. The behaviour of erase before write generally causes significant challenges for traditional file systems such as FAT16 or FAT32, NTFS, and ext3 among others. This is because the write behaviour is drastically different from rotating magnetic media [PH12, p.581] as a write operation now becomes two distinct operations. Two approaches had been suggested to deal with this challenge [CPP^{+}09, ZYWY09, KRKC11]. The first approach is to use a flash aware file system that has been specifically

designed to accommodate and exploit flash device attributes. Flash aware files systems will be further expanded on in Section 3.2. The second approach is to use the aforementioned translation layer to act as an intermediary between the physical device and the service or application requiring the use of storage (being either a file system or direct application requiring raw device access). This address translation is referred to as a Flash Translation Layer (FTL). This layer is responsible for swizzling data addressed as it is physically relocated. The FTL is further discussed in Section 3.3. Regardless of the strategy used, flash-based memory technologies must utilize an address translation scheme and a write normalization strategy to be considered functionally useful [MFL11].

Block erasure can cause significant management issues for the host system as other pages in the target block may contain data that is currently being used by the system. Pages containing active data are referred to as *live* pages.

Definition 3.3. A *live page* is a page that contains data that is currently being used by the host system.

Before an erase can proceed on a block with live pages, the data needs to be copied out to prevent loss. The management of this process and the act of copying introduces significant overhead into the process of re-writing in-place. As an alternate strategy, instead of writing the page back to the same physical location, the page could be written to an already cleared area leaving the remaining pages in the original block unperturbed.

When a page is physically moved, management challenges are introduced into the system using the data if physical data referentiality is being used. To combat this potential problem, services or applications on the host will refer to data through a translation interface that will offer a logical address. The interface will manage the mapping between the supplied logical address and the physical (on the device) address of the data page allowing for maintenance of referential integrity while being transparent to the user.

3.2 Flash Aware File Systems

In general, most file systems assume that data can be mutated in-place which introduces significant challenges for flash memory. For file systems to interact in a more favourable fashion with flash memory, focus has shifted to file system architectures that do not use in-place updates. Section 3.2.1 will discuss the evolution of file system for hard disk drives that do not rely on

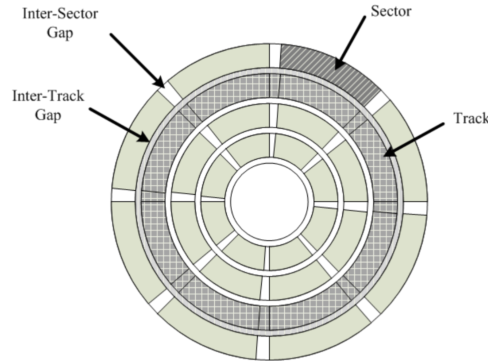


Figure 3.1: Rotating Magnetic Hard Disk Drive

in-place updates. Section 3.2.2 examines current trends for flash specific file systems.

3.2.1 Log Based File Systems

The most common data store for general purpose computing is the rotating magnetic drive [Sta13] and is commonly found in most non-embedded computing applications. The most commonly available hard disk drive used today was developed by IBM in 1973 [AF02] and is referred to as the Winchester drive. It was a significant developmental milestone as it introduced the concept of a low mass flying magnetic head that was able to move across the magnetizable platters to different locations under servo control which is now the defacto standard for all rotation magnetic media [Sta13]. The rotating magnetic drive consists of a series of rotating platters coated with a magnetizable substrate that allows the surface to magnetized using a recording head. In order to store a bit of data to the platter, the head induces a magnetic field onto the patter at a given location; the absence of a magnetic field is used to record a 0 bit. Hard disk platters are physically partitioned into concentric circles called tracks (Figure 3.1) separated by gaps, which allows that hard disk head to move to a certain position and read data from the track moving under the head. The track is further divided into blocks or sectors. The gaps between sectors and tracks play a critical role in data access as it allows for alignment and timing for data.

Definition 3.4. The *sector* or *block* is the minimal readable or writable unit of data for a rotating magnetic hard disk drive [Sta13].

A hard disk manufacturer will define a sector and track size for a device,

and the internal control circuitry for the disk will be responsible for moving the head to the requested position. When reading or writing data to or from the disk, two operations are required. First, the internal control circuitry will move the head to the correct physical track on the drive and the associated time is referred to as the seek time.

Definition 3.5. The *seek time* is the time it takes for the read head to move from its current track to the target track as specified by the hard disk controller.

In the best case, the head will not have to move as it may already be on the correct track; in the worst case, the head will have to move from the inner most track to the outer most track. Once the head has been moved to the correct track, the controller must wait for the correct sector to pass under the head such that the data can be then written or read from the device.

Definition 3.6. The *rotational latency* is defined as the time it takes for the target sector to move under the read/write head. It is frequently expressed as the average rotational latency and computed as the time it takes for one-half a rotation.

This action introduces significant latency into the system and is one of the most significant limitations of the technology. The delay incurred from the combined rotational latency and seek time is referred to as the disk access time and is a significant motivator in the move to non-mechanical storage solutions.

As a result of the high cost of mechanical movements in the system, numerous files systems have been developed in an effort to improve logical and physical data layout on the drive such that access times are minimized File systems will allocate adjacent sectors into a logical groups reserved for a single file to exploit spatial and temporal locality; when data is being accessed in one sector, the likelihood that adjacent sectors from within the same track will be accessed at the same time is high [PH12]. If sectors are physically grouped on the same track, then the access time can be minimized as a track-to-track seek. The grouping of sectors is referred to as an Extent.

Definition 3.7. An *extent* is a contiguous area of storage that has been pre-allocated by a file system. Extents are also known as block runs [Gia99, p.9] as logical data is allowed to span multiple contiguous physical blocks.

With the formation of extents, file systems attempt to minimize file fragmentation, where contiguous logical data is written in different sectors

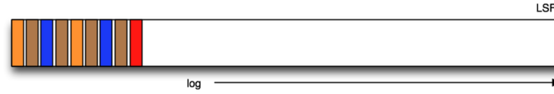
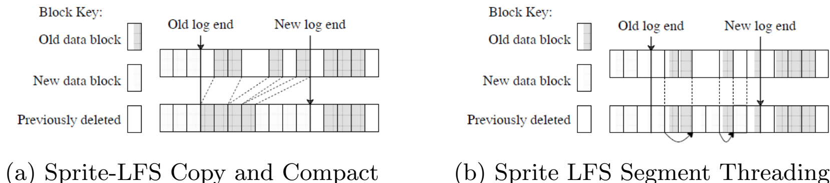


Figure 3.2: A Log Structured File System

or tracks leading to high access times when contiguous logical bytes are being accessed. Extents are especially useful when logical byte streams are being written to or read from disk as it allows the file system to access contiguous blocks without having to access a separate track [Gia99, p.16]. Rosenblum and Ousterhout [RO92] first proposed a novel file system to exploit extent access and thus minimize access times called Sprite-LFS. Their work proposed a log-structured file system that offered significantly improved access times over other files systems. It is built on the premise that with increased memory and cache sizes, large components of files can be stored in SRAM memory [ODH+85]. The caching of data in memory can be used to satisfy read requests, allowing disk access patterns to be dominated by writes [RO92]. With traditional file systems, when a byte is to be written, the target sector and track is located, the head is moved to the correct track position and the value is updated in-place as allowed by the re-writable nature of magnetic media. While efficient in terms of space utilization, a time penalty is incurred with the access time; if the access pattern for writes is random in terms of logical location within a file or between files, the access times can dominate over write times consuming up to 90% of the total bandwidth [RO92]. While the idea of utilizing a log to improve file system performance was introduced by Hagmann [Hag87, ea90], these systems used logging primarily as a support structure for crash recovery as opposed to focusing on improvements in write speeds [RO92]. With the log-structured file system, the log was used for a fundamentally different purpose. With previous works, the log was used as a temporary data store but Rosenblum and Ousterhout proposed using the log as the primary data store and maintaining no other structures on disk [RO92]. While a radical design departure from other file systems, the concept proved to be appropriate for write intensive applications and offered an order of magnitude improvement in performance with respect to raw write speeds [RO92].

In a log based file system, all data regardless of whether it is an update to an existing block or a new block is written to the tail of the log (Figure 3.2). This allows writes to be made in a continual sequential order without having to incur unnecessary access times as the head moves to the target block to



(a) Sprite-LFS Copy and Compact (b) Sprite LFS Segment Threading

Figure 3.3: Sprite-LFS Space Management Strategies

be updated. As blocks are updated, a version numbering scheme is used to track the most current version of a block. Sprite-LFS also utilizes fixed sized extents called segments [RO92] that are composed of numerous blocks. The file system will buffer writes to disk until it can completely fill a segment and then writes the complete segment, which can impose significant SRAM requirements. For optimal performance of the file system, a number of large free extents must be maintained. As blocks are invalidated in the log due to multiple updates to the same block, the log will become occupied with stale data. Beside occupying space, this also leads to segment fragmentation which impacts performance. To overcome this, the log will periodically require cleaning to remove any stale blocks with the exception of the most current block. An operational constraint of Sprite-LFS is that any block in a segment cannot be rewritten until all live data has been removed. To deal with this, Sprite-LFS uses two different strategies; the first being a copy and compact strategy and the second being a segment interleaving strategy known as a threaded log. When the file system performs a compact and copy, long lived data is combined into a single segment as shown in Figure 3.3a. Unfortunately, the copying operations introduce additional delay. The second strategy used is segment interleaving where the file system will overwrite or create new segments in areas that are completely stale that may be located between two active segments. While this does improve space utilization, Rosenblum and Ousterhout note that this will lead to increased fragmentation and limits the number of large contiguous writes, eliminating any advantage of the file system [RO92]. Sprite-LFS uses a combination of both strategies to ensure that an acceptable number of suitably sized extents are available, but these processes can negatively impact the performance of the file system under certain workloads [SBMS93].

While Sprite-LFS introduced improvements in terms of write performance, limitations before it would be suitable for use in a production environment include issues with a large memory footprint and write validation issues if

disk space was not available [SBMS93]. Regardless, it is considered to be the foundational work for all log based files systems as well as offering potential solutions for performance related issues with flash memory.

3.2.2 Logging Based File Systems for Flash

One of the fundamental challenges encountered with the introduction of Flash technology in secondary storage is that it does not have the same write properties as rotating magnetic media. While Flash memory does not suffer from issues of access delay due to the lack of moving parts, they have a significant limitation when working with files systems that are designed for devices that re-write in-place. With the inability of flash to re-write data without first erasing a page, traditional file systems that mutate data in-place offer challenges both in terms of the amount of time to complete the operations as well as accelerating the wear on the device and shortening the service life.

While flash memory does not have the same limitations in terms of seek and rotational latency that drove the development of the log-base file system [RO92, SBMS93], Kawaguchi et al. [KNM95] observed that the write pattern of logging files systems would be compatible with Flash technologies as they do not rewrite blocks or extents in-place; they only append changes or additions to the end of the log of disk which is compatible with the attributes of flash memory. Kawaguchi et al. successfully implemented a log based file system on flash memory but not without encountering some limitations in terms of performance with mixed hot and cold data as well as the impact of cleaning operations on overall performance. Kawaguchi et al. encountered problems with data consistency as well as suffering from increased write times as the amount of data being stored increased due to a decrease in the ratio of valid to invalid blocks. This forced the system to be engaged in a large number of erase operations with as much as 60% of the total operations being consumed by writes associated with valid page movement during erasures [KNM95].

With the use of flash memory in wireless sensor networks, work has focused on developing file systems that can accommodate its different performance characteristics. While suitable for general purpose computing, a direct implementation of these systems on a microprocessor is infeasible due to lack of system resources [DNH04]. Custom designed file systems such as JFFS, JFFS2 [Woo01] and YAFFS [LP06], which share a common evolutionary point with LFS, are not suitable for use in resource constrained systems as they maintain data structures in RAM for file system control. These flash

file systems are designed to work on systems with large amounts of RAM and processing power.

Smaller systems have evolved from these that are more suitable for wireless sensor platforms that are constrained in terms of power and resources. Matchbox [GLvB+03], a byte structured file system implemented in TinyOS, supports basic wear levelling as well as multiple files. It is designed for logging applications but supports only append operations making the modification of existing files not possible. Additionally, its RAM footprint will grow with the number of files in the system [DNH04].

In [DNH04], the authors present ELF, which is a log based file system. It relies on the host microprocessor EEPROM to store the directory structure for the system. Unlike traditional log based file systems, it caches multiple writes to reduce the write overhead. It maintains log entries on separate pages to improve fault tolerance. It also maintains a garbage collector to reclaim pages and relies on a page write counter stored in the metadata of each page. Garbage collection is only triggered when the number of free pages drops below a given threshold which is stored as a bitmap in SRAM. Overall, the SRAM requirements offer limitations in terms of application to memory constrained devices.

Microhash [ZYLK+05] provides a primitive framework for storing and indexing temporal data based on page chaining. It maintains numerous data structures in RAM and utilizes a naive garbage collection strategy. In practise, the run time RAM requirements make it infeasible for memory constrained devices [MDC+09].

Capsule is a cross device file system that uses object abstractions to store data [MDC+09]. If offers a wide selection of data objects such as stacks, streams and queues to store data but requires a large SRAM footprint. It relies on per object buffering in microprocessor RAM and is based on a logging file system. Similar to other systems, it supports a garbage collector which is triggered when the amount of available space in the system falls below a certain threshold. It also supports check pointing and rollback of objects.

In practise, regardless of the features offered by the flash aware file system, they must manage a logical to physical address translation scheme and write normalization. Table 3.1 compares and contrasts the features and limitations for flash aware file systems in addition to highlighting the target computing platform.

Table 3.1: A Comparison of Flash Aware File Systems.

File System	Medium	Platform	Pro	Cons
Sprite-LFS	Spindle Disk	General purpose	Log based system writes in sequential order Avoids random in-place writes.	Large SRAM requirements. Unsuitable for resources constrained systems.
LFS (Kawaguchi)	Flash	General purpose	Appends modifications to end of sequential log.	Mixing of hot and cold data impacts performance. Operations dominated by valid page movement during erase operations. Unsuitable for resources constrained systems.
JFFS, JFFS2	Flash	General purpose	Avoids random in-place writes.	Large RAM requirements. Unsuitable for resources constrained systems.
YAFFS	Flash	General purpose	Support for large block devices. Data integrity.	Large RAM requirements. Targeted for NAND flash. Unsuitable for resources constrained systems.
Matchbox	Flash	Embedded (TinyOS)	Supports multiple files and basic wear levelling.	Append only operations. SRAM footprint grows with number of files in system.
ELF	Flash	Embedded	Caches writes. Directory structure maintained in EEPROM. Maintains logs on separate pages for improved fault tolerance.	Large SRAM requirements.
Microhash	Flash	Embedded	Supports indexing for temporal data.	Large SRAM requirements.
Capsule	Flash	Embedded	Provides data abstraction layer. Supports data objects such as stacks, streams and queues.	Large SRAM requirements.

3.3 Flash Translation Layers

Flash translation layers are responsible for providing a mapping scheme between logical units of data in an application to physical units of data on a device. Depending on the design of the FTL, there may or may not be a direct mapping between the logical and physical storage units.

Definition 3.8. A *logical page* is the smallest read or writable unit of data from an application or service [ZYWY09, DZ11].

Chung et al. [CPP+09] suggest that the FTL is actually composed of three components. In addition to the logical to physical mapping, the FTL also needs to provide wear levelling to ensure uniform wear across the device which in turn will maximize lifetime due to erase failure. Further, the FTL needs to handle power off recovery (POR) as the FTL may become corrupt in the event of a sudden loss of power which is common in embedded systems. On restart, the FTL needs to recover and ensure that the data on the device is consistent and available. Deng and Zhou [DZ11] further suggest that a fourth component should also be included in the FTL which is Garbage Collection (GC). In the majority of previous works, focus has strictly been on more efficient storage by minimizing erase operations and thus increasing the life of the device. The amount of data that is moved between the host in addition to the amount of energy expended in the operations has not been considered; both of which are key points of consideration for memory, pin and power constrained devices.

As pages are relocated through re-writes or wear levelling, old pages are left un-erased and un-used. If left alone, the device will eventually run out of erased or free pages. Pages that have been abandoned during this process are invalid. Pages that contain data are considered to be active, valid or live.

Definition 3.9. A *free page* is a page that has been erased and does not contain any data.

Definition 3.10. An *invalid page* is a page that contains data but is no longer linked to an application, service or active logical mapping.

Definition 3.11. An *active*, *valid* or *live page* is a page that is involved in an active mapping and is linked logically to an application or service.

To deal with the issue of increased invalid pages during operation, the FTL needs to track the state of each page in terms of use (that being free, active or invalid) through direct or indirect means. When data is moved,

the original page is marked as invalid through a tracking method. When the number of invalid pages on the device reaches a given threshold, the FTL will trigger the garbage collection process. This will in turn erase invalid pages, converting them back to free pages. In the process of garbage collecting, valid pages may be sharing a block with invalid pages. In order to prevent the loss of data during erasure, the valid pages are moved out of the block being targeted for erase which is also referred to as compaction. Once moved, the block erasure can proceed.

3.3.1 FTL Taxonomies

In previous works, focus has primarily been on three different styles of FTLs for NAND flash and how these algorithms can be made as efficient as possible for that architecture [CPP+09, DZ11]. Little work has focused on viable FTL algorithms for serial NOR flash and fail to address key concerns for constrained devices. FTL designs are strongly related to memory cache strategies and share common design traits with either a fully associative cache, n-way associative cache or a directed mapped cache [PH12, pp.479-481]. Using these models, FTLs are generally categorized as page level (fully associative) mapping technique which provides a high level of granularity and can be considered to be fully associative, block level (direct mapped) mapping technique which has a lower degree of granularity and a hybrid (n-way) mapping level technique which borrows from techniques used in both page and block mapping schemes. The following sections will examine fully associative or page level techniques, block level mapping and hybrid mapping techniques. Each technique offers trade offs in terms of SRAM footprint, performance and endurance. Performance is gauged by the minimization of erase operations and the maximization of block usage before erase. Strategies do not consider the data transfer and energy costs.

Fully Associative Translation Layers

A page level or fully associative translation layer is considered to be a naive algorithm [CPP+09] that was first presented by Ban [AB95] in 1995. In previous works, page mapping schemes such as DAC [CLC99] generally outperform other taxonomies [LJKK08] but have the largest SRAM footprint as a mapping must be stored for each page. Regardless, it offers the best performance in terms of device utilization There is a one to one mapping between physical and logical pages [DZ11]. This is typically stored as a 2-tuple with the logical page number as the primary key as can been seen in

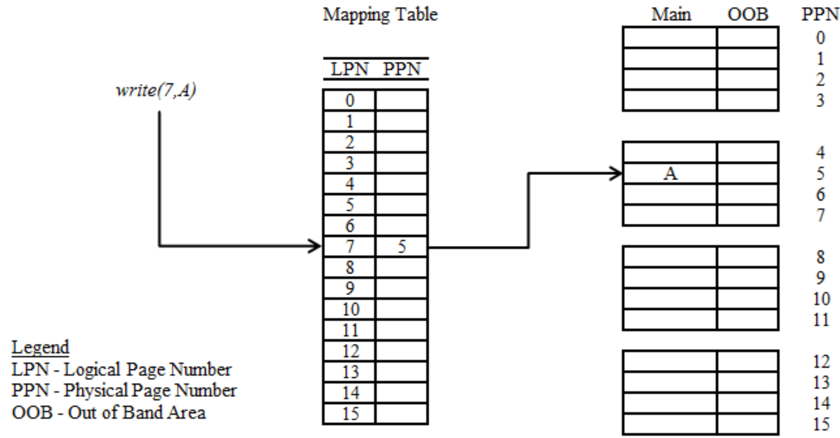


Figure 3.4: Page Mapping Scheme for Flash Address Translation

Figure 3.4. In this example a flash memory consists of 16 pages with four pages per block. The blocks are classified as either data (D) blocks or update (U) blocks [CLC99, DZ11]. Data blocks are used by FTL to store application data and U blocks are used by the FTL for administrative purposes. In this example there are three D blocks and one U block. When a write command is issued to the FTL such as *write (7,A)*, the FTL will attempt to write A to the logical page number 7. To accomplish this, the FTL will lookup the mapping record of logical page 7 and see that it maps to physical page 5. The FTL will write A to physical page 5 if it is free. If the physical page is active or invalid, the FTL will select a free page to write the data to, update the logical to physical page mapping and then mark physical page 5 as invalid.

If there are no free pages in the system, the FTL must select a victim block to erase which incurs a management cost. The block may be selected based on the largest number of invalid pages or some other victim block metric [CPP+09]. The FTL will then move the active pages from the victim block and the page being written to the U block. The mapping table will then be updated to reflect the changes. The FTL will then erase the victim block which will become the new U block.

In order to offer recoverability, page mapped FTLs must ensure consistency in the mapping table after a power off (either planned or unplanned). This can be done by periodically flushing the mapping table to flash memory or by encoding logical page numbers in the out of band area for each page.

After a restart, the FTL must scan to find or rebuild the table [CPP+09] before the FTL is usable.

A flash memory with n pages requires n tuples in SRAM. This can prove to be a daunting method for both general purpose computers and memory constrained devices. Consider the following case for a 1 megabyte flash memory device with a 512 byte page size. For this device, there are 2,048 pages that need to be mapped in SRAM. To encode this address space, a short (2-byte) int can be used. Thus, using a 2-tuple of 2 short ints (one to encode the logical page, one to encode the physical page), each tuple is 4 bytes in size producing a mapping table of 8,192 bytes. While this seems to be quite small, this table would be impossible to encode on most 8-bit microprocessors due to the limited amount of SRAM.

In the design of the paged mapped FTL, Deng and Zhou [DZ11] suggest that two key items must be considered. First, the size of the mapping table can present a significant challenge especially for highly memory constrained devices as each physical page must have one mapping tuple in the translation table. To further complicate matters, the table needs to be persistent; thus it also needs to be stored in non-volatile memory. The memory constraints can be reduced as only a small part of the table may actually be in SRAM at any one time depending on the nature of the access patterns [GKU09]. To accomplish this, caching strategies can be employed to reduce the amount of thrashing on address translations that may occur and impact overall system efficiency.

Definition 3.12. *Thrashing* is the rapid exchange of data from memory with data in secondary memory (flash or disk storage).

The second major consideration highlighted is that of overall query efficiency. The performance of queries on the FTL should not degrade with the growth of the system which may be the case if multiple translation tables need to be brought into memory.

When compared to the block level and hybrid schemes, the paged mapped strategy offers the best performance [CLC99, CPP+09, GKU09, DZ11]. This is partially due to the ability of page mapped FTLs to delay erases as long as possible as a result of the flexible and unconstrained page placement policies. The scheme ensures that an erase block is fully utilized before being targeted for erase. Additionally, there is no requirement that data be clustered into block regions with reduces the overhead on the FTL [JKJ+10]; as long as pages are free and above the given threshold, the system can continue to operated without having to invoke the garbage collector. Little work has been put into page level schemes for NAND and NOR flash due to

the stigma over performance restrictions with SRAM based mapping tables. Two works have emerged that address these concerns: DFTL [GKU09] and LazyFTL [MFL11]. Both compare performance to hybrid and block level mapping strategies highlighting the fact that page mapped schemes will offer better performance in terms of flash utilization, erasure and wear.

Block Level Translation Layers

A block mapped translation scheme makes a trade off in terms of performance for size of translation table footprint. Unlike page mapped strategies where there is a high level of granularity, block mapped strategies group pages into blocks that align with erase units on the physical flash memory device and are by far the most built upon strategy.

With a block level FTL, pages are still available at the logical level but are grouped into logical blocks for write operations by the system. Each logical block will map through to a physical block in flash memory. Each logical page in the logical block will map to one and only one physical page in the physical block based on its offset in the block [CPP+09, DZ11]. Because of the spatial page restrictions on where a page can exist in a block, block level translations are referred to as in-place schemes [DZ11]. A significant limitation of block mapped schemes is due to the erase-before-write nature of flash. With a block mapping scheme, when a single page is updated the entire block must first be erased. Instead of erasing in-place, the block (both valid and invalid pages) with the exception of the target page will be copied to an erased block and then the update will be made which can incur significant management overhead [DZ11].

Unlike the previous strategy, block mapping techniques present m logical blocks in the mapping table where each block can contain between 1 and n logical pages. The record consists of a 2-tuple mapping between logical block number and physical block number with the logical block number as the primary key. Figure 3.5 shows the general layout of a block mapped FTL. In the simplest form a hash function determines what logical block a page is located in, and is computed as

$$LBN = LPN \div m \tag{3.1}$$

where LPN is the logical page number of the page to be accessed, LBN is the logical block number a logical page will belong to and m is the number of available blocks. The logical block is mapped to a physical block number and stored in the FTL mapping table. The physical page number of the

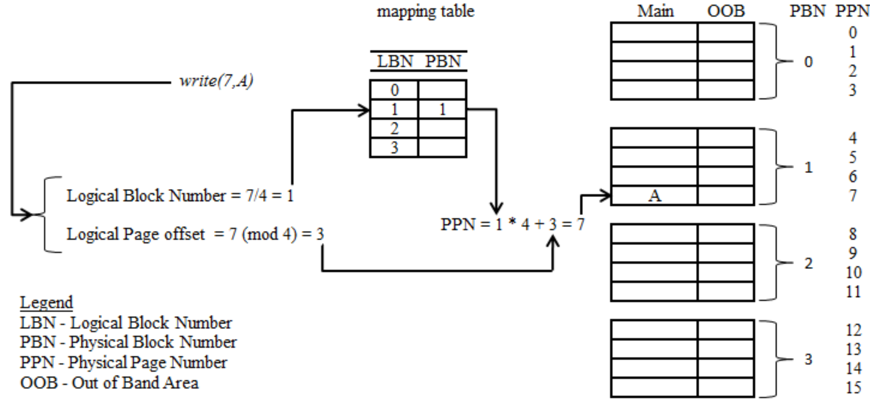


Figure 3.5: Block Mapping Scheme for Flash Address Translation

logical page is calculated as

$$PPN_{offset} = LPN \pmod n \tag{3.2}$$

where PPN_{offset} is the physical page number within a given block as an offset relative to the first page of the block and n is the size of the block in pages.

Consider the example in Figure 3.5 in which the flash memory consists of 4 blocks and each block consists of 4 pages where the write command *write (7,A)* is issued. The operation will attempt to write A to logical page 7. To accomplish this, the logical block must first be computed using Equation (3.1)

$$\begin{aligned} LBN &= LPN \div m & (3.3)\\ &= 7 \div 4 & (3.4)\\ &= 1 & (3.5)\end{aligned}$$

where the logical page will have membership in logical block 1. Using this value, the FTL will determine the physical block that the page will be located in. The physical page offset will then be calculate using Equation (3.2) as

$$\begin{aligned} PPN_{offset} &= LPN \pmod n & (3.6)\\ &= 7 \pmod 4 & (3.7)\\ &= 3 & (3.8)\end{aligned}$$

where the offset of the logical page in the block will be 3. Once the logical block number and physical page offset have been calculated, the FTL will retrieve the physical block number from the mapping table.

Chung et al. [CPP+09] note that while block mapping can reduce the size footprint of the mapping table in SRAM, there are performance limitations if the system issues sequential write commands to pages with membership in the same logical block. This will lead to thrashing of a block due to the increased number of re-writes required [CPP+09]. This will cause overall system performance degradation [DZ11] as the system will have to deal with a copy and erase for every operation. This performance concern has been addressed with the introduction of log based block mapping schemes [LPC+07, CP07, KC08, KJKL06, LSKK08, CLP09] which can be categorized as performance enhancing algorithms [KRKC11] as the main design focus is on reducing thrashing and minimizing the erase cost overhead.

A log block scheme reserves a series of pages in a logging block that are used for updates instead of having to continually re-write the target page and block. Eventually the logged pages need to be merged with the target block. The log pages and target block are copied to a new block in an operation called a *merge* [KKN+02]. Once the merge is complete, the original block and log block will be targeted for erasure. The specific method varies depending on the scheme. In the literature, the term *hybrid FTL* is often used to refer to a log based block mapped FTL. This leads to confusion as it is fundamentally different from a hybrid FTL and caution should be taken to disambiguate the two concepts.

Regardless, the hybrid strategies that are introduced in the following section employ block level mapping strategies and still suffer from performance issues regarding the conversion of log blocks to data blocks through the merging process.

Hybrid Translation Layers

Both the page level and block level mapping techniques present shortcomings in terms of translation table size in SRAM or erase-before-write limitations. To address these issues, a hybrid translation scheme [CPP+09] that offers performance advantages from both page level mapping and block mapping has been suggested in [KKN+02, BsKGyL02, KC08]. Similar to the approach used in block mapping schemes, hybrid mapping uses a logical to physical block mapping to locate a physical block [CPP+09, DZ11] which offers an attractive SRAM footprint for the translation table. As introduced previously in Section 2.6, a block consists of contiguous set of pages that

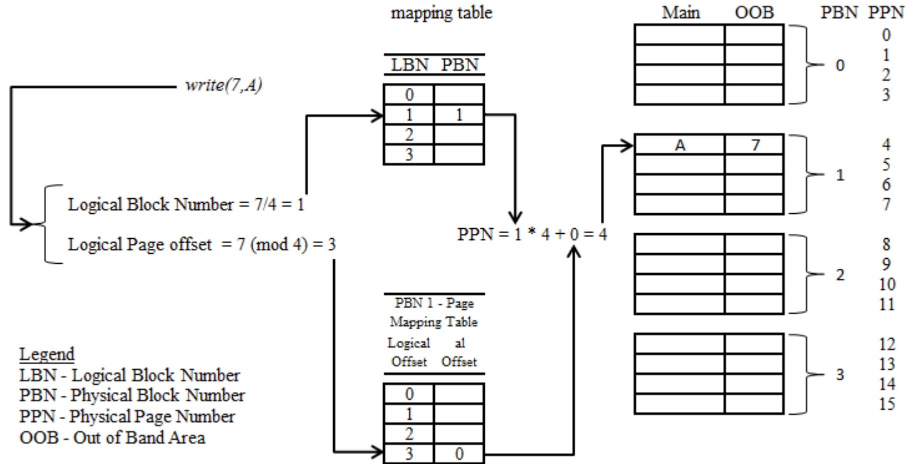


Figure 3.6: Hybrid Mapping Scheme for Flash Address Translation

form a minimal erase unit. The number of pages in a block is defined by the manufacturer and can vary between devices but typically start with eight pages per block as a minimum. Unlike block mapping, hybrid mapping techniques remove the direct page mapping requirement from within a block. Similar to a set-associative cache [PH12, pp.479-481], hybrid schemes allow for an intrablock page to be placed in any free page within the physical block. While this helps to manage the erase-before-write induced thrashing encountered by block mapping schemes, more complex intrablock page mappings are required as a single page may exist numerous times in a give block, especially under heavy sequential rewrites. As a result, the strategy introduces complexity as page mappings need to be stored on a block level. In the most naive form [TS99], the FTL will use a linear scan to find the most current version of a given page using information stored in the OOB area of a page. For small block sizes, this may prove to be an acceptable strategy but can become unmanageable as block sizes increase. Research improvements [LBP08, KLCB08, LYL09, JKJ+10, WLQS11] have focused on how to better encode page mapping information.

3.4 FTL Schemes

In practice, current FTL schemes often are built from the integration of page, block and hybrid schemes borrowing the best performing components

from each. As a result, challenges exist to strictly classify a scheme as one specific type. In the literature there is a large degree of idea recycling with subsequent works being released on relatively minor variations or improvements on a theme to offer increased performance based on a specific attribute. The following section introduces the commonly built on FTL strategies with an attempt to offer a grouping based on performance characteristics. Further details on specific FTL strategies are found in Appendix A.

3.4.1 Page Level FTL Schemes

DFTL: Demand Based Flash Translation Layer

While the page-level flash translation layer is an efficient design due to its similarity to a fully associative cache [PH12, p.479], Gupta, Kim and Urgaonkar [GKU09] highlight that this will result in a large translation table that will be stored in SRAM. They highlight that for a 16 GB memory device, the mapping table will be approximately 32 MB. Their work focuses on enterprise level databases utilizing NAND flash-based SSD technologies and is strictly a flash mapping interface. Highly memory constrained devices will not have anywhere near this level of SRAM available. To address this challenge they propose a method by which the amount of data stored in SRAM can be minimized through the selective caching of page-level address mappings. While the work shows a significant improvement in performance for enterprise level systems utilizing NAND flash, it proves to be an infeasible solution for memory and bandwidth constrained devices. From the memory constrained approach, they highlight a valid point; they claim that due to block level mappings, devices may suffer from a high-level internal fragmentation as well as performance degradation due to excessive garbage collection overheads [GKU09]. They argue that due to the temporal nature of data, the cache based strategy proposed can be a suitable solution for memory constrained devices. Unfortunately, this factor alone does not warrant its suitability for serial NOR flash devices. As highlighted previously with constrained SRAM sizes, this strategy would lead to heavy page-mapping trashing over the serial data bus as more time would be spent on mapping table transfers. This would lead to a performance decrease due to page-mapping movement. Gupta, Kim and Urgaonkar go on to further argue that due to the fact that they are a page-level map scheme, they minimize operations such as full and partial block level merges which are problematic sources of performance degradation in block level schemes which makes their strategy overly favourable. While this may be the case, the

system fails to consider data and page level mapping consistency as well as power on recoverability. While it may be a completely suitable strategy for enterprise-level systems it is infeasible for use in resource constrained systems.

CFTL

CFTL [PDD10] is a hybrid block mapping scheme that can switch between write and read optimized schemes dynamically, utilizing an efficient cache strategy for managing FTL mapping pages. At its core, it is a page mapped FTL, which is stored in flash memory. It stores in SRAM eight cache blocks and page map tables based on temporal and spatial locality of data. Additionally, a master mapping table which tracks all other mapping tables is also stored in SRAM. The authors claim that with spatial and temporal locality, this is important but with flash memory there are no variable latency penalties; the only penalty cost comes from having to look up translations in the FTL. Thus the FTL lookup will be dominated by nothing other than the cost of swapping in and out the lookup tables from flash memory. For a bandwidth constrained device, this presents a significant bottleneck and should be avoided. With the CFTL scheme, a write counter for each page is also maintained in SRAM which is used to identify hot and cold pages. This increases the overhead significantly in terms of both data storage structure and page re-allocation. While the authors assert some minor improvements, due to updates being able to be placed in any page as a result of the page level mapping, they fail to consider this complicates both garbage collection and wear levelling operations. They also fail to identify page conversions costs when switching from page to block level mapping. Additionally, they have failed to identify how block level access happens as well as how they handle consistency issues between flash memory tables and SRAM caches, issues with overall record integrity and power-on recovery. Performance comparisons are made against DFLT [GKU09] and claim significant improvements but fail to provide details on testing. While it may speed up FTL translations and look ups, it does not address any other core FTL design issues.

LazyFTL

Ma, Feng and Li [MFL11] present a low latency and high scalability FLT for enterprise use. They assert that their proposed FTL is the highest performance ever presented in terms of efficiency and effectiveness but limit their comparisons to hybrid and block-level mapping strategies. They do

not compare the performance of their FTL to DFTL or other page mapped strategies.

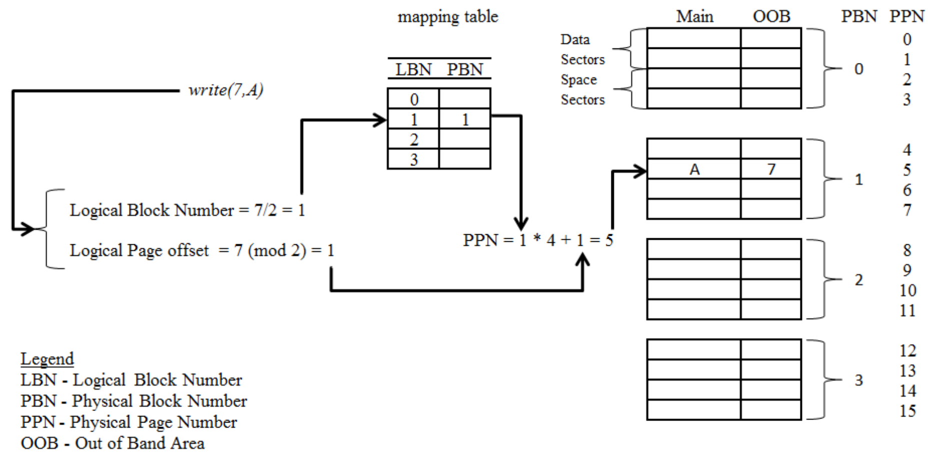
LazyFTL delivers a lower overhead system through the elimination of merges as well as improved system recovery, redundancy and fault tolerance by storing page mappings in a specific area of flash that gets updated in a lazy manner. The authors claim that since their strategy is a page based scheme, there is no need for merges which significantly reduces any overhead that may be encountered with block-based or hybrid mapping schemes. In performance comparisons, LazyFTL outperforms many other hybrid and block level mapping schemes. This is generally due to the fact that page level mapping schemes generally have higher resolution than other schemes [CPP+09].

With LazyFTL, page level mapping schemes are stored in flash and brought into SRAM as required, utilizing a least recently used (LRU) replacement strategy. While they claim that their scheme is unique, the principles of operation are similar to DFTL in terms of flushing page mappings back to flash. A limitation is encountered with how mappings are stored. As page mappings are stored in a defined area in flash, the memory device will suffer from non-uniform wear and contribute to premature device failure. It is not suitable for use with bandwidth constrained devices as this method results in large amounts of data being moved bidirectionally on a limited bandwidth bus. Additionally, memory constrained devices could not support this strategy due to SRAM limitations.

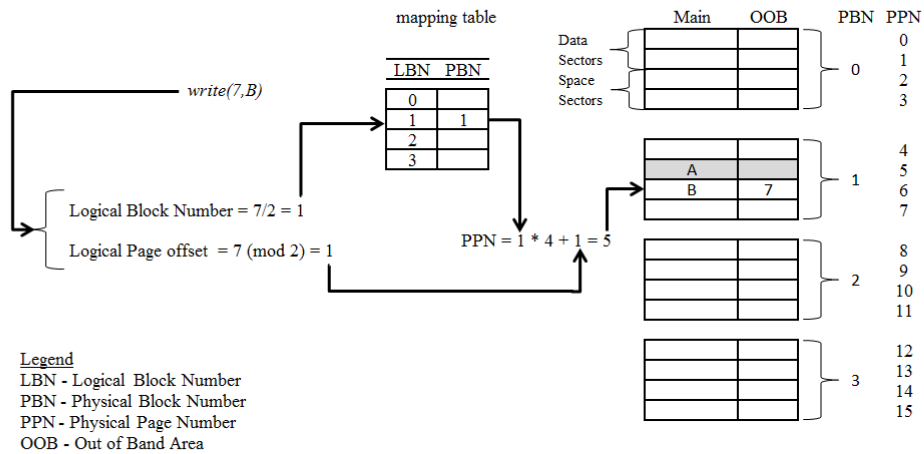
3.4.2 Page Based Logging Schemes

Mistubishi

One of the earliest hybrid schemes known as Mitsubishi was proposed by Takayuki Shinohara [TS99] in 1999 and is a log based design. While mappings occur at the block level on the flash device, the block is divided into two sets of pages. The first set of pages is used to maintain the first instance of new data written to a block. The second set of pages is used as a log and are termed by Shinohara as “space sectors” [KRKC11]. It is important to note that with this implementation data that is destined for one logical or physical block can only use the associated sector space for the block. As a result, the number of successive writes to a given block before it needs to be compressed or merged is limited by the log size and compounded by the use of fixed logical offsets. Additionally, the overall usable capacity of the memory is reduced due to the strict 1-to-1 association of a space sector



(a) Mitsubishi Write to Unallocated Logical Page



(b) Mitsubishi Update to Previously Allocated Logical Page

Figure 3.7: The Mistubishi FTL Scheme

to a given block.

Figure 3.7 shows the general operation of the Mitsubishi scheme where a block is composed of four pages with two pages being used for data and two pages being used as space sectors. Consider Figure 3.7a where the system issues the command *write(7,A)*. The system will determine the logical block number (LBN) using Equation (3.1), with the logical page offset (LPO) within the block being calculated using Equation (3.2).

As the file system has not previously allocated space for the page, a physical block number will be assigned from the logical to physical block mapping table. The data is targeted to be written into the page at offset of 1. Since the page has not been previously allocated, the data A will be written into the location. The logical page number of the data is then written into the out of bounds area for the target page. If the page at offset of 1 in the target block had previously been allocated (Figure 3.7b), the write would not be able to proceed. In this case, the Mitsubishi scheme will select the next available page from the spare area by way of a linear probe and write the data to that page. The logical page number will then be updated in the out of bounds area of the target page. In the case where there are no free pages available in the spare section, the Mistubishi scheme enters a compaction (merge operation) phase where a free block is acquired from the FTL and only the valid pages from a fully utilized page are copied. Once the coping is complete, the old page is abandoned and erased.

While the Mitsubishi scheme reduces the number of erases [KRKC11], the performance gain is proportional to the size of pages in the spare area. Additionally, the overall capacity of the memory store is also reduced due to the strict allocation of spare pages within a given block. Performance limitations are also encountered when updating frequently accessed pages or when mixing hot and cold data. In order to find the next available page, a linear scan is performed to find the next available free spare page which introduces an inconsistency into the write time performance. This leads to longer write times as the page is continually updated. This is due to the system scanning further into the spare area with each successive write. In terms of reads, performance limitations are also encountered especially when attempting to access cold data in a page with a large percentage of hot data. In order to find the current version of the target page, the Mitsubishi scheme starts scanning at the bottom of the spare area and moves backwards towards the front scanning of the out of bounds area for the target logical page number. Based on the update scheme, the newest page will be the first one encountered. If the target page consists of cold data mixed with a high percentage of hot pages, the system will have to scan numerous pages

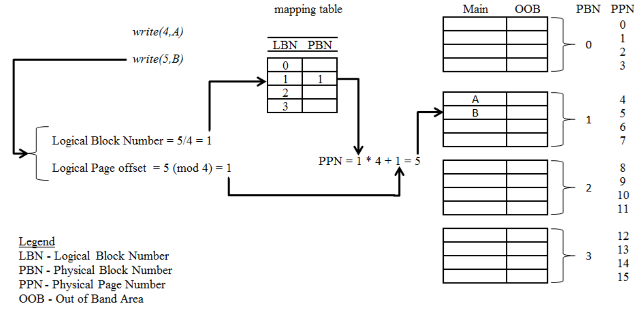
before finding the target page. This leads to inconsistent read performance especially with mixed hot and cold data. Additionally the scheme does not deal with any recovery, data consistency or write normalization issues.

ANAND and FMAX

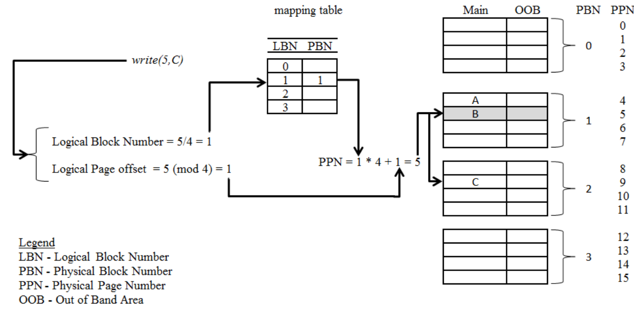
Ban and Hasharon [AB99] introduced two different FTL schemes called ANAND and FMAX which are similar to the log based design presented with Mitsubishi. The key difference with ANAND and FMAX is that logged data is no longer required to be constrained to pages with the same block as required by the Mitsubishi scheme [KRKC11]. With the proposed method, both ANAND and FMAX utilize separate physical blocks for the logging of data with each logical block being mapped to two physical blocks where the first block is the target physical block allocated in the mapping table and the second block is used as the log block to receives pages that have been previously written in the target physical page. The key difference between ANAND and FMAX is how data is written to the log block.

ANAND utilizes the log block as a shadow of the target block which is an exact copy of the logical block but with a different physical block number. As a result of this, ANAND only can support in-place updates. Consider the example for ANAND in Figure 3.8a where the operations *write(4,A)* and *write(5,B)* proceed. Using Equations (3.1) and (3.2) the logical block and page numbers are calculated and the physical target block resolved from the mapping table. As the logical pages are both previously unallocated, the writes proceed unencumbered. In Figure 3.8b when the operation *write(5,C)* attempts to proceed at physical page 5 (in physical block 1, logical page offset 1), it cannot proceed as the physical page has been previously allocated. As a result, the data must be written into the shadow block which has been allocated as physical block 2. The page in the target block is marked as stale (invalid) to indicate that a newer version of the page exists and the update is written into the same logical page offset in the shadow block.

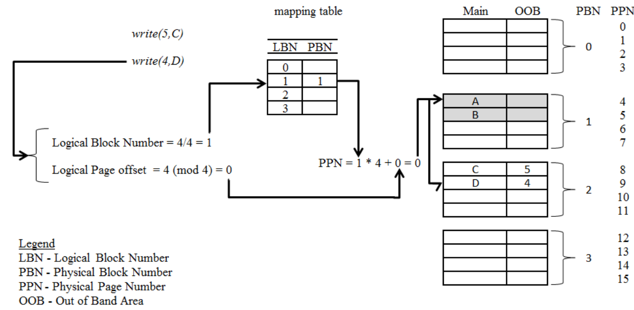
Limitations exist with ANAND in the same fashion as with the Mitsubish FTL scheme. Due to the direct shadow mapping, only 1 update can be performed per page. If a third update was to proceed as *write(5,D)*, a merge operation would occur [KRKC11]. This characteristic will cause a large number of merge operations for sequential writes. A new block would be allocated by the FTL and the live pages from both the target and shadow pages would be merged along with the update into the new block. The FTL would then release the original target and shadow page for erasure. Depending on the types of access patterns in the data set, in addition to



(a) ANAND and FMAX writing to Unallocated Pages



(b) ANAND Update to Previously Allocated Logical Page



(c) FMAX Update to Previously Allocated Logical Page

Figure 3.8: The ANAND and FMAX FTL Scheme

having a mix of hot and cold data within a specific block, a merge operation will occur at every third update leaving to high levels of rewrites and erasures. An advantage with ANAND is that linear scanning of the shadow block is not required as it is using a block mapping scheme where a page can only be placed in one set location per block. This offers faster access times for read operations when compared to previous strategies.

To address the issues of limited logging ability, FMAX utilizes the log block in a different fashion than ANAND. Instead of shadowing the target block, FMAX treats the log block strictly as a linear log of pages similar in fashion to a logging file system [RO92, KNM95] where updates are written in an out-of-place fashion [KRKC11]. For unallocated pages, FMAX proceeds in the same fashion as ANAND as in Figure 3.8 differing in operation only when a write to a previously allocated page must occur. In Figure 3.8c, the operation *write(5,C)* proceeds assuming that the operation *write(4,A)* in Figure 3.8a has already completed. As the target physical page has previously been allocated, the page in the target block is marked as stale and the write proceeds in the log block. Unlike ANAND where writes are restricted to the same logical page offset in the target block, FMAX proceeds to write unrestricted to the next available physical page noting the logical page number in the out-of-bounds area of the page. For the operation *write(4,D)*, the same process occurs and it is subsequently written out-of-place to the next available page in the log block with the logical page number noted in the out-of-bounds area. This technique allows merging operations to be reduced as the merge will now only take place when the log block is full. The rate at which merges occur is now dependent on the block size and better accommodates access patterns having a mix of hot and cold data within a specific block. The limitation to FMAX is on reads; unlike ANAND where data position is deterministic and can be in only one of two places, FMAX allows for numerous rewrites of the same page. As a result when accessing a page the newest version must be found which results in a linear scan through the log block to find the current logical page.

3.4.3 Block Based Logging Schemes

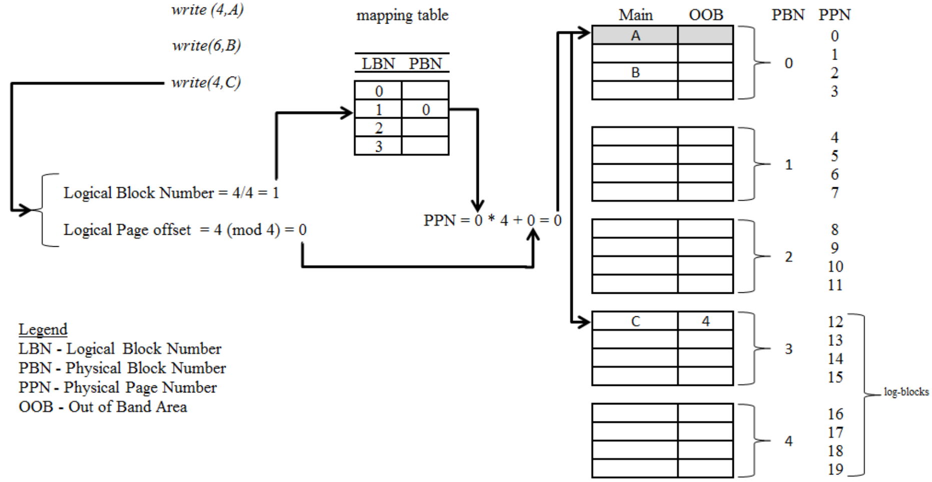
Unlike page based logging schemes where each block has a dedicated logging area either within the same block such as with the Mitsubishi scheme or within a separate block as with FMAX and ANAND, the block based logging schemes extend this concept such that a specific area is reserved for logging operations and that logging blocks are no longer mapped on a one-to-one basis [KRKC11]. That is, many pages share a smaller number of

log blocks in an effort to improve write and merging performance. Sharing similar design concepts with a log based file system [KNM95] updates are written to the log area in a sequential fashion.

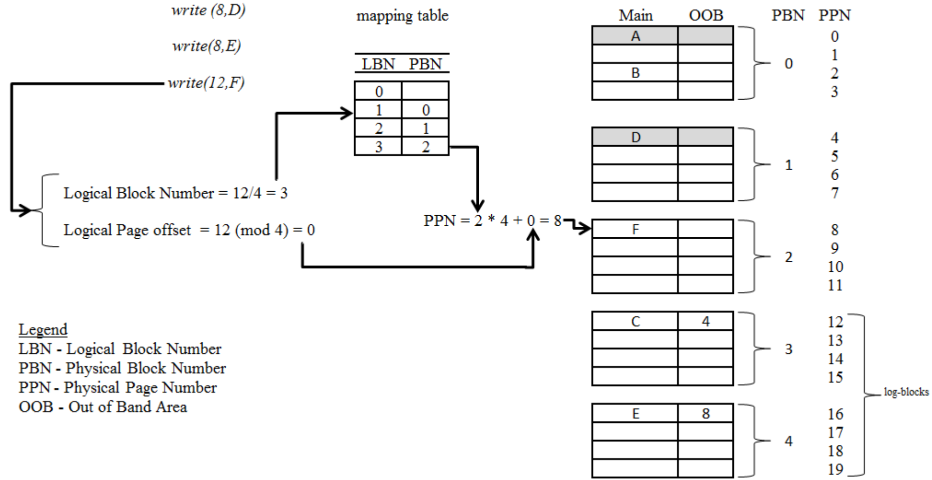
BAST

To deal with the limitations of block mapped flash translation layers, Kim et al. [KKN+02] proposed the seminal strategy *Block Associative Sector Translation (BAST)*. This is a log block type mapping scheme that introduced two different levels of granularity and was primarily targeted at the compact flash market. In addition to the normal block mapping structure, the system maintains a series of log blocks that are managed at the page level in addition to data blocks. The log blocks are used strictly as temporary storage when a page update is required for a block. There is a one to one mapping between log and data blocks where a log block can only be associated with a single data block. Unlike previous strategies though, there is not a one to one relation between data blocks and logs blocks. That is a data block may not have an associated log block depending on the relative hotness of the data compared to the data in the remainder of the system. To avoid the under utilization and overhead of unused log blocks in previous works, BAST limits the number of available log blocks significantly such that all data blocks will share a limited number of log pages [LPC+07]. This ensures that log blocks will continually be in reuse. Kwon et al. [KRKC11] note that due to this approach log blocks must be managed in an efficient fashion. Instead of erasing and rewriting a block when a page update request is made, a log block (which has been previously erased) is allocated and used to store the updated page similar in operation to LFS [RO92]. The out-of-band area is used to store the logical address of each page. Once all log pages for a given data block have been consumed or another block requires the use of a log block and none are available, the system will enter a merge operation where the most current log pages will be merged with their target logical block and re-written into an erased block in flash memory. The system will then choose a new log block for the log to use. The page level mapping is stored in SRAM to allow for efficient operations with respect to speed.

Consider the following example in Figure 3.9 where each block contains 4 pages and the system has two log blocks. The operations *write(4,A)*, *write(6,B)*, *write(4,C)*, *write(8,D)*, *write(8,E)*, and *write(12,F)* are done. The logical block number and page number are calculated using Equations (3.1) and (3.2) respectively using four pages per block. For the first two write operations (Figure 3.9a), the target pages do not contain any data so the



(a) BAST Write to Unallocated and Allocated Logical Page with Available Log Block Space



(b) BAST Write to Unallocated Block without Log Block Space

Figure 3.9: The BAST FTL Scheme

writes proceed directly on the target pages in physical block 0. For the third operation of *write(4,C)* the target page is already allocated so the update must proceed in the log block. The system will allocate physical block 3 as a log block for physical block 0 and proceed to update the data in the first available page. It records the logical page number of the updated page in the OOB area and updates the page mapping table in SRAM for the log. It will also mark the original data page as invalid to indicate that the data has been updated in the log. For the next three operations of *write(8,D)*, *write(8,E)*, and *write(12,F)*, the first two will proceed first by completing *write(8,D)* into physical block 1 as denoted by the block mapping table and then will update *write(8,E)* into the second log block (physical block 4), record the logical page number in the OOB for the logged page and then invalidate the page in the original target block. For the last operation of *write(12,F)*, the system will proceed to update the value into the previously unallocated block 2. If another write operation was to proceed for logical page 8, BAST would attempt to update the target page but detect that a value had previously been written to the page. It would then attempt to write the update value to the log, but no more log blocks are available. At this point, BAST would have to select a victim block that will be merged with the original target block. Once the victim log-block has been merged, the block will be erased along with the old data block and returned to the system as free blocks. The FTL will then select a new log block from the pool of free pages allowing the final write operation to complete.

With BAST, the merge operation can lead to a high level of erase-writes, depending on the number of log blocks and the access patterns of the data. In the worst case, a merge operation will require n page reads and n page writes followed by two block erases. A special case exists that is refereed to as a *switch*, where log pages have been written to the log block in a logically sequential fashion. To reduce the number of read-writes and erases, the log block is simply converted to a data block; this operation only requires that the old block be returned for erasure.

With BAST, instead of storing block mapping information in the out of band or in SRAM, the scheme dedicates a set of blocks in flash that are exclusively used as *map blocks*. The blocks are organized in a page orientated fashion. A two level mapping scheme is used where a map directory is stored in SRAM. Mapping data pages are brought into memory as needed using a caching model. This presents limitations for bandwidth and memory constrained device. Depending on the amount of data and the width of the data path between the memory and processor, a significant number of mapping block data transfer operations are required to read or write

data. Since map blocks cannot be updated in-place due to erase-before-write restrictions of flash memory, map blocks are consumed in a round robin fashion and handled in a similar fashion to data blocks which leads to extensive, non-uniform wear.

Numerous variations on block based logging exist, such as FAST [LPC+07], EAST [KC08], LAST [LSKK08], JFTL [CLP09], and SAFTL [Wu10]. While each variation offers incremental improvements, they are unsuitable for use with resource constrained systems with serial NOR flash. Details on each variation can be found in Appendix A.1.1.

3.4.4 Block Set FTL Schemes

A block set scheme groups adjacent physical blocks into a larger block set where the set will have self contained data and log blocks. The operations are typically based on a log-block scheme [KRKC11] in terms of block utilization and merging but differ in terms of addressing. As they are targeted at large block memory and large capacity devices, they offer the advantage of reducing block mapping sizes due to the decreased granularity. The basic formation of a block set will consist of a fixed number of data blocks with a variable number of log blocks [KRKC11]. While based on a log-block scheme in terms of operation, the techniques use a hybrid technique to improve block set performance in terms of merge costs.

Superblock FTL

One of the earliest block sets schemes was proposed by Kang et al. [KJKL06] where sets of logical blocks are grouped into block sets. The work was extended in [JKJ+10]. Each block set is mapped as a single entity with a block level addressing scheme. A block set is referred to as a super block. Pages inside each super block are addressable at a page level forming a hybrid mapping scheme with three levels. The highest level is the super block map that is maintained in SRAM along with the page global directory (*PGD*). The PGD will contain numerous entries for page middle directories (*PMD*) which in turn point to one of four page tables. Each page table will contain a physical page and block number for the data. While the page level mapping allows pages to be placed freely in any location within the superblock, it presents a long and overly complicated lookup scheme. Kang et al. divide the blocks within a superblock to be either a data block (*D-block*) or an update-block (*U-block*) which forms a contained log block system. The page level mappings for each superblock is stored in the OOB area of pages

within the block. Thus, as new pages are written within a block, the page mapping is simultaneously written into the OOB area. As the OOB area is limited in size, this limits the number of pages that can be included into a superblock.

In terms of operation within the superblock, it performs in a similar fashion to the FAST scheme with the exception of separating data into hot and cold logs to improve merging performance. While the scheme does offer increased performance over BAST and FAST (Appendix A.1.1) for large memories, it does so through the reduction of merging and erase options through the page level granularity available in each superblock. While the superblock scheme allows large memories to be addressed, the operational overhead in terms of SRAM footprint is similar to other block level schemes. Additionally as noted by Kwon et al. [KRKC11], having the mapping table in the OOB area can lead to excessive partial reads and writes for accessing a single page of data which can be detrimental for resource constrained devices. The experimental results offered by the authors suggest that superblocks consisting of 2 to four blocks offered the best results. In the more extensive analysis presented by Jung et al. [JKJ+10], they claim that the performance of their algorithm is in many cases comparable to a page level FTL scheme but in reality, the performance depends on the data patterns. With some patterns, the scheme offers significant savings over block level schemes but with other patterns there is no discernible advantage. While the authors of [KJKL06] and [JKJ+10] have targeted the FTL for large block devices in general purpose computing, they bring to light a key point that FTL performance is strongly dictated by access patterns and data arrangement logically within a block. While they have attempted to produce a general purpose FTL, some FTL schemes are better for certain types of data. In practice, it does not perform as well as page level addressing but does offer some improvements over FAST. Thus, as the number of superblocks will be similar to the number of blocks for a smaller memory, it offers no advantage for memory constrained systems.

3.4.5 State Based FTL Schemes

A state based FTL scheme does not treat a specific block as an atomic entity. Instead, blocks are allowed to transition through various states that offer different levels of functionality to the system as initially presented with EAST (Appendix A.1.1) where a log block initially started out as a shadow copy of the target block allowing only in-place updates. To prevent the log from an early merge (and thus postponing erase operations), the log

block was allowed to change state and support out-of-place updates such that all pages could be utilized before erasure. State based FTL schemes build on this concept and allow blocks to switch state between in-place and out-of-place as well as being tagged as complete. It also allows blocks to be tagged for deletion without having to be immediately erased thus reducing the run time demands on the system.

STAFF

In 2007, Chung and Park [CP07] proposed an improvement to the naive block mapping strategy with STAFF with a focus on reducing block erasures. This was based wholly on their earlier work presented in [CPJK04]. They propose a state machine that encodes the state of each page in the out of band area of a given block but the mapping strategy is fundamentally unchanged from the basic block mapping scheme. Limitations exist as the device must support *Partial-Page Programming* (PPP) (Definition 2.5) where distinct areas of the OOB area can be written in isolation a limited number of times before having to be erased. PPP is not available on all NAND flash memory devices; thus STAFF is limited to specific devices. Their methodology allows the encoding of free, obsolete, modified in-place, complete in-place, and modified out-of-place states. STAFF enumerates the possible block states as:

- *Free* (F) state: an unwritten, erased block
- *Obsolete* (O) state: a block that contains no valid or live data and is a candidate for erasure.
- *Modified In-Place* (M) state: the block has pages that have been written in an in-place fashion. That is, the pages have been written in the log block at the same logical page offset as in the target block. The block may still contain unwritten pages.
- *Complete In-Place* (S) state: the block is completely occupied and all the pages have been written in an in-place fashion.
- *Modified Out-of-Place* (N) state: the block contains at least one page written in an out-of-place fashion. That is, the page has been written at a different logical page offset than its original offset in the target block.

As the state space contains five states, the state of each page can be encoded with three bits. STAFF also allows for a single logical block to

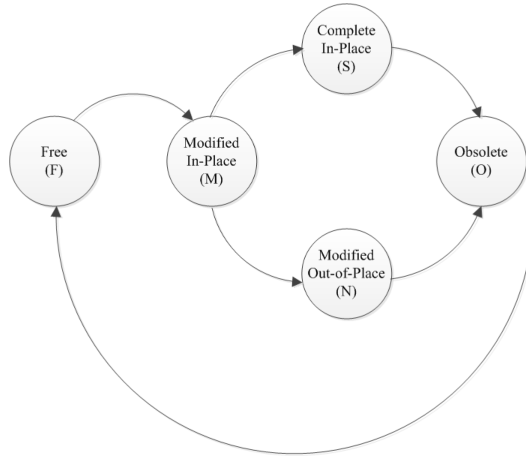


Figure 3.10: The STAFF State Machine

map to two physical blocks to allow for updates in a logging type operation. The state of each page is governed by allowable state transitions shown in Figure 3.10. All blocks start out as being free and can only transition to a modified-in-place state in the case where a single page is updated. When a second page is updated, the block can transition to one of three states. It can transition to a Modified Out-of-Place, if a page write attempt is to a logical page offset that is already occupied. If a write attempt is made to a logical page offset that is unallocated the page will stay as Modified-In-Place. Thirdly, if a write attempt in made to a page at a logical page offset that is unallocated and is the last available page in the block, the write will proceed and the block will transition state to be Complete In-Place.

For blocks that have transitioned to a Modified Out-of-Place, only two possible transitions exist. If a write attempt is made to the block and the block still contains live data, it will stay as Modified Out-of-Place. If all of the pages in a Modified Out-of-Place block have been invalidated, the block will transition to the Obsolete state which means it can be erased immediately. For blocks that have transitioned to a Complete In-Place state, as long as the block contains live data, it will stay in the Complete In-Place state until all pages become invalid at which time it will be transitioned to being Obsolete.

One of the features that they introduce is the concept of partial reprogramming in the out of band area, which is only supported by some flash memory devices. By exploiting this feature, STAFF can change the state of a page without having to re-write an entire page thus effectively reducing

re-write and erase costs. The authors claim that due to this feature, their algorithm offers better overall write performance of more than 5 times through low-cost swapping and merging operations. While based on simulated results, the work highlights the performance improvements than can be realized via partial page programming. A detailed discussion with examples on the operation of STAFF is found in Appendix A.1.2.

Since this is strictly a block mapping scheme, the authors suggest that this strategy is suitable for use on embedded systems due to the smaller size of the mapping tables. Unfortunately, while this may be true for numerous embedded 32-bit processors, the mapping table size is still too large for highly memory constrained 8-bit devices as the mapping table is still stored in SRAM. Kwon et al. [KRKC11] highlight another factor that makes this technique undesirable for memory constrained devices. As the number of N state blocks is unknown and the system maintains a page mapping table in SRAM for each N state block, there is no realistic run time upper bound estimate in the SRAM requirements.

Another significant shortcoming that is not highlighted is the cost of writing into the out-of-band area. It also requires that the memory will support out of band rewrites. This is not available on all memories and is strictly limited to some families of flash. While the concepts presented with this algorithm offer performance improvements, STAFF is not suitable or feasible on many embedded devices utilizing serial NOR flash.

Variations on STAFF have been proposed [CPRH05, CPK11] to offer improved performance with large block devices. The variations use the same state machine proposed by STAFF to track the state of each block but map multiple host operating system sectors to a single large block. Similar to STAFF, the variations are not suited for use with many embedded devices utilizing serial NOR flash. Detailed discussion of the variations are found in Appendix A.1.2.

3.4.6 FTL Improvement Schemes

As addressed by Chung et al. [CPP+09] and Kwon et al. [KRKC11], FTLs should address issues not only of physical to logical page translation, but also recovery, erasure and wear levelling operations. In practice, few works address these holistic requirements, but focus primarily on extending the life of the device by minimizing the number of erase operations to a device through different page and block organization schemes. This section introduces works that offer potential solutions to the holistic requirements without directly addressing page and block organization strategies.

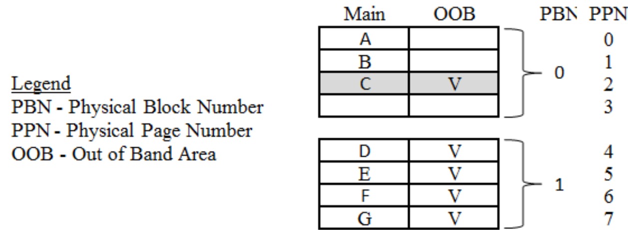


Figure 3.11: PORCE Block Validation

PORCE

One of the significant limitations of most FLT schemes is the failure to address recoverability of the system [CLRL08] in the event of power failure, system corruption or normal power cycling operations. As previously noted by Kwon et at. [KRKC11], recoverability should be addressed as a key requirement for any FTL schemes. This is particularly true in embedded applications [CLRL08] where devices are routinely power cycled under normal operation. Chung et al. present PORCE [CLRL08] as a suitable, generalizable recovery scheme for use with any FTL. However in fact, due to its fundamental principle of operation, it can only be used with FTLs that operate strictly with in-order writes. This eliminates any FTL that relies on invalid but underutilized block re-use to extend the life of the device.

PORCE divides the protection into two different failure modes. The system offers protection during write operations to ensure that data is consistent in flash memory in the event of a failure due to a sudden power removal as well as protection for generalized power off and restart. To ensure consistency, PORCE utilizes a block valid mark to track the status of a block. Previous strategies [CKK10] take a naive view that record the consistency state in the out-of-bounds area for every write operation which are considered to be two discrete write operations per page of data. In the event of a failure, the system scans all pages looking for a valid tag in the corresponding OOB area. If a page is located that does not have a valid tag, it is considered to be potentially inconsistent. In the generalized case, FTL systems offers no strategy for addressing the failure other then by discarding the page in a similar fashion to a database log rollback operation. This technique of continual validation leads to a high system overhead due to the number of writes required as OOB writes require a separate operation.

PORCE divides consistency operations into two different classes of oper-

ations. A failure can occur during normal operations, such as when a system loses power during a write. Alternately, a system can encounter a failure during a merging or cleaning operation where pages are being copied and compacted from one block to another such that the original block can be deleted. To address the first failure mode, PORCE attempts to reduce the overall number of writes by using the valid mark only after the last write of valid data in a block. Its operation assumes that writes will be sequential in nature and consist of more that one page and that each write is self contained and atomic. When writing four pages to a block only the last page will incur the cost of writing the valid tag to the OOB area implying a collective validation. That is, if pages are written in a consecutive fashion, and page n is valid, then it implies that page $n-1$ is also valid. Consider the example in Figure 3.11 where the operations *write(1,A), write(2,B)* and *write(3,C)* occur in a consecutive fashion in physical block 0. After the third write operation a valid mark is placed in the OOB area of physical page 3. In the event of failure, the system will scan all pages looking for the valid tag. If it finds a page with a valid tag as the last page written in a block, then the data is considered to be consistent. If data is found to have been written in a block but no valid tag is found then all but the last page are considered to be valid. Thus, the last page is considered to be inconsistent and the previous page marked as valid. It offers no mechanism for recovery of the inconsistent page under normal operations.

While this does offer a performance improvement over the naive strategy, it relies on two fundamental operational principles. It assumes that all operations (both writes to data blocks and write to update or buffer blocks) will proceed in a sequential fashion. While the authors claim that PORCE can be used as a generalized recovery strategy for most FTLs it is limited only to block level FTLs that do not rely on block reuse in dirty blocks or strategies that only use in-place updates. Secondly, it assumes that data writes will consist of multiple consecutive page writes. While this may be a valid assumption of general purpose computing systems, it may not be true for resource constrained systems where data writes may consist of a single record that must be committed to persistent storage due to the lack of SRAM resources for buffering and to reduce the risk of data loss in the event of a system failure. Consider the write operations in physical block 1 in in Figure 3.11. If *write(4,D), write(5,E), write(6,F)* and *write(7,G)* were isolated write events and thus each require their own valid mark in the OOB area. As a result of this, PORCE would be required to write a valid tag for every write operation, essentially offering no performance advantage over the naive strategy where two writes are required for every data write.

Additionally, it requires PORCE to completely scan every single page during a recovery operation to ensure that it has located the most current valid mark.

The second failure mode that is addressed by PORCE is a failure that occurs during a cleaning process as the result of free pages not being available to the system. In this operation, only valid pages are copied from one block to another, allowing the data block and buffer block to be compacted into a new, single data block with the invalid data and buffer blocks being set for erasure. With the operation, Chung et al. note that there is a significant risk of introducing data inconsistency in the event of a failure mid operation. To address this mode, PORCE utilizes a database REDO style log to track the operations during this phase. The system maintains a separate log where transactions are recorded detailing the cleaning process. At the start of an operation, the type of operation and the target (victim) and destination blocks are recorded in the log along with a log ID. Once the operation completes, a corresponding commit record is written to the log. In the event of a failure, the system will scan the log looking for log records and the corresponding commits. If a record is found to exist without a corresponding commit, the system is able to redo the operation. While this appears to be a reasonable strategy, a significant challenge exists with this operation. The cleaning operation involves multiple copies and erases. Based on the algorithm presented, uncommitted operations in the log are completely redone and do not consider the state of individual pages in blocks. If the failure occurs after the target block has been erased, but before the commit is written to the log, the system will lose referential integrity to the data. Upon restart, the target blocks that are involved in the transaction may not contain valid data due to erasure but still be used to update the destination block, thus producing inconsistent data.

In terms of overall performance, Chung et al. present a performance analysis utilizing STAFF. The performance of the recovery algorithm is compared to the naive strategy and overall does perform better but does increase the overall number of erase/write operations when compared to STAFF without any recovery option. Additionally, it appears to have highly variable performance depending on the type of data and access patterns. Operations without large sequential writes will encounter significant additional overhead. While a potentially plausible strategy for a general purpose computing system where large consecutive pages of data may be accessed, it is not suitable for use on resource constrained systems due to the increased number of operations as well as the risk of data inconsistency introduced through the logging operations.

Reuse-Aware NAND FTL

Wang et al. [WLW+10] have developed a reuse-aware NAND flash translation layer (RNFTL) which focuses on minimizing the number of merge operations through strategic management of free pages in data blocks. The scheme uses the similar mapping algorithm presented with FMAX and ANAND [KRKC11] where each active or target block can be associated with a log (replacement) block such that updates can occur without having to immediately update the original block. The proposed strategy attempts to minimize the merging of active blocks that contain a large number of free pages. Unlike other schemes, after a merge of an active (or primary) block with its associated replacement block, the target block is not immediately sent for erasure after being invalidated. The algorithm attempts to reduce erase counts and extend the lifetime of NAND flash by applying a reuse strategy in the flash translation by examining the number of free pages left in the block and attempting to maximize the block utilization before the block is erased. The authors claim that RNFTL improves utilization and minimizes erase counts compared to previous methods.

The primary difference between RNFTL and other logging strategies is that invalid blocks are put on a preserved block list such that they can be used as log blocks for updates. A block may be sent for erasure while there are still free pages available which is dependent on data access patterns. Some access patterns will utilize a small percent of the block leaving the majority of the block unused. A block that is only partially filled and that can be reused for logging is referred to as a *dirty block*. Blocks that are no longer used but are not completely full are transferred into a preserved block list which contains a list of dirty blocks if the number of free pages in a block is above a preset threshold. A block with a total number of free pages below the threshold is sent for erasure immediately without reuse. This ensures that a dirty block with a significant number of free pages can be reused and fully utilized before it is selected for erasure thus delaying erase operations. This introduces significant overhead in terms of data management and also limits the classes of NAND memories that can be supported. This strategy is reliant on the fact that NAND memories can support out of place updates. While this is true for older small block architectures, this is not true for new large block devices which enforce the constraint that data must be written in a sequential fashion per block.

Due to the reuse strategy, the scheme allows for high levels of reuse in terms of block management by focusing primarily on reducing erase operations but it offers nothing new in terms of the flash translation layer

mapping and table management. The authors claim that wear levelling is explicitly improved, but it appears that this is generally due to the decrease in page erases. They are unclear on how this is actually accomplished in terms of wear levelling strategies. In fact, the correlation between the performance of the wear leveller and the total number of erase operations is a commonly addressed theme in previous works. If the reuse threshold is not set correctly, performance of the scheme is not better than a hybrid scheme [WLW+10] and functionally aligns with the FMAX strategy. If the threshold parameter is tuned correctly, RNFTL does present a significant advantage over previous schemes as it offers better space utilization. Blocks are fully utilized (to a given threshold) before they are sent for erasure which in turn decreases overall number of block erasures. Additionally, this strategy does not differentiate between write patterns for hot and cold data which impacts the overall performance. A page with a large amount of cold data will not be allowed to be reused thus will not increase the overall performance of the algorithm.

RNFTL shows good performance with appropriate parameters due to extensive block reuse, but its suitability for devices is limited. As with FMAX, page writes to a block may occur in an out of order fashion. This excludes it for use on large block devices. Through the reuse of dirty blocks being used as replacement blocks (log type blocks) wear is more uniformly spread out across the device as well as reducing the number of erase cycles required. The consequences of this are that the erase operations will run less frequently. This behaviour leads to improved wear levelling but not necessarily by design.

RNFTL offers performance benefits similar to previous strategies but the state of blocks need to be explicitly tracked. For many constrained systems this additional overhead makes it an unattractive choice. What is not considered is that the duty block list must be maintained in a specific space in flash memory. This will lead to non-uniform wear if dirty page operation rate is very high or very low compared to the data rates in the main flash section. This partitioning violates the goal of wear levelling which is to uniformly spread wear across the device. This also can generate problems for devices where adjacent block erase thresholds need to be maintained within a physical bound.

A significant limitation of the scheme that is not discussed is the read performance. With other strategies that use a fully associative strategy, pages will be updated to the replacement block in a strictly increasing fashion. As a result, the most current version of a specific page can be found by scanning backwards from the latest entry. When it encounters the logical

page that is being search for, it will be the most current version. With RNFTL, this strategy cannot be employed as live pages will be intermingled with previously invalidated pages. To further confound this, previous versions of the same logical page that is being read can exist anywhere in the dirty block. As a result, every single page in the block will have to be checked to see which one is the most current. Additionally, the work does not present a strategy for being able to resolve page versions. As a result of this behaviour, the read performance of the algorithm is poor compared to other strategies.

While the authors propose a suitable strategy for reducing the number of erase cycles and limiting wear across the device, they have not changed or improved the fundamental flash translation layer block mapping algorithm. It offers no performance increase in terms of address translation and increases look-up complexity when attempting to locate pages in the replacement blocks during reads. Regardless of any improvements, the FTL mapping table is still maintained in SRAM. Additionally, it is targeted for NAND flash and does not consider the cost of data movement between memory devices and SRAM buffers. While this is an improvement for high endurance applications, its complexity offers little value for embedded systems for storage capacity and data rates are orders of magnitude lower.

Janus FTL

Janus-FTL [KKC+10] is a hybrid page/block mount solution offering mixed mode addressing similar to SAFTL [Wu10]. The major contribution is that the FTL can adapt to specific types of work load and is particularly targeted and suited for solid-state hard drives. The work attempts to find a balance between erase and merge operations such that the number of units for a given workload is minimized The FTL algorithm analyses the utilization of free and invalid space to decide between erase and merge operations, allowing the use of free pages in invalid blocks log type operations to improve performance in a similarly fashion to the earlier works of RNFTL [WLW+10].

The design of Janus-FTL relies on page mappings to be kept in SRAM for recovery and performance which limits its suitability for resource constrained systems. The authors compare the performance of Janus-FTL against other strategies [KKC+10] and was found to have mixed performance results.

ShiftFlash

ShiftFlash [HZW11] is designed to introduce the concept of continuous data protection for solid-state hard drives. Its overall architecture is not

feasible for small devices due to its overall size and complexity. It offers nothing new in terms of flash translation strategies but does allow for high-level rollback through the use of data snapshot and backups. It stores a mapping to superseded data as a set of tuples in SRAM. As a result, it enforces sequential writes to prevent overwriting of snapshotted data.

It utilizes a user-defined protection window in which writes are guaranteed to be protected from corruption or loss. This significantly impacts how garbage collection functions as invalid pages in the protection window cannot be reclaimed due to the fact that they may contain data under protection. This results in a lower number of free blocks being available to the system as well as having to manage multiple garbage collection thresholds.

The most significant change from other FTL strategies is with the operation of the garbage collector. When triggered, the garbage collection forces a merging of superseded pages from the protection window and produces a new system snap shot. This is a complex operation as a protection window may have multiple copies of a single superseded page. Thus, the garbage collector must scan all superseded versions to find the most current one.

ShiftFlash relies on inter-flash memory chip wear levelling and garbage collection strategy to realize performance increases as its target platform is solid-state drives. This results in extensive data transfer between memory devices. This approach is not feasible on embedded systems due to the loss of memory and a highly constrained data bus.

The operation of the garbage collector is used to trigger the advance of the protection window. This is similar to protection offered by the FlaReFS [FL11], but with significantly more overhead and complexity. Additionally it only is designed to work with multi-chip NAND flash devices and is not necessarily compatible with all FTLs due to its write patterns.

While ShiftFlash offers a suitable continual protection for enterprise level systems, no analysis with respect to flash translation performance, utilization or garbage collection performance is done. The work does not address fault tolerance and recovery.

3.4.7 Suitability of FTL Schemes for Serial NOR Memories

The development of flash translations layers has primarily focused on improved utilization of NAND in terms of overall lifetime. The lifetime of a flash memory device is directly linked to the number of erase cycles. Numerous FTL strategies have been suggested to address this concern. Appendix A.2 summarizes the presented FTLs and highlights the type of mapping scheme used for each, as well as the key contribution of each

algorithm in addition to any special requirements. All FTLs utilize one of three underlying mapping strategies; a page mapping scheme, block mapping scheme or a hybrid scheme. Many of the FTLs presented are a variation on a common theme, with the differences focusing on how to extend the life of the device through lowered erase counts and improved page/block write management. Of the work examined, none focus on the cost of data transfer between the host and device which is a key design parameter for serial NOR devices. While the FTLs presented are targeted at NAND flash, many would theoretically work with serial NOR flash but in practice are infeasible as they produce high levels of overhead for constrained systems. Ideally, an FTL for an embedded system utilizing serial NOR flash must minimize the amount of data transferred across the bus. The volume of data is directly related to energy consumption, a key parameter in resource constrained systems.

As a result of the unreliable nature of embedded systems, the FTL must offer fault tolerance and recoverability guarantees. This is not offered by the majority of the FTLs. Additionally, many of the FTLs assume that a large amount of SRAM is available as they are designed for general purpose computers. The FTL for a constrained device must work with a minimum amount of SRAM due the physical limitations of many devices, while still offering reasonable performance. While many of the FTLs offer significant insight into design issues with flash memories, none are suitable for use with serial NOR flash on constrained devices.

3.5 Open Research

With the pervasive nature of embedded devices, the desire to collect and analyze data is increasing. While different solid state strategies exist, flash based memory technologies are the best candidate until other solid state memory technologies become more viable in terms of cost, energy and capacity requirements. While consumer devices have the ability to run a variety of file and operating systems, most resource constrained devices do not have this luxury, being limited by both memory and power requirements. Additionally, they are typically focused on a single set of tasks, such as temporal data collection or process operations. As a result, complex file systems are not required. For many applications, storage will require only create, append, read, and delete operations. Many embedded applications will use their own data storage strategy and only require an abstraction layer to the lowest level of hardware.

Flash memory presents significant technical challenges due to the lack of

in-place updates in addition to only providing a page or block level interface to the application. Utilizing a flash translation layer, a developer can easily produce a system that will allow for the low level management of data without the complexity of a large file system.

While significant research has been undertaken in the area of flash translation layers, the majority of work has focused on large systems utilizing NAND flash. The core mapping algorithms for page mapping remain relatively unchanged with efforts focusing on reducing the number of erase operations which in turn extends the life of the device. Many works present small variations on a common theme focusing on reducing costly erase operations and extending the life of the device but do not consider the cost of data transfer and management.

For resource constrained devices, the current FTL systems do not offer truly viable candidates. While targeted at NAND devices, many pin constrained devices cannot use these devices as they are physically pin limited. They tend to favour serials devices such as Adesto's serial Dataflash due to the low pin count interface. Additionally, the majority of serial memories are NOR flash and offer different read and write constraints compared to NAND flash which prevents the optimal use of many FTL strategies. Current implementations also require large SRAM footprints, making the majority of them unsuitable for memory constrained systems. An oversight with the current strategies is that no consideration or analysis is made in terms of the amount of data transferred between the host and memory device. For serial accessed devices, this is a critical consideration as energy consumption is incurred with the movement of data across the serial bus as there is a correlation between energy consumption and data transfer.

With current FTL strategies, little attention is given to the storage and size of mapping tables. This presents challenges in terms of consistency, resource availability and energy cost. For page level strategies, the available SRAM on many resource constrained devices is not large enough to maintain the map leading to extensive page transfer between host and device through caching strategies. To complicate this, the mapping tables will continually need to be flushed to flash memory to ensure consistency in the event of a failure. This introduces significant overhead on the serial bus as well as being a large energy consumption source. While block or block-sets produce smaller mapping tables, more data must be moved between host and device due to the increased block size which increases bus utilization and risk of data loss and is not suitable for small data records.

One common attribute missing in the collective works is the movement of data into and out of memory buffers. While current FTL strategies make

no concession for this, efficient and strategic utilization of buffers can be explored to reduce data transfer between host and device [FL11]. Another key issue that is not considered is the consistency of systems in the event of a failure. This is present on both the data and metadata level. Systems that use block level association present risks for data corruption. This is due to the number of page writes that must occur during a block move and hence increases the risk of a failure due to data corruption as a result of a fault during a write operation.

For memory and resource constrained devices, FTL requirements are different than with general purpose computing systems. A balance must be stuck between SRAM utilization, data transfer and consistency. An ideal system would have low energy consumption as well as minimizing data movement while ensuring a high degree of consistency. Low data movement not only reduces energy consumption but also reduces the risk of data loss and system inconsistency. For a system to have a small SRAM footprint, the minimum data block size must be larger. For block level schemes, this increases the complexity of data management and well as energy costs. Additionally, due to the nature of environments where embedded systems are being used, an FTL should offer a degree of data consistency and recoverability as system faults may occur producing a potential source of data loss. In situations such as environmental monitoring applications, data consistency during collection is critical. Events may be singular in nature and the loss of data unacceptable. To balance all required aspects for memory and resource constrained systems, the FTL must have a small unit of addressability which reduces the risk of data loss and minimizes data transfer. While page mapped FTLs offer this solution, currently the page mappings are too large to be of use.

With NOR flash being the most common choice of flash memory for embedded systems, few FTL choices exist. Atmel has produced an FTL for the serial NOR Dataflash but it is incompatible with 8-bit processors. It has a large SRAM requirement as is only available as a pre-complied library for specific Atmel 32-bit devices making it infeasible for use across multiple platforms. No work to date offers an ideal solution that balances all of these aspects in a holistic FTL system nor considers memory device specific algorithms to take advantage of architectural features that can offer performance gains that is cross platform and addresses the challenges in the 8-bit embedded space.

Chapter 4

Write Strategies for Serial NOR Flash

> If the person you are talking to doesn't appear to be listening, be patient. It may simply be that he has a small piece of fluff in his ear.
>
> A.A. Milne - Winnie-the-Pooh (1882 - 1956)

Embedded devices need to be able to store and process data. The Internet of Things (IOT) involves devices such as wireless sensor networks and mobile computing platforms interacting with each other [GIMA10]. It is anticipated that by the end of this decade there will be between 30 and 50 billion devices participating in the IOT [Wit13]. One of the key aspects of the Internet of Things is the sensing and sharing of data and environmental parameters automatically in numerous domains [AIM10].

IOT vendors such as Cisco anticipate the direct sharing of data between devices driving the need for local storage and processing [Eva01]. Devices such as the Telos, Btnode, MicaZ [BPC+07] platforms have been previously used as research and development platforms. Recently, the Arduino [Sev14] family of devices has driven low cost development and exploration. These devices are typically small 8-bit devices [ASSC02] with power and persistent storage constraints as well as minimal memory (often less than kB SRAM) [DNH04]. Energy availability and secondary storage are key factors.

While NAND Flash has found numerous applications such as mobile handsets and primary storage solutions, NOR flash is still the primary choice for use in embedded applications [ZSI11] primary due to its low read latencies and data integrity. It maintained market dominance over NAND flash until 2005 by measure of market revenue [AA11]. The adoption of NAND flash did not move at the same rate due to the significantly more complex write patterns due to the block structure of the device [SCKS08]. This chapter examines how

write strategies optimized for serial NOR Dataflash can significantly improve device performance including fewer page erases, faster write operations, and less energy consumed. The techniques examined can be used to improve data consistency, extend the field lifetime of memory devices, and simplify write techniques for resource constrained devices. The technique applies to many applications and devices. In this work, the target devices are 8-bit processors which are commonly used due to low cost and complexity [Mur15] and are well-suited for many data collection and logging applications. These applications are still well suited to the 8-bit processor which is being driven by a growing number of IoT systems utilizing this technology [Mur15]. The 8-bit architecture is still the most commonly used device today accounting for almost 40% of all microcontroller device sales in 2014 [TBHR15].

One of the largest challenges for resource constrained devices is the lack of SRAM for temporary storage of data. Devices such as the Arduino utilize SD cards for secondary storage, but SD cards required too much SRAM to use with resource constrained devices.

Embedded devices have limited SRAM and use either NAND or serial NOR flash for persistent storage. NAND flash has faster performance and larger capacity than NOR flash but requires a higher pin count and more complex data management strategies. NOR flash has simpler management requirements and would be a more useful technology if its write performance was more comparable to NAND flash. For systems that are unable to use an FTL, serial NOR Dataflash offers the ability to erase single pages before writing, but at a much higher cost than a block erase. It does provide an attractive solution as no additional management is required. The worst case scenario is that a page needs to be erased for every operation to satisfy the erase-before-write constraint if the device is unable to maintain an FTL type algorithm. The drawbacks are that the operations are expensive both in terms of time and energy.

In many of the smallest applications, serial NOR flash memory is used. A commonly found alternate strategy is serial NOR Dataflash, a NOR flash based technology with additional buffers to facilitate the movement and storage of data. While it offers advantages over SD cards due to this feature, users of the device are directly required to manage write and erase constraints commonly found with flash memory.

This chapter presents serial NOR Dataflash optimized writing strategies that greatly improve its performance and usability for embedded devices.

4.1 Write Strategies for Improved Performance with Serial NOR Dataflash

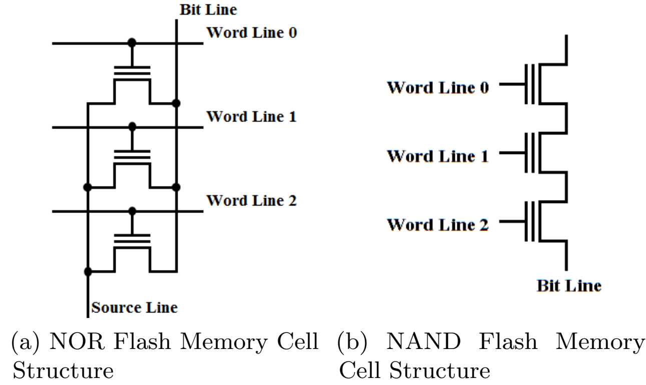
Understanding NOR flash memory technology allows for write optimizations. Flash memory technology is based on the floating gate MOSFET which has a similar architecture to the MOSFET used in SRAM. The floating gate MOSFET [KS67] can encode a persistent state for a long period of time without the requirement that it be continually powered.

As discussed in Section 2.2, the floating gate of the MOSFET is used as an electron trap to encode information. In the erased state, flash memory is set to a logical '1' via the absence of electrons in the floating gate. The lack of electrons allows for the establishment of an electric field and generation of a conduction channel. When programmed, electrons are injected into the floating gate which prevents the formation of an electric field and thus the formation of a conduction channel. As noted previously, the method by which electrons are injected into the floating gate and the quantity differ depending on the type of flash memory.

NAND flash uses *Fowler Nordheim* (FN) tunnelling for both erase and write operations whereas NOR only uses FN tunnelling for erase operations. For write operations, *channel hot electron* (CHE) injection is used to inject electrons into the floating gate. This operation is self limiting in such that the injection operation will stop when sufficient charge has built up proportional to the strength of the electric field being applied.

Another difference is the configuration of memory cells. With the NOR architecture (Figure 4.1a) each element in the matrix has its control gate connected to a word line and bit line connected to the drain [BCMV03]. This allows the matrix to address a single element in the memory array without disturbing any other element. NAND flash shares a similar configuration in terms of the word line which is used to activate the control gate of the element or elements being read. The single largest difference in the architecture between NAND and NOR is how the bit line is connected. Unlike in NOR memory, the source and drains of the memory cells in NAND flash are linked together (Figure 4.1b) in a daisy chain fashion [BCMV03] which can lead to disruption of neighbouring cells.

These observations leads to a unique opportunity with NOR flash that is not possible with NAND flash. When a page is erased, each cell will encode a logical '1' (lack of charge). Unlike NAND flash where charge levels during programming are continually controlled as well as suffering from neighbour write disruptions, when a NOR memory cell is written (set to logical '0'),



(a) NOR Flash Memory Cell Structure

(b) NAND Flash Memory Cell Structure

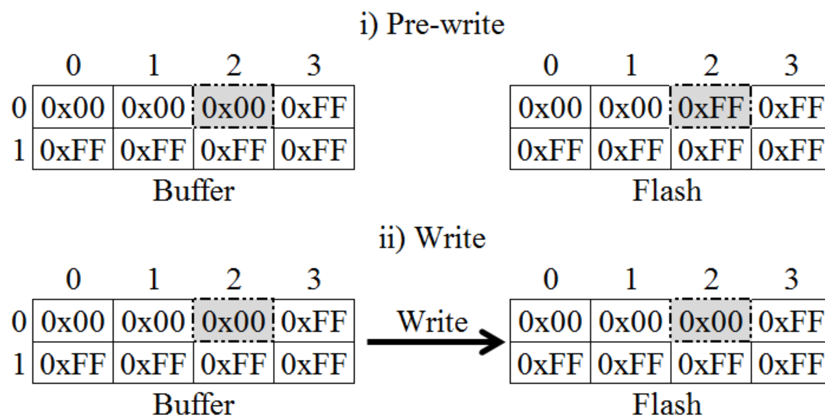
Figure 4.1: Memory Cell alignment for NOR and NAND Flash

charge is allowed to accumulated in the floating gate via CHE injection. If an attempt is made to write a logical '0' to the cell again, the existing build of charge will prevent any additional charge entering the floating gate due to the self limiting properties of CHE injection. It is hypothesized that this unique observation allows for the re-writing of memory cells under specific conditions which can be exploited to increase device performance.

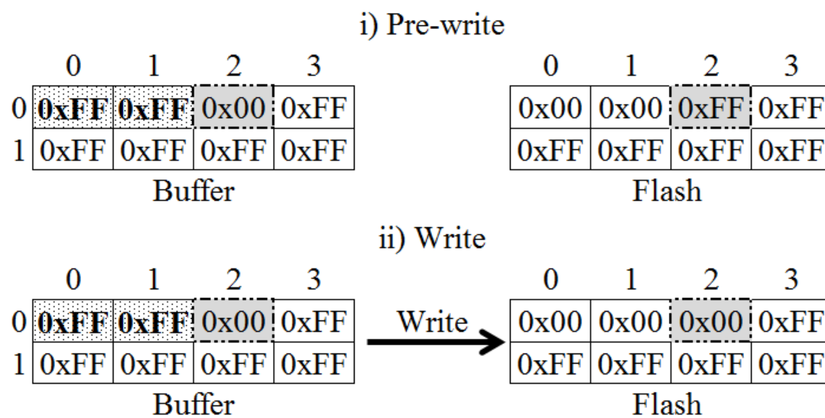
4.1.1 Write Operations for Serial NOR Flash

In developing the understanding of the structure of NOR flash, it is conjectured that a NOR cell may be re-written as long as the transition is from logical '1' to logical '0' without disturbing neighbouring cells due to the internal structure of NOR flash (Figure 4.1a) for page accessible serial NOR Dataflash.

Consider the following state transitions for an erase NOR memory cell where CS_{erased} is the state of the cell (logical '1') after it is erased and is considered to be the normal state before any write operation. The state of allowable transitions is shown in Figure 4.3. Under normal conditions where the cell is being written, the allowable bit transitions can be represented as an AND operation between the state of the current cell state and the data being encoded as shown in Table 4.1a. Based on the previous conjecture on allowable cell transitions, the complete set of operations can be fully expressed as an AND logic function as shown in Table 4.1b where $CS_{initial}$ is the initial state of any cell, and $CS_{after\,write}$ is the final state. It is observed based on the table, that if the initial state of the cell is a logic '1', then the cell can be transitioned to either a logical '1' or logical '0' whereas if the cell



(a) Overwritting



(b) Masked Overwritting

Figure 4.2: Overwriting strategies for data movement from SRAM buffer to flash page for Serial NOR Dataflash

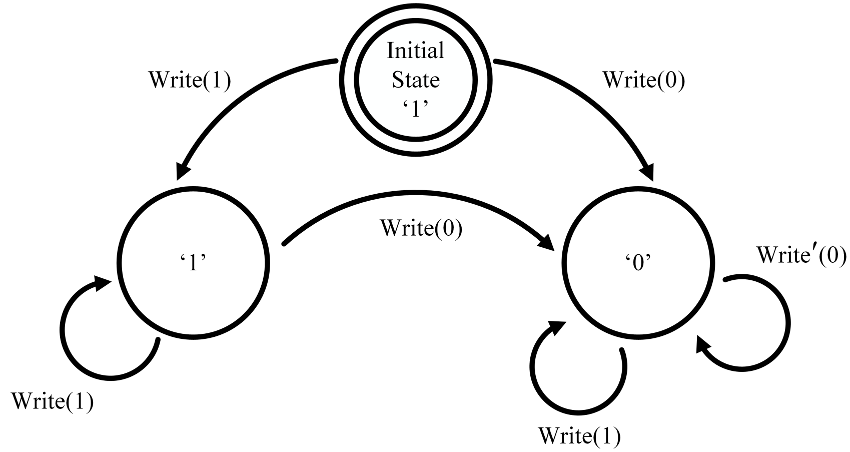


Figure 4.3: Memory Cell Write Transition States for Serial NOR Dataflash

Table 4.1: Write State Truth Tables for a NOR Memory Cell.

(a) Writes to Erased NOR Cell

CS_{erased}	Data	$CS_{after\ write}$
1	1	1
1	0	1

(b) Writes to an Any State NOR Cell

$CS_{initial}$	Data	$CS_{after\ write}$
1	1	1
1	0	0
0	1	0
0	0	0

to be written is a logical '0' regardless of the data being written, the final state will be unchanged.

Under normal operation, data is loaded into the SRAM buffer and then written to a previously erased page. In the case of append type operations, data is held in the SRAM buffer, new data appended, the target page erased and then rewritten. In this type of operation where new data is being written to previously erased locations with existing data being unchanged, it is hypothesized that the buffer can be rewritten back to the same pages without having to occur an addition erase operation. Figure 4.2a demonstrates this *overwriting* operation for a simplified example where the page and buffer size are 8 bytes. In this example, locations 0 and 1 have previously been written by the buffer to flash. In overwriting, new data is appended to location 2 (indicated by dashed outline) and then the buffer is written back to the same flash page. As the contents of the previously written locations are unchanged,

only new locations will be modified.

A limiting factor in the overwriting operation is that a write to a previously written location will still induce CHE oxide degradation. It is hypothesized that this can be minimized by changing write patterns to previously written cells. When a logical '1' is transitioned to logical '0', charge is injected into the floating gate. By examining the state transition diagram (Figure 4.3), independent of the current cell state, a write with a logical '1' will transition back to the current state. In the case where the cell is already a logical '0' and a logical '1' write is attempted, no CHE injection will occur leaving the cell in its previous state. This transition can only physically occur with FN tunnelling that is developed during an erase cycle. Figure 4.2b demonstrates this *masked overwriting* operation for a simplified example where the page and buffer size are 8 bytes. In this example, locations 0 and 1 have previously been written by the buffer to flash. In masked overwriting, new data is appended to location 2 (indicated by dashed outline) and previous written locations in the buffer are masked to 0xFF (indicated in bold). The buffer is written back to the same flash page. The contents of the previously written locations are unchanged due to the forbidden 0 to 1 transition described, modifying only the unmasked, unwritten area.

For data storage operations this is a desirable operation as it allows new data to be appended to a page without having to occur additional erase/write cycles as the page can be rewritten in place. It is estimated that the savings in terms of page erases and energy consumption is significant compared to write operations that are written to fresh pages for every commit. This will offer record level consistency without having to occur high levels of erase operations, and presents the opportunity for bit vector mutation in place.

4.2 Hypothesis about Rewrites

Given this understanding of the fundamental NOR flash architecture, the following hypotheses are tested:

Hypothesis 1: A serial NOR Dataflash page can be overwritten in place with no data loss as long as the only bit transitions are from 1 to 0 or 0 to 0.

Hypothesis 2: The page write time in serial NOR Dataflash is proportional to the number of bit transitions from 1 to 0 written.

Hypothesis 3: When overwriting a serial NOR Dataflash page, utilizing a bit mask (masked overwriting) of ones for all bits applied to previously written data in a page will improve performance while maintaining data correctness.

4.3 Experimental Results

The experiments were run on a serial NOR Dataflash memory device (AT45DB161E from Adesto Technologies [Ade15]).

To validate Hypothesis 1, memory pages were continually re-written with page data that contained an increasing number of zeros. The first write had zero zeros, the second had one (in the first bit), the third write had zeros in the first two bits, and so on. Each page consists of 512 bytes (4096 bits). Each page on the device was written 4096 times with an increasing number of zeros. This was repeated for each device page for a total memory device writes of 16 777 216 times without a single bit error.

These results validate Hypothesis 1 that it is possible to overwrite a NOR flash page without data loss with the constraint that all bit transitions are from 1 to 0 or 0 to 0.

The second hypothesis examines the correlation between write patterns and write times. It suggests that there is a correlation between the number of bit transitions (1 to 0) in a write operation and the amount of time to complete the buffer write to the main flash page. In this experiment, the number of 0's in the SRAM was increased after each write to the same page in memory. Two test conditions were used to examine if specific patterns impacted write times: a baseline where the target page was erased before each write and a test using overwriting (Figure 4.2a). The test conditions were evaluated with an increasing number of zeros starting with byte zero incrementally written to a flash page. Two SRAM buffers were used on the serial NOR Dataflash. One buffer was used for the masking test condition and the second buffer was used to maintain a mirrored state of data in the flash page for validation of contents. The test was repeated across multiple device pages. Least squares analysis was performed on each test condition. The results and number of observations are presented in Table 4.2 which shows a strong correlation between the number of 1's written with respect to time.

Figure 4.4 shows the time in milliseconds to complete writing the nth byte under a given test. For the baseline test (Figure 4.4: Erase Before Write), the flash page was erased before each write. The graph shows that the time of the device to complete the write of the buffer to the target page in flash memory is directly related to the number of 0 values being written. This is supported by the high degree of correlation (Table 4.2) between 1 to 0 transitions per page and write times. The confidence interval for each data set is also shown on Figure 4.5, but the interval is imperceivable as it is very small.

To examine the cost per bit with respect to time, an increasing number of zeros starting with byte zero were incrementally written to a series of pages. New data was transferred to the SRAM buffer. As a baseline, the buffer was written to a previously erased page. This was repeated with the number of transitioned bytes (from one to zero) increasing by one on every write. After each write, the flash page was erased with the buffer maintaining state. The results (Figure 4.4) show the time of the device to complete the write of the buffer to the target page in flash memory. Results demonstrate a high degree of correlation and monotonically increasing nature between the number of write transitions per page and write times.

For the overwriting test (Figure 4.4: Overwrite without Mask), the flash page was only erased before the initial write. Each subsequent write to the page in flash used the overwriting technique (Figure 4.2a) where the SRAM buffer maintains consistency with the values written to flash and updates a single byte in a previously unwritten area of the target flash page. The results demonstrate a similar linear relationship and high degree of correlation (Table 4.2), but faster write times were observed. This is due to faster equalization times for CHE injection when writing to previously written location as cells already contain excess electrons.

From the high degree of correlation observed, it is the act of attempting to modify the floating gate memory cell that is the dominant factor in writing, but the initial state of the cell in memory and buffer will impact write times.

Hypothesis 3 was that single byte write times could be increased through strategic management of previously written data in the buffer (Figure 4.4: Overwrite with Mask). Data within a page that is not actively being written is masked with a high logic state (Figure 4.2b). From Hypothesis 2, the write time is linearly related to the total number of 0 bits in the buffer being written. Thus, when writing a page to the device, it may be valuable to have all bits in high logic state (1) except for the bits being written (see Figure 4.2b). This not only decreases page write times but will not degrade existing cells through additional CHE injection.

As in the previous test, one serial NOR Dataflash SRAM buffer was used for consistency and validation checking of the flash page being written. It maintained the true state of what should appear in the flash page after each write to check the correctness of the masked overwrite. The second buffer implemented the overwriting technique (Figure 4.2b). The same test operations were used as in the previous test, except during each write, the location of the byte being written was unmasked and updated in the second buffer, written to the flash page and then re-masked. Least squares analysis was performed on each test condition. The results of each test and number

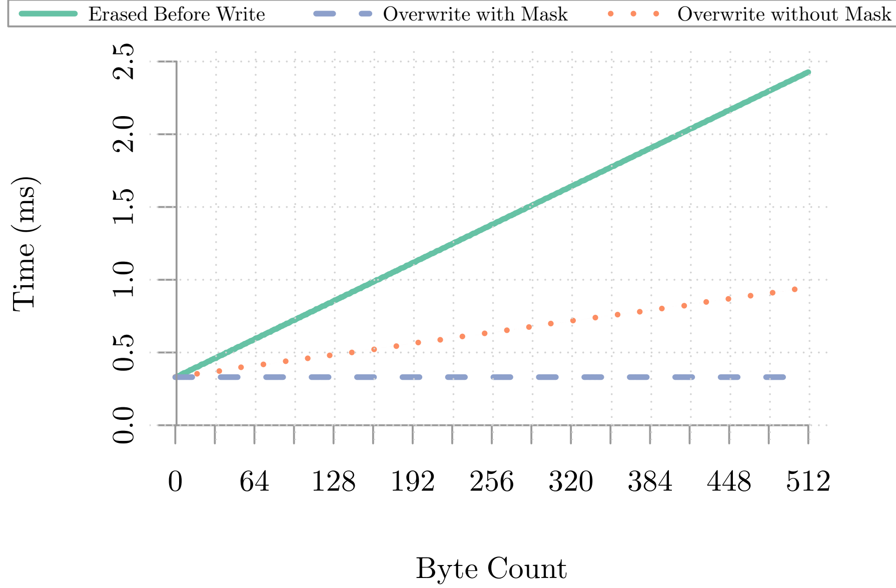
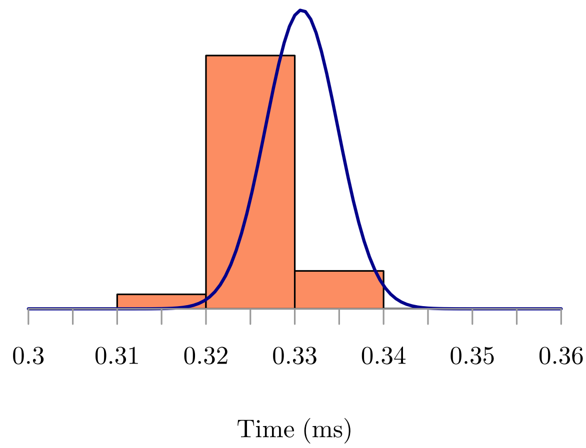


Figure 4.4: A timing comparison of overwriting techniques for serial NOR Dataflash.

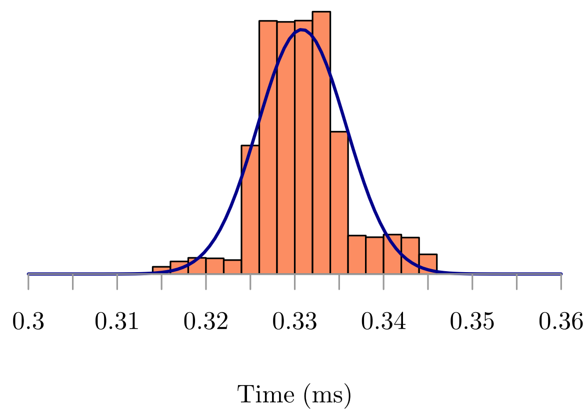
Table 4.2: Least Squares Coefficients for Writing Techniques

Write Strategies	Erase Before Write	Overwrite	Mask Overwrite
Slope	$4.099{\times}10^{-3***}$	$1.201{\times}10^{-3***}$	$2.016{\times}10^{-7***}$
	$(3.793{\times}10^{-7})$	$(5.647{\times}10^{-7})$	(3.459e-8)
Intercept	$3.287{\times}10^{-1***}$	$3.311{\times}10^{-1***}$	$3.308{\times}10^{-1***}$
	$(1.122{\times}10^{-4})$	$(\ 1.670{\times}10^{-4})$	$(1.023\text{e}{\times}10^{-5})$
Observations	2 097 152	2 097 152	2 097 152

Standard errors in parentheses.
(***) indicates significance at the p=0.01 level.
Each column contains regression coefficients for the linear model for a given write strategy.



(a) Measured Single Byte Write Times



(b) Single Byte Write Times with Jitter

Figure 4.5: Distribution of Single Byte Write Times with Masked Overwritting.

Table 4.3: Timing Comparison of Overwriting and Erase-Before-Write Operations

Method	Number of Erase Operations	Time for Erase Operations (ms)	Time for Writes (ms)	Total Cumulative Time (ms)
Erase-Before-Write	512	6144	861.38	7005.38
Masked Overwriting	1	12	190.23	202.23

of observations are presented in Table 4.2.

In examining the results in (Figure 4.4: Overwrite with Mask), it was found that the write times are not proportional to the actual number of 0's in the main flash page but to the number of 0's in the buffer being written. Unlike the baseline condition or overwriting condition, the page write times using masking were constant for each append operation as it prevented additional 0 to 0 writes due to previously written values. This suggests that strategic use of write patterns by way of masked overwriting when modifying a previously unwritten field in a page can lead to significant write time improvements for small data writes over other techniques. From Figure 4.5, the mean value of the normal distribution is shifted right on the histogram. It is conjectured that this is due to sampling errors and the resolution of the timing system compared to actual times to complete the operations. To accommodate for this in analysis, jitter was added to the sample set producing the results in Figure 4.5b in order to more clearly visualize the distribution supporting that isolated single byte write times across are normally distributed and the uniformity of the behaviour across the device.

In practice, data management on embedded systems is challenging. Users often choose to treat the serial NOR Dataflash as a viable re-write-in-place type device when in fact they are unaware of the high cost associated with erase-before-write operation that is implicitly being conducted. Using re-writing versus writing with erase has the potential to significantly reduce operational times and energy consumption. Consider Table 4.3 which compares write times and masked overwriting and erase-before-write append operations for appending 1 byte values in a sequential fashion to a 512 byte page. The table shows the total cumulative time including page erase costs

between each write for non-masked writes. Masked overwriting reduces the time required for append operations which corresponds to a reduction in total energy for the operations and number of erase operations. Additionally, wear on the device is reduced as page erases are not required between each write and masking reduces oxide degradation on rewrites.

4.4 Use Cases

For embedded applications, utilizing the ability to append or modify existing data without having to incur erases extends the life of the device as well as simplifies data management for common applications. The following use cases show how masked overwriting offers performance improvements in both time and lifetime.

4.4.1 Data Logging

Data logging applications [FG12] often use serial NOR Dataflash, and energy consumption and lifetime is paramount with the goal of minimizing service and maximizing field life. With data logging, the system is configured to take a series of defined measurements at regular periodic intervals. Devices such as the Arduino have found significant inroads in these applications [FG12] due to the low cost and ease of use. Common applications record a 12-bit analogue-to-digital converter every 1 minute and store the data to persistent storage.

An application logging a 2-byte record every minute writes to the same logical page 256 times (512 byte page size). Without overwriting, each record stored causes a full page write to a newly erased flash page. In comparison, using masked overwriting allows the application to write to the same page 256 times (appending a new record after each write). Using masked overwriting is significantly faster for page writes and reduces page erases by 256 times. For each write operation 2 bytes of data and 2 bytes of mask are transferred between the host and memory for a total of 4 bytes per write. Overwriting improves write performance, extends lifetime, and reduces energy requirements by reducing the ratio of writes to erases while being able to maintain record level consistency.

Utilizing overwriting with serial NOR Dataflash makes it competitive with storing data on NAND flash technologies such as a SD card. Although NAND flash has inherent speed advantages over NOR flash, NAND flash must always read and write complete pages. SD cards do not maintain internal buffers, so a complete page must be transferred to the host, modified

Table 4.4: Comparison between raw SD storage and serial NOR Dataflash (SNDF) using masked overwrite strategy.

	Memory Device	Record Size (bytes)	Data Transfer Size (bytes)	Data Overhead per write	Time (ms)
Data Logging	SD Card	2	512	510	2.61
	SNDF with masked overwriting	2	4	2	0.40
Bit Vector	SD Card	1	1024	1023	4.57
	SNDF with masked overwriting	1	2	1	0.33

and then written back to the SD card. With masked overwriting and serial NOR Dataflash, only the new record is transferred to the device as it is able to buffer pages internally using the SRAM buffers and a new page is not needed for each write.

Table 4.4 presents actual device measurements using these two strategies. For a single record write, serial NOR Dataflash with masked overwriting significantly outperforms the SD card. In the course of one day, this translates to 1440 records. For the SD card this results in 737 280 bytes transferred versus 5 760 bytes resulting in a savings of approximately 99%. Additionally, the erase operations have been reduced by a factor of 256 which directly translates to significant energy savings and increased device lifetime without having to use a complex page remapping strategy.

4.4.2 Bit Vectors

Bit vectors [RSW05] allow for compact storage of data. Bit vectors are not as efficient in persistent flash memory as data must be read and written at the page level. Consequently, updating a single bit in a bit vector in flash requires a whole page to be written. They are highly desirable structures due to their compact nature, and can be found in secondary storage systems to indicate the status of a given record known as a tombstone. This approach requires little storage overhead but presents a significant I/O overhead especially for serial accessed memories.

Overwriting with serial NOR Dataflash offers a major advantage when using bit vectors. Not only can the amount of data be reduced, but using the masked overwrite strategy, extra page copies and erases can be minimized.

To accomplish this, a write mask is brought into one of the SRAM buffers on the memory device. As demonstrated, the mask can be written over live data without impacting the state of the data. Taking advantage of the proposed strategy, the host can then update the location of the bit vector and overwrite the data in memory assuming it is an allowed transition (1 to 0). Since only one field has changed in the buffer, it can be written back to its original page leaving the original data intact. Compared to utilizing a bit vector strategy for record management with SD cards, this strategy significantly reduces data transfers and writes as well as minimizing data movement and erases on the serial NOR Dataflash. Table 4.4 presents actual device measurements using these two strategies with masked overwriting offering a performance improvement.

4.5 Comments on Overwriting

Strategic management and writing of data using *masked overwriting* has the potential to increase serial NOR Dataflash lifetime, decrease energy consumption per operation and reduce average time per write operation. The result of this work supports the ability to do in-place append operations or bit manipulations for serial NOR Dataflash and provides an analysis of device performance based on write patterns.

This strategy reduces the complexity of data management as well as reducing energy costs and extending serial NOR Dataflash field life through lower write cost and a lower ratio of erase to writes on device. Understanding data write patterns can allow for a system to reduce the energy required (for both write and erase operations) as well as reducing the complexity of data management on device. Overwriting and Masked overwriting improves operation times when the system is not required to rewrite an entire page. It supports the correlation between actual write patterns and write times on devices. This understanding allows for the use of strategic write patterns to minimize write times and energy consumption. In the case with masked overwriting, the time for write can approach the lower feasible bound where the actual write time directly corresponds to record size.

Chapter 5

Flash Translation Layer for Serial NOR Flash

> The secret, I don't know... I guess you've just gotta find something you love to do and then... do it for the rest of your life. For me, it's going to Rushmore.
>
> Max Fischer - Rushmore (1998)

For memory and resource constrained devices, FTL requirements are different than with general purpose computing systems. A balance must be struck between SRAM utilization, data transfer, and total energy costs. A ideal system would have low energy consumption as well as minimize data movement. Low data movement reduces energy consumption and the risk of data loss and system inconsistency. Due to the nature of environments where embedded systems are being used, an FTL should offer data consistency and recoverability as system faults may occur causing data loss. While page mapped FTLs offer this solution, currently the page mappings are too large to be stored in SRAM for these devices. No work to date offers an ideal solution that balances all of these aspects in a holistic FTL system nor considers memory device specific algorithms to take advantage of architectural features that can offer performance gains for these systems. No FTL solution exists for serial NOR Dataflash for 8-bit resource constrained embedded devices.

5.1 Serial NOR Dataflash

Serial NOR Dataflash is a unique NOR flash device that has a page orientated design with a high speed serial interface which makes it especially suitable for resource constrained systems. Unlike other technologies, the memory device contains two SRAM buffers that are used to hold and transfer

data as it is moved in and out of the main flash memory block (Section 2.2.2 Figure 2.12). Data is transferred to and from the device in a serial fashion using the SP which is a four wire master-slave synchronous bus. Two lines, Master Out Slave In (*MOSI*) and Master In Slave Out (*MISO*) are dedicated for the movement of data between the memory device (slave) and host (master) processor with a Clock (*SCLK*) being used to synchronize communications. A fourth line, Chip Select (*CS*) is used for signalling between host and memory device. The host indicates to the memory device the start and end of data transmissions using CS line.

To quantify device input-output performance, the number of command operations transmitted to the device and the amount of data is analyzed and expressed in bytes. Three major groups of commands are used to control data flow between the host device and memory. All operations start with the transmission of a single command (*CMD*) byte followed by an additional number of address or data bytes including:

- internal operations that move data between the main memory block and SRAM
- write data from the SRAM buffers to main flash block
- erase operations

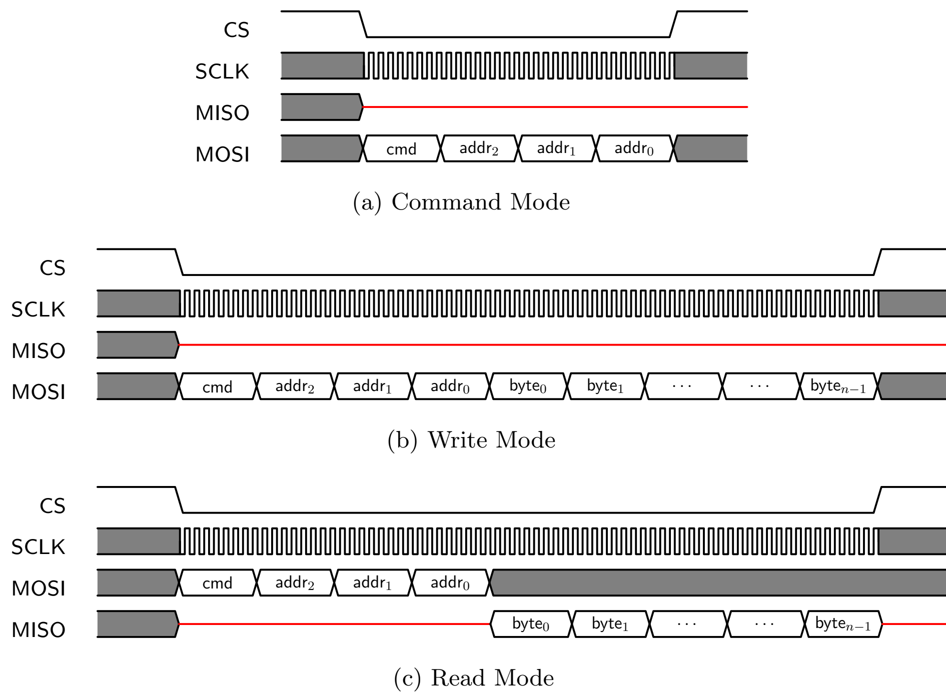
and are signalled using command transfer mode. Figure 5.1a shows state of the SPI lines during a command transfer. To transmit a command from the host to the device, the host initiates the operation by de-asserting the CS line. The host then transmits the command plus additional addressing bytes on the MOSI line in serial fashion. Once transmission is complete, the host signals the end of transmission by reasserting the CS line. Operations within the memory are self timed and the completion of an operation is determined by periodic polling of the internal status register via the SPI bus. The cost (in bytes transmitted) can be expressed as

$$C = CMD_{byte} + addr_{byte_2} + addr_{byte_1} + addr_{byte_0} \tag{5.1}$$

where C is a command operation, CMD_{byte} is the command byte associated with the operation and $addr_{byte_n}$ is used to encode buffer or page addressing information for the specific command being transmitted. The number of bytes is directly related to the time based on bus speed.

The cost of a write operation of n bytes (Figure 5.1b) is

$$W(n)_{bytes} = CMD_{byte} + addr_{byte_2} + addr_{byte_1} + addr_{byte_0} + n \tag{5.2}$$



(a) Command Mode

(b) Write Mode

(c) Read Mode

Figure 5.1: Serial NOR Dataflash Read and Write Timing Sequences Where ▭ Represents One Byte of Data

where n in the number of bytes being transmitted. This can simplified as it contains a common command sequence as expressed by Equation (5.1), producing

$$W(n)_{bytes} = C + n. \tag{5.3}$$

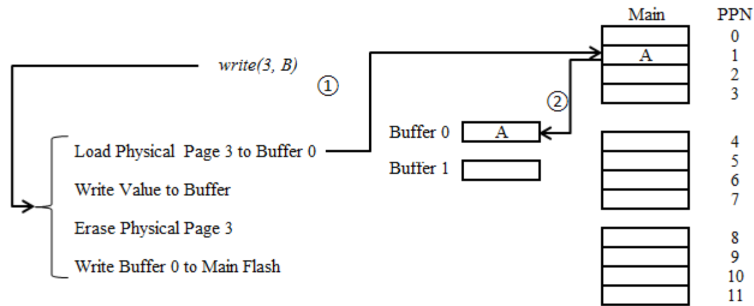
For read operations (Figure 5.1c) the cost of a read operation is

$$R(n)_{bytes} = C + n. \tag{5.4}$$

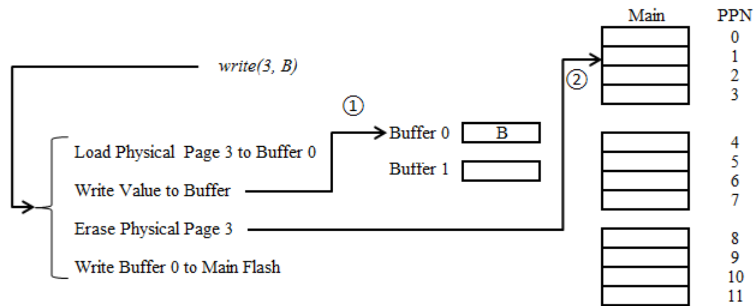
In addition to the cost in terms of the number of bytes being transmitted over the bus, consideration must be given to the internal self-timed operations of the Dataflash memory. While most internal operations are very fast (order of nanoseconds), operations such as transferring data from the flash block to one of the internal buffers or writing data from one of the internal buffers to the flash block incurs measurable time as well as consumes energy. As a result, the count of data transfer operations either from SRAM to flash or from flash to SRAM buffer is considered as a performance metric.

A combination of data movement operations are required to move data onto the device or off the device. Data movement on the device is facilitated by the on-board buffers. Figure 5.2 demonstrates the operations required to write data to an existing page on the device for serial NOR Dataflash. The host processor first sends a Command message to the host to move data from a selected page in flash to one of the two SRAM buffers. In the example (Figure 5.2a), the host is processing *write(3,B)*. The host processor instructs the memory to move physical page 3 into buffer 0. Once the page has been buffered, the host processor then issues a write message to update the data in buffer 0 (Figure 5.2b action ①) where the data is transferred from the host and written in a sequential fashion to buffer 0. Due to the physical constraints of flash memory, the buffer cannot be written back to the same physical location (unless utilizing operations highlighted in Chapter 4). The host processor must request that physical page 3 be erased with a command message (Figure 5.2b action ②), before proceeding with the transfer of buffer 0 to page 3. Once the erase operation has competed, the host processor issues a command message to initiate the transfer of buffer 0 to the physical page in flash (Figure 5.2b).

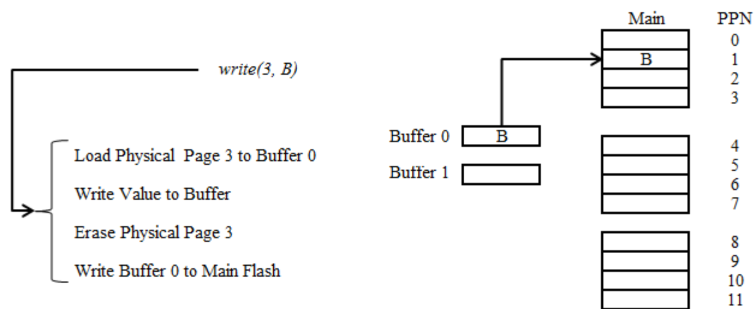
Figure 5.3a demonstrates the operations required to read data from an existing page on the device. The host processor first sends a Command message to the host to move data from a selected page in flash to one of the two SRAM buffers. In the example, the host is processing *read(3)* which attempts to read the contents of physical page 3. The host processor instructs the memory to move physical page 3 into buffer 0 with a command message



(a)

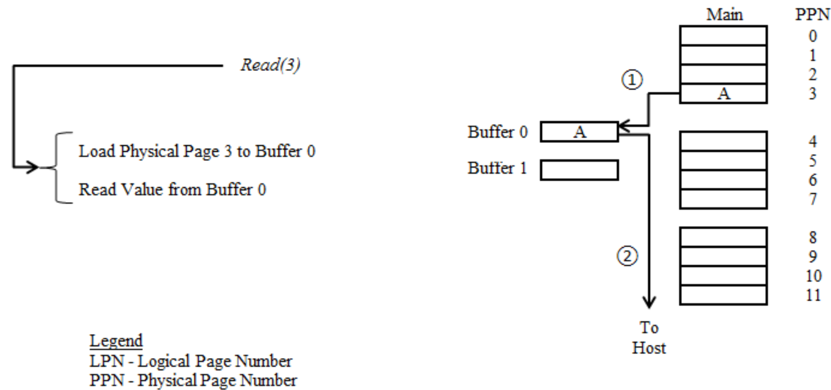


(b)

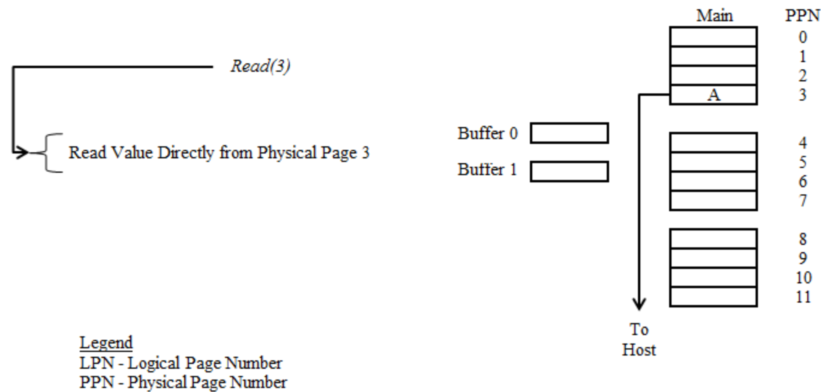


(c)

Figure 5.2: Serial NOR Dataflash Write Operations



(a) Buffered Reading



(b) Direct Reads Bypassing Internal Buffers

Figure 5.3: Serial NOR Dataflash Read Operations

(action ①). Once the page has been transferred to the buffer, the host processor issues a read command (action ②) which initiates transferring the buffer back to the host in a sequential fashion.

One unique feature of the AT45 family of devices is the ability to support direct reads from the main flash block. Data can be directly accessed without having to first buffer it in the SRAM buffers and without disturbing data held in the on device SRAM buffers. The command sequence required to initiate a direct read operation is the same format as a command to read from a buffer (Figure 5.1c), thus offering no additional overhead. Unlike in the example with buffered reading where the host is required to send a command to initial a page transfer to buffer before reading the memory, the host processor can issue a single command which will read the data directly

from main flash memory block, bypassing the buffers completely. This offers a performance increase as it allows data to be accessed from the memory device without having to first flush data that may be residing in a buffer in addition to reducing the number of commands issued to the device. The direct read operation allows data to be accessed from any position in the page and of any length. This is particularly useful as it allows for a host system to read small units of data without having to move an entire page to a buffer or to host memory.

Consider the case for direct reads versus buffered reads. The following energy cost calculations are for the AT45DB161E flash memory device [Atm04] and demonstrate the advantage of using direct reads over buffered page reads. The AT45DB161E is a 16 megabit device with 4096x528 byte-pages. The Low Frequency Continuous Array read supports bus speeds up to 33 MHz which is higher than the clock speed of many low-power 8-bit processors. This operation requires four setup bytes followed by one byte for each sequential data to be clocked onto the SPI bus (Figure 5.1c) which can be represented by Equation (5.4). In this analysis we exclude edge transition times as they are considerably smaller than device setup times. The manufacturers nominal values are used for timing and power analysis. The byte cost for accessing is

$$R(n)_{bytes} = C + n. \tag{5.5}$$

where n is the number of sequential bytes to be read from a page. It then follows that the total time to transfer n bytes of data off the flash device on an SPI bus is

$$\begin{aligned} t_{DR} &= \frac{R(n)_{bytes}}{SPI_{clk}/8} \\ &= \frac{C + n\ bytes}{SPI_{clk}/8} \end{aligned} \tag{5.6}$$

where SPI_{clk} is the SPI clock rate in MHz, and t_{DR} is the time in seconds to read n sequential bytes.

For a series of bytes to be read from the device using a buffered read, the page containing the bytes of interest must be first read into one of the two SRAM buffers and then transferred across the SPI bus. Similar to the direct read, there is a four byte setup cost to initiate the page transfer to buffer in addition to a 200 µs delay while the page is being transferred to the buffer. Once the page is loaded into the buffer the data can then be read requiring four setup bytes and then one clock byte for each data byte required. From

Equation (5.1) and (5.4), the total time for a buffered read is

$$\begin{aligned} t_{BR} &= \frac{C}{SPI_{clk}/8} + 200\,\mu s + \frac{C + n\ bytes}{SPI_{clk}/8} \\ &= \frac{2C + n\ bytes}{SPI_{clk}/8} + 200\,\mu s \end{aligned} \tag{5.7}$$

where n is the number of sequential bytes to be read from a page, SPI_{clk} is the SPI bus speed in MHz and t_{BR} is the byte time is seconds for a buffered read. The SPI clock speed is expressed in bits per second.

Operation time can be directly related to energy consumption. The total energy per byte for a direct read can be expressed as

$$E_{DR} = I_{Read} * V * t_{DR} \tag{5.8}$$

where I_{Read} is the current draw for the flash memory device for a read operation, V is the operating voltage, and E_{DR} is the energy cost per byte in Joules or a direct read operation.

It then follows from Equation (5.7) that the energy to read n bytes with a buffered page read is

$$E_{BR} = I_{Read} * V * t_{BR} \tag{5.9}$$

where E_{BR} is the energy cost per byte in Joules for a buffered read.

Consider the following example based on typical component values [Ade15] for accessing a one byte value from the flash memory using direct read where the operating voltage V=3 volts, the nominal read current I_{Read}=0.007 amp, and an SPI bus speed is 4 MHz From Equation (5.6) the total access time is calculated as

$$\begin{aligned} t_{DR} &= \frac{C + n\ bytes}{SPI_{clk}/8} \\ &= \frac{4+1}{4\,\text{MHz}/8} \\ &= 10\,\mu\text{s}. \end{aligned} \tag{5.10}$$

The estimated total energy cost for a byte read from Equation (5.8) is

$$\begin{aligned} E_{DR} &= I_{Read} * V * t_{DR} \\ &= 7\,\text{mA} * 3.0\,\text{V} * 10\,\mu\text{s} \\ &= 0.21\,\mu\text{J}. \end{aligned} \tag{5.11}$$

Consider the same single byte operation utilizing a buffered page read with the same configuration using Equation (5.7). Using Equation (5.6) the access time for the operation is calculated as

$$\begin{aligned} t_{BR} &= \frac{2C + n\ bytes}{SPI_{clk}/8} + 200\,\mu\text{s} \\ &= \frac{8+1}{4\,\text{MHz}/8} + 200\,\mu\text{s} \\ &= 218\,\mu\text{s} \end{aligned} \tag{5.12}$$

The estimated total energy cost for a byte read from Equation (5.8) is

$$\begin{aligned} E_{BR} &= I_{Read} * V * t_{DR} \\ &= 7\,\text{mA} * 3.0\,\text{V} * 218\,\mu\text{s} \\ &= 4.478\,\text{mJ}. \end{aligned} \tag{5.13}$$

In comparing the two different read methods, direct reads have significantly lower overhead compared to buffered reads, regardless of length as presented in [FL11] where $t_{DR} \ll t_{BR}$. This is directly related to the time to move data from flash to the internal SRAM buffer. Direct reads are favoured over buffered page read regardless of size due to the constant overhead of transferring to data to the buffer and should be exploited in embedded system design. The fixed overhead in comparing the two methods can be determined by subtracting Equations (5.6) and (5.7) as

$$t_{overhead} = t_{BR} - t_{DR} \tag{5.14}$$

$$= \frac{2C + n\ bytes}{SPI_{clk}/8} + 200\,\mu\text{s} - \frac{C + n\ bytes}{SPI_{clk}/8} \tag{5.15}$$

$$= \frac{C}{SPI_{clk}/8} + 200\,\mu\text{s} \tag{5.16}$$

In cases where data is being consecutively read from a single page without any other operation, direct reads offer a small advantage over buffer reads as the data can be held in the buffer after it has been loaded from flash thus incurring a one time cost (Equation 5.7) for the command and delay to load the page. All subsequent reads are at the same cost for both direct reads and buffered reads.

The direct read is especially suitable for operations were a small amount of data is required from flash in between other operations that utilize buffers. In the case where a single byte field is required to be accessed from a location a direct read offers significant advantages not only in speed but in flexibility.

In the case where a buffered read is utilized, data that is currently in the buffer may have to be evicted forcing the system to incur an additional write in the worst case. By utilizing direct reads, not only is read performance increased, it frees the buffers for write only operations. This is a key design consideration in the development of the FTL as it relies on the ability to access mapping structures directly from flash memory, eliminating the need for mapping structures to be stored in memory on the host processor while maintaining flexibility with memory buffers in addition to fast data retrieval.

5.2 The Flash Resident FTL

A significant limitation with raw device access with flash memory is that data cannot be re-written in place without the page first being erased. This limitation is addressed with using a flash translation layer.

*Fla*sh *Re*sident FTL (FlaReFTL) is a flash translation layer that is transactionally robust with a small static RAM footprint targeted for small sensor nodes and resource constrained 8-bit devices. It strives to balance addressability, speed, robustness, consistency and data transfer against energy, wear and utilization The FlaReFTL system is targeted for the Adesto family of AT45 Dataflash memory devices. Unlike other FTLs presented in Section 3.5, FlaReFTL maintains the entire flash translation structure in flash memory resulting in an FTL with a small memory footprint. This makes it suitable for use with the most resource constrained devices. It utilizes direct read operations to efficiently access translation data from flash memory without having to transfer data pages to either internal SRAM buffers or host memory. Additionally, it utilizes masked overwriting (Chapter 4) to reduce erase operations and improve write append operations.

Key features of FlaReFTL include:

- Robust and consistent flash translation layer targeted to exploit physical characteristics of serial NOR flash
- A small SRAM footprint
- Fast, low energy reads that are well suited for data analysis problems
- Deterministic wear levelling and garbage collection
- Tunable consistency levels for improved energy management
- In-place appends for data.

5.2.1 A Fully Associative Mapping Strategy

One of the numerous challenges found when developing algorithms for resource constrained devices is the lack of RAM. As previously noted, many of the existing FTL strategies are unsuitable for use due to their large SRAM requirements. On the one extreme, page-mapped schemes allow for the most efficient utilization of memory, but the mapping sizes are too large to fit into SRAM. At the other extreme, block level and hybrid schemes can produce reduced mapping structures, but the increased block size becomes a management issue as large amounts of data must be moved between flash memory and SRAM during merge operations. Not only does this increase the complexity of data management, it increases the window of risk for data loss. If a fault is generated while non-committed data is on the bus, it will result in overall data loss. The block size for merging must also be able to fit in SRAM which is not feasible on many small devices. Ideally, a system suitable for resource constrained devices will minimize the amount of uncommitted data being transferred between host and device.

FlaReFTL maintains almost the entire structure of the FTL in external flash memory minimizing the need for large volumes of uncommitted data to be transferred off device. Data is never removed from the memory for modification, but utilizes the flash SRAM buffers as temporary storage. FlaReFTL utilizes a page level mapping to ensure full block utilization without consuming large amounts of SRAM. The lowest level of the system provides physical to logical page translation as well as control blocks for monitoring page status and wear levelling By using the ability to perform direct reads from the flash memory, on chip buffers are used strictly for writing data. As a result, no page data is buffered in microprocessor RAM, leaving memory free for other application components. A buffer manager is responsible for controlling the allocation of the buffers on the device thus allowing the flash SRAM buffers to function as paged memory. This allows FlaReFTL to function on even the most memory constrained device and eliminates the need for a user to manage the state of the buffers. Unlike other FTLs, no separate garbage collection is needed as the garbage collection is an integral part of the FTL design.

5.2.2 Read and Write Operations for FlaReFTL

FlaReFTL allows a user to read and write data to flash memory, and exposes logical pages to the user. Users are prevented from accessing physical pages as the FTL is responsible for managing the movement and storage of

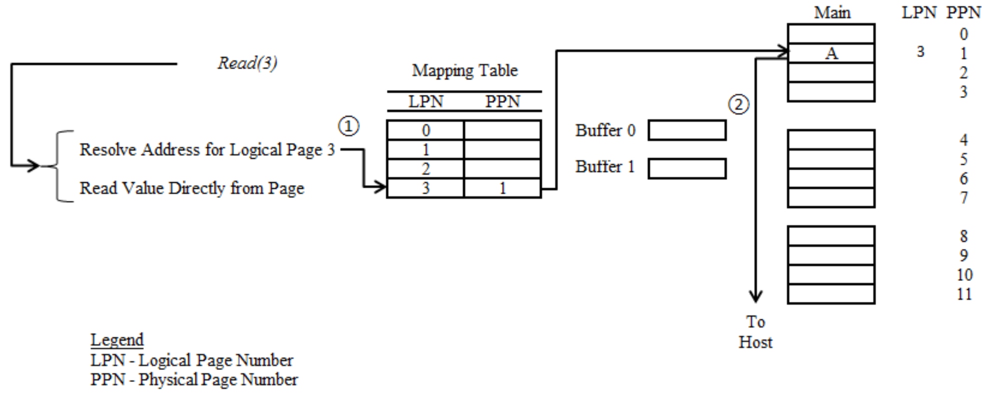


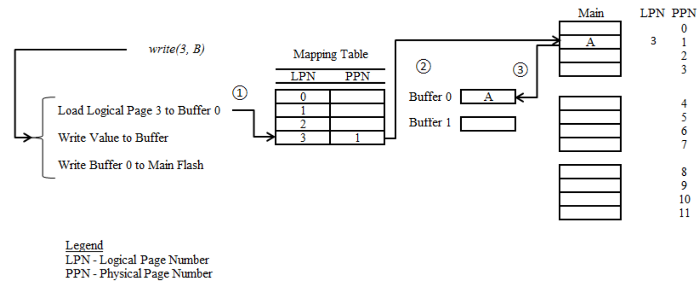
Figure 5.4: FlaReFTL Data Movement During Read Operations

data on the memory device.

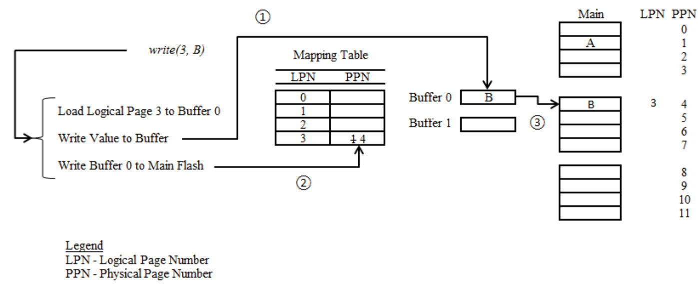
For read operations, FlaReFTL manages access to the translation tables and reading data from physical flash pages. Consider the example where the user wants to read the contents of logical page 3 as shown in Figure 5.4. For this example it is assumed that logical page 3 has been previously allocated by the system. The user first issues a *read(3)* to the system. The FTL will initially resolve the physical address for logical page 3 (action ①). Unlike other FTL systems that may be required to load mapping table pages from flash to SRAM, FlaReFTL utilizes direct reads (Section 5.1) to access translation information without inducing additional burden onto the system.

In this example the logical page resolves to physical page 1. The FTL then presents the requested data to the user (action ②) using direct reads. By exploiting direct reads, the FTL is able to provide data to the user without having to disrupt the contents of the memory SRAM buffers. No noticeable additional operation time is incurred as data is not moved from flash block to a buffer during a direct read.

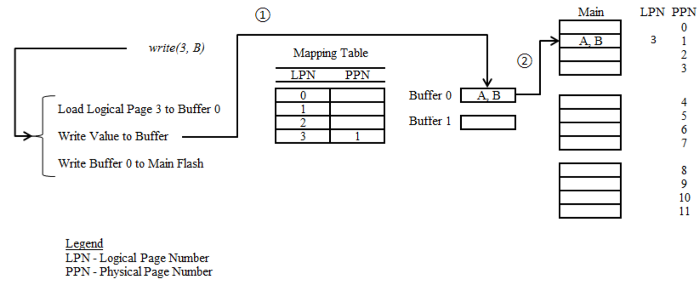
For write operations, FlaReFTL manages access to the translation tables, the physical movement of data between buffers and the tracking and allocation of logical and physical pages. Consider the example where the user wants to write data to logical page 3 as shown in Figure 5.5. For this example it is assumed that logical page 3 has previously been allocated by the system. The user first issues a *write(3,B)* to the system. The FTL will initially resolve the physical address for logical page 3 (Figure 5.5a action ①) utilizing direct reads. In this example the logical page resolves to physical page 1 (action ①). The FTL selects a buffer from the available pool of SRAM buffers



(a)



(b)



(c)

Figure 5.5: FlaReFTL Data Movement During Write Operations

available on the memory device and transfers the physical page to the buffer (Figure 5.5a action ➁). Once the page is in the buffer, the data is updated (Figure 5.5b action ➀). In this example, two data movement operations (flash block to buffer (load operation) and buffer to flash block write (store operation)) are used.

FlaReFTL supports multiple write modes. If page level consistency is selected, the page will be held in the buffer until it is forcibly evicted at which time it will be written to flash. This mode allows for subsequent writes to the same logical page without having to occur additional flash write operations and consuming free, erased physical pages. This operation reduces the number of store (write) operations required.

The second mode allows for record level consistency where after each write to a page, the buffer is written to a new physical page. To accommodate this operation the FTL requests a new physical page and transfers the updated buffer to the new physical page ((Figure 5.5b action ➁). To complete the operation, the FTL then updates the translation pages with the new physical address of the logical page (Figure 5.5b action ➂) consuming free, erased physical pages. The process of selecting a new page and updating translation table addresses is discussed in Section 5.4. This mode also incurs a larger number of load and store operation are a result of continually updating table translation addresses.

Whenever a page is written to flash, the FTL will allocate a new physical page from its reserves of available pages and exchange the old physical address and new physical address. Physical pages that are returned to the system will eventually be erased by the garbage collector. When the system is able to complete an action without having to exchange a physical page, it effectively staves off the chance of the system being required to garbage collect and erase pages which is a high energy and time costly activity.

The third write mode utilizes masked overwriting (Chapter 4) to reduce free, erased page consumption. The operation is termed as a low-energy write as it does not require a new physical page to be allocated by the system. This write operation is only available for append operations to a given page to previously unwritten locations. It is left to the user to manage their data but is well suited to sequential data logging operations. If the user is using the append write operation, *write(3,B)* will append to the page (Figure 5.5c action ➀) that was loaded into the buffer. Instead of requesting a new physical page (and engaging in a page exchange), the data in the buffer will be overwritten back to the same physical page ((Figure 5.5c action ➁). This operation offers record level consistency without burdening the system with addition page writes as well as reducing the failure opportunity window.

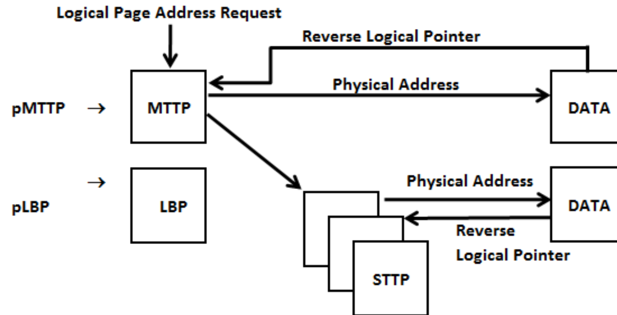


Figure 5.6: FlaReFTL Core Architecture

5.2.3 Architectural Overview

The FlaReFTL system has three different page designations that serve different functions as shown in Figure 5.6. Table Translation Pages (*TTP*) store the physical address of each logical page in the system as they are allocated. Each logical page is stored as a two byte integers starting at index 0. The byte address of each logical page in a TTP is calculated by dividing the logical page number by 2. At each position, the physical page address of the given logical page is stored. The system has one Master TTP and numerous Secondary TTPs. The Master Table Translation Page stores the physical addresses for all other TTPs in the system. The MTTP is always allocated to logical page 0, with the STTPs following in the immediate logical pages. For example, a system with 4096 x 528 byte pages will map to one MTTP and 15 STTPs. User data is stored in Data pages. Records are inserted to a logical data block by a user.

Definition 5.1. A *Record* is a contiguous number of bytes written from user space to a logical data page.

Management and placement of records are the responsibility of the application functions.

All pages contain and rely on an out-of-band data section for encoding page metadata. For a 528 byte page, 512 bytes are used for data while the remaining 16 bytes are used to store metadata. Each page has a class identifier which indicates the type of page it is along with the timestamp of the last time the page was fully written to flash. Physical pages that are associated with a logical page contain a reverse logical page pointer which identifies which logical page is assigned to a specific physical page. This

information is used by the wear leveller to enforce consistency as well as determining if a page is live. A special bit vector page is used to control the allocation and status of logical pages in the system. The Logical Busy Page (*LBP*) encodes the status of each logical page in the system. The state of a single page is encoded via a bit vector. When a logical page is allocated, a single bit is set to indicate the state of the either the physical or logical page. A single 528 byte page can be used to encode the status of 4096 pages in the system as a bit vector but is expandable for larger devices. In the LBP, each byte segment maps to a physical 8 page block which aligns with the device erase block. If a page is free, the corresponding bit is set to 0; if it is busy the bit is set to 1. Unlike other designs, the system does not record or care about the erase state of a page due to the nature of operation for the wear leveller and garbage collector.

Storage Structure

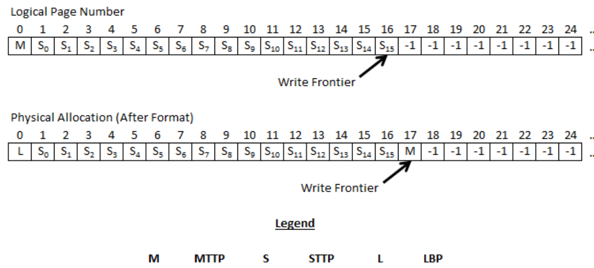


Figure 5.7: Memory Allocation Overview

Figure 5.7 shows the structure of memory after initialization. During the initialization of a new storage space, the system will write out the required TTPs and BP. During this operation, the LBP is written out to the system for the allocated logical pages required to maintain the TTPs. New pages will be available and allocated to the TTPs. The last physical block written out is the MTTP. This space indicates the start of the write frontier (Section 5.3.3).

Physical Page Allocation

Physical page operations are divided into three operational classes: *new physical page* allocation where the system provides a new physical page resource, *physical page exchange* where a previously allocated physical page is exchanged for a new physical resource and *physical page return* where a

previously allocated physical resource is returned to the pool of available physical pages.

The allocation of physical pages is not directly tracked by the FTL. Physical pages are allocated in a greedy fashion by the system and are provided by the garbage collector. The detailed operation of the garbage collector is discussed in Section 5.4. The system accesses free physical pages at the write frontier. (Section 5.3.3) and the provisioning of a new physical page does not incur any additional memory operations. Physical pages can be linked, unlinked and exchanged from logical and control pages.

When a physical page is requested by the system, the page allocation algorithm will retrieve the address of the next available physical page from the write frontier memory pointer stored in SRAM. Physical pages when allocated are bound to either a logical data page, logical TTP or a control page (LBP). The pages metadata is updated to reflect this state change and will not be considered allocated until the linking is competed.

During the normal operation of the FTL, when a page that is already defined in an existing 2-tuple is written back to flash memory, the existing physical page must be written to a new physical address as in the general case page data cannot be written back to the same physical location due to the erase-before-write constraint. This is called an exchange operation. During this operation produces a new physical address for the logical page. This is accomplished by retrieving a free physical page from the system and writing the updated logical page data to the new physical location. The system then links the new physical page to the corresponding logical page in the TTP. This unlinks the original physical page which will be eventually reclaimed by the garbage collection mechanism. After a successful exchange operation, the FTL will update the 2-tuple with the new page binding information (5.2.4).

When the system is done with a physical resource, it is returned to the system and is accomplished by unlinking the physical in the TTPs or from a control page memory pointer in SRAM. This will allow the garbage collection mechanism to eventually reclaim the page.

Logical Page Allocation

Logical page operations are divided into two operational classes: *new logical page* allocation where the system provides a new logical page resource, and *logical page return* where a previously allocated logical resource is returned to the pool of available logical pages. Logical page exchanges are not required as logical pages are bound to a physical resource which is the level where exchanges occur. Unlike other pages in the system, the LBP is not

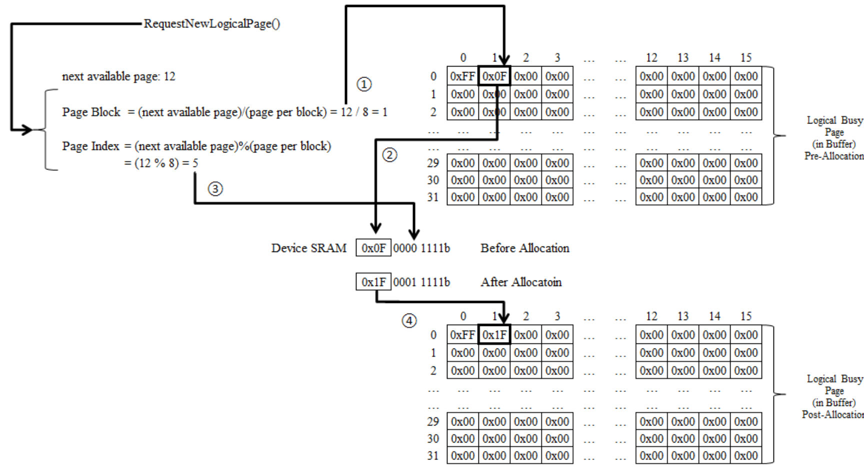


Figure 5.8: The Management of Page State with Bit Vectors

mapped to a logical page. The FTL manages the physical location of this page directly and maintains its addresses directly in host SRAM. Figure 5.8 demonstrates the operation of the LBP when a logical page is allocated by the system. Consider the operation where a new logical page is requested assuming that the LBP has been loaded into an available flash SRAM buffer. Logical pages are allocated in a greedy fashion. Starting with low bytes pages, available resources are consumed in incremental order. For logical pages, this will allow pages to be consumed in an increasing order. The FTL tracks the position of the next available page in the system. In action ①, the page block that the page belongs to is computed and transferred into host SRAM (action ②). The bit that corresponds to the next available page is computed and asserted in action ③. The modified byte is copied back to the buffer. The LBP is the written to a new physical page and the LBP memory pointer is updated in the FTL.

Logical pages are not bound to a physical page during allocation. The system assigns logical page resources based on the page availability in the LBP but it is the responsibility of the FTL to bind the logical page resource to a physical page and update mappings.

When a logical page is requested by the system, the page allocation algorithm requires a new physical page for the updated LBP. As a physical page for the LBP had previously been allocated by the system, the previous physical page must be returned to the system in a page exchange similar to what was described with the operations for exchanging a physical address.

In computing the worst case cost for allocating or returning a logical page, the system is required to load the LBP, allocate or return the logical page with a bit level operation, exchange the old physical address for the LBP for a new addresses and then write the LBP to the new location. From Equations (5.1), (5.4) and (5.3) the cost of byte transfer overhead is calculated as

$$\begin{aligned} LP_{new} &= C_{load_log} + R_{log}(1) + W_{log}(1) + C_{store_log} && (5.17) \\ &= 2C + R(1) + W(1) \\ &= 2C + (C+1) + (C+1) \\ &= 4C + 2 \\ &= 16 + 2 = 18\,\mathrm{B} && (5.18) \end{aligned}$$

where the LP_{new} is the cost in bytes to request a new logical page, C_{load_log} is the cost of loading the LBP, R_{log} is the cost of loading the page block bit vector for modification, W_{log} is the cost of writing back the page block bit vector and C_{store_log} is the cost of writing the LBP back to flash. After the operation is complete, the LBP pointer is updated in the FTL to point to the new physical location of the LBP. If the operation was a request, the system will also maintain the new logical number for binding to a physical page.

In the analysis of Equation (5.17), a logical page allocation will incur one load operation (C_{load_log}) and one store operation (C_{store_log}) in addition to the cost for data transfer.

5.2.4 Address Translation

The core translation concept with FlaReFTL is that the logical to physical page translation is encoded in flash memory and not in host SRAM. This reduces the risk of failure in the event of a fault as well as reducing host memory requirements. A system consists of a logical address space containing N pages numbered 0 to $N-1$. The FTL is responsible for mapping the logical page address to a physical page address such that the user is unaware of the physical address of the logical page. The FTL is required to update and move the position of the data due to erase-before-write and write-in-place constraints. The mappings can be expressed as a collection of 2-tuples as

$$(PageNum_{logical}, PageNum_{physical}) \tag{5.19}$$

where $PageNum_{logical}$ is the logical page number for a given page and $PageNum_{physical}$ is the physical page number. The mappings are ordered

by logical page numbers. The size of N determines the number of bytes required to encode a single page address (Section 5.2.3).

The mapping pages are logically ordered from page 0 to page 15. The first page, labelled M, is the Master Table Translation Page (MTTP) followed by fifteen Secondary Table Translation Pages (STTPs). Each Table Translation Page (TTP) stores physical addresses of logical pages in the system. The MTTP stores the physical addresses for the first 255 logical pages including the 15 STTPs in a two level tree. This structure reduces the host memory requirements as FlaReFTL only is required to store the physical address for logical page zero in SRAM.

By exploiting direct reads and the physical to logical mapping structure, the physical address of any page in the system can be determined in at most 2 direct reads of 2 bytes each. To lookup the physical address of a logical page, the TTP is determined as:

$$TTP_{number} = LPN/(number\ of\ pages\ per\ TTP) \tag{5.20}$$

and the position of the Logical to Physical Page translation in a given TTP is determined as:

$$LP_{index} = LPB \bmod (number\ of\ pages\ per\ TTP). \tag{5.21}$$

If the logical page is located in the MTTP, then the physical address of the logical page can be accessed in a single direct read operation. Figure 5.9 highlights the process for resolving the physical page address for a logical page within the first 256 address tuples.

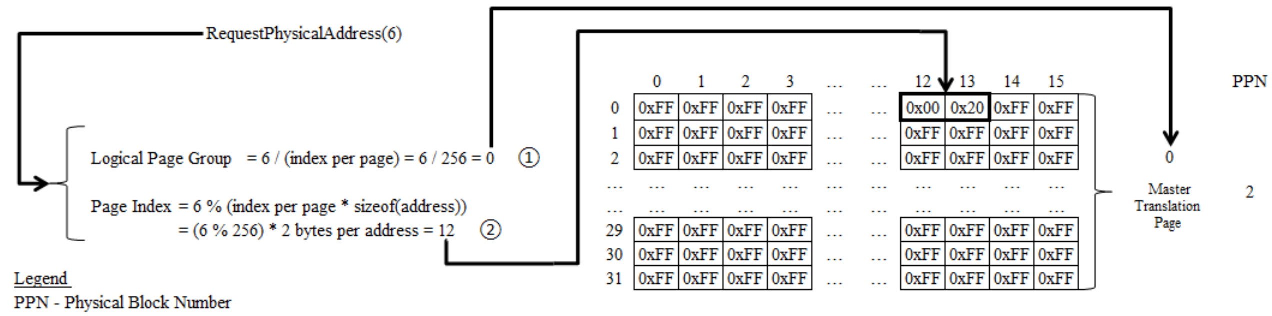


Figure 5.9: FlaReFTL Address Resolution Using Master Translation Table

Consider the example where the physical address for logical page 6 needs to be resolved. The FTL first calculates the logical page group (translation page) for the given logical address. In Figure 5.9, action ① the logical page group is computed using Equation (5.20) where the logical address is divided by the number of logical pages per translation page resulting in a logical page group of 0. This indicates that the logical page is found in the master table translation page. After the correct translation page is computed, the page index is computed using Equation (5.21) in action ② resulting in an address index of 12.

With the correct index computed, the FTL utilizes a direct read operation to read 2 bytes at index 12 which encodes the physical address of logical page 6. From Equation (5.4), the cost in byte transfer overhead for physical address resolution is

$$R_{MTTP} \quad = \quad C + 2 = 6\,\mathrm{B} \tag{5.22}$$

where R_{MTTP} is the cost in bytes to resolve an address from the Master Table Translation page, and C is the command overhead for a direct read operation.

From the calculation of the logical page group, the FTL determines if the address being resolved is not in the master table translation page. If the logical page group is not 0 (that being the the address that is being resolved is not found in the first table translation page), the FTL will use the logical page group to access the physical address of the corresponding secondary table translation page. Secondary Table Translation Pages are assigned to logical pages 1 though 15 and their respective physical page addresses encoded in the MTTP. The physical address of the STTP is read from the MTTP using the TTP_{number} which is determined by the lookup index for the respective logical page. After the physical address of the STTP is read, the physical address for the pages being resolved can read from the LP_{index} position in the corresponding STTP.

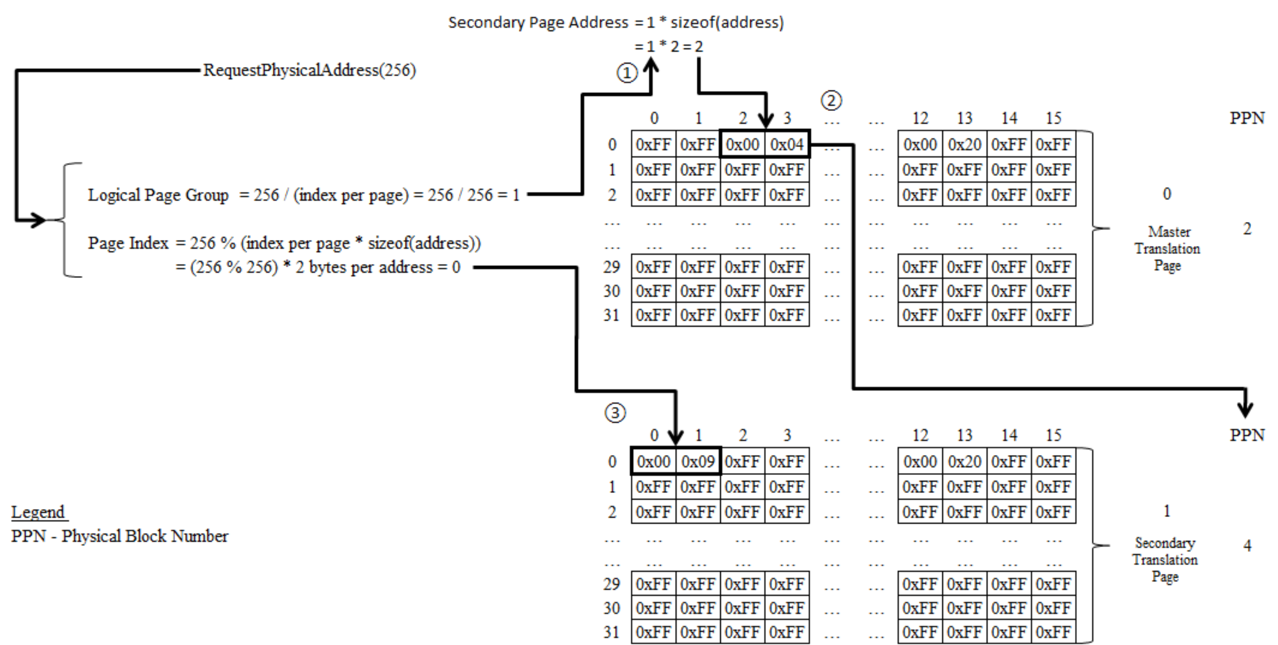


Figure 5.10: FlaReFTL Address Resolution Using Secondary Translation Table

Consider the example where the physical address for logical page 256 needs to be resolved. The FTL first calculates which logical page group (translation page) that the logical address is located in. In Figure 5.10, action ① the logical page group is computed using Equation (5.20) where the logical address is divided by the number of logical pages per translation page resulting in a logical page group of 1. This indicates to the FTL that the address being resolved is located in one of the secondary translation pages. In action ②, the FTL uses the logical page group number to compute the index offset which contains the physical address for the first secondary translation page. A direct read operation is used to read 2 bytes at index 2 which encodes the physical address of the first secondary translation page at physical page 4. The page index is computed using Equation (5.21) in action ③, and the physical page address resolved in action ②.

With the correct index computed, the FTL utilizes another direct read operation to read 2 bytes at index 0 in physical page 4, which encodes the physical address of logical page 9. From Equations (5.4) and (5.22), the cost in byte transfer overhead for physical address resolution is

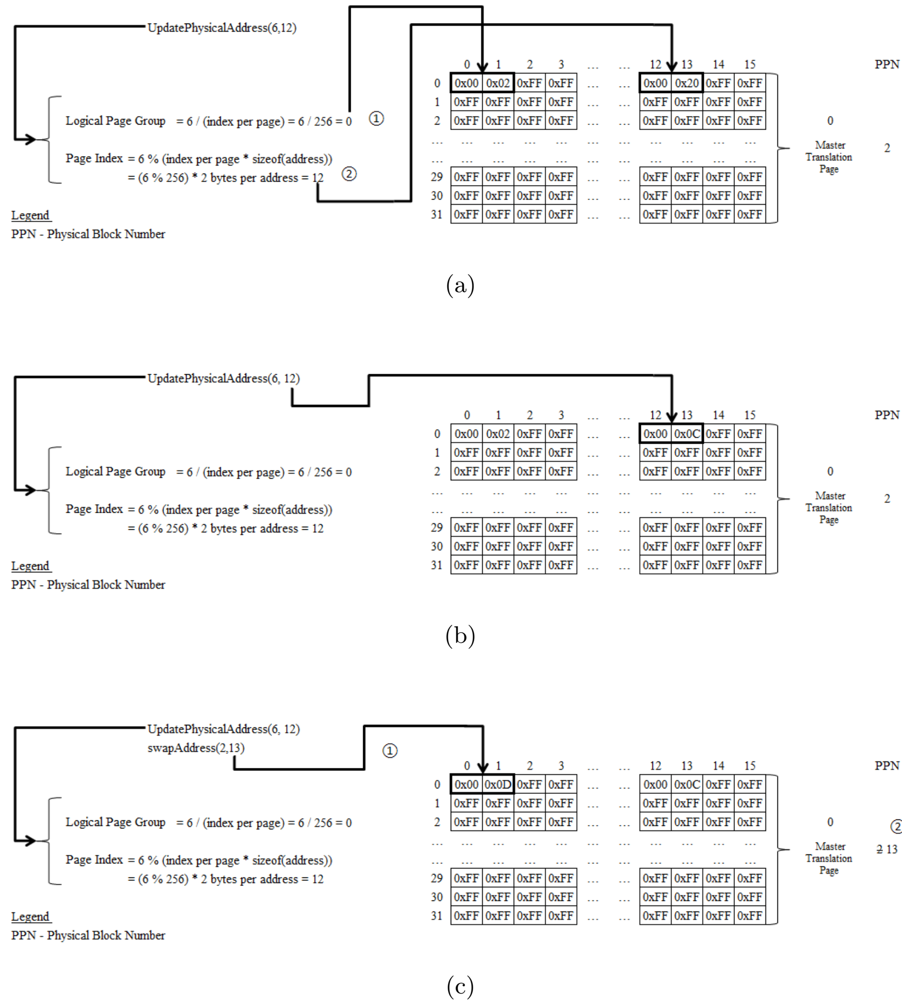
$$\begin{aligned} R_{STTP} &= R_{MTTP} + R(2) \\ &= C + 2 + C + 2 \\ &= 2C + 4 = 12\,\mathrm{B} \end{aligned} \tag{5.23}$$

where R_{STTP} is the cost in bytes to resolve an address from a Secondary Table Translation page and C is the command overhead for a direct read operation.

By using direct memory reads, page address resolution look ups can be accomplished orders of magnitude faster in terms of energy and time and without having to use flash buffers or host RAM to store translation information. No addition load or store operations are required

Updating Address Resolution Information

When a logical page is updated by the FTL, the address in the translation tables is required to be updated with the new physical page where the given logical page is now stored. To determine the TTP page that is associated with the given logical address, the logical page group of the page is computed using Equation (5.20). If the logical page group is 0, then the address is located in the MTTP and can be done in a single update operation whereas if the logical page group is not 0, then the address is located in a STTP which requires updates to both the STTP (for the logical page address update) as well as updates to the MTTP for the updated address for the STTP.



(a)

(b)

(c)

Figure 5.11: FlaReFTL Updating Master Translation Table

In the case where the address is located in the MTTP, the page index is computed using Equation (5.21) and the MTTP is brought into a buffer for writing. The physical address of the logical page is then updated at the correct page index. Before the MTTP can be written to flash memory, a new physical address is required for the MTTP which is accomplished using the page exchange operation. The new physical page address is then written to index 0 (which corresponds to logical page 0 for the MTTP) and finally the MTTP is written back to flash at the new physical address and the pointer for the MTTP updated in SRAM.

Consider the example where the physical address for logical page 6 is to be updated with the new physical address 12. The FTL first calculates which logical page group (translation page) that the logical address is located in. In Figure 5.11a, action ① the logical page group is computed using Equation (5.20) resulting in a logical page group of 0, indicating that the logical page is found in the master table translation page. The MTTP is then brought into an available buffer on the memory device for updating. The page index is computed using Equation (5.21) in action ② resulting in an address index of 12. The physical address for logical page 6 is then updated with the new physical address of 12 (Figure 5.11b). Once the address has been updated, a page exchange for the MTTP is undertaken.

In this example, the MTTP was previously stored in physical page 2 and after the exchange a new physical page of 13 has been provided. The calculation of the next available physical page is determined by the write frontier and incurs no memory access operation. Then physical page address of logical page 0 is then updated to 13 (Figure 5.11c) at which time the MTTP can be written back to physical page 13 in flash.

The cost for updating an address in the MTTP is computed with Equations (5.1), (5.3) and (5.17) as

$$
\begin{aligned}
Update_{MTTP} &= C_{load} + W_{LP}(2) \\
&\quad + W_{MTTP}(2) + C_{store} && (5.24) \\
&= 2C + 2W(2) \\
&= 2C + 2(C + 2) \\
&= 4C + 4 \\
&= 16 + 4 = 20\,\mathrm{B} && (5.25)
\end{aligned}
$$

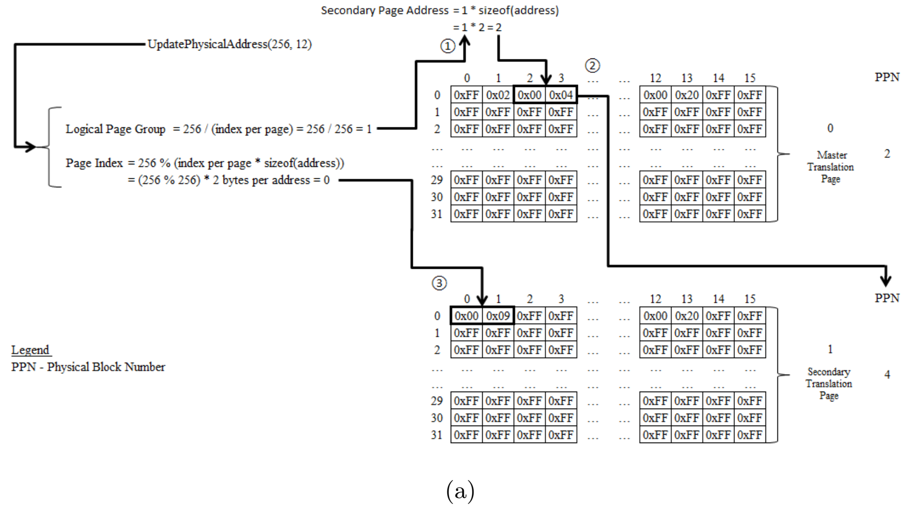
where C_{load} is the cost to load the MTTP, W_{LP} is the cost for updating the physical address for the logical page, W_{MTTP} is the cost of updating the physical address for the MTTP (logical page 0) and C_{store} is the cost for

writing the MTTP back to flash as the new physical page address. In this case as the page being accessed is located in the MTTP, an address resolution is not required for the MTTP as the physical address of the page is stored by the FTL in host SRAM. The MTTP memory pointer then updated in the FTL in SRAM to reflect the new physical location. From the analysis of Equation (5.24), updating a logical address in the the MTTP incurs a one load operation (C_{load}) and one store operation (C_{store}).

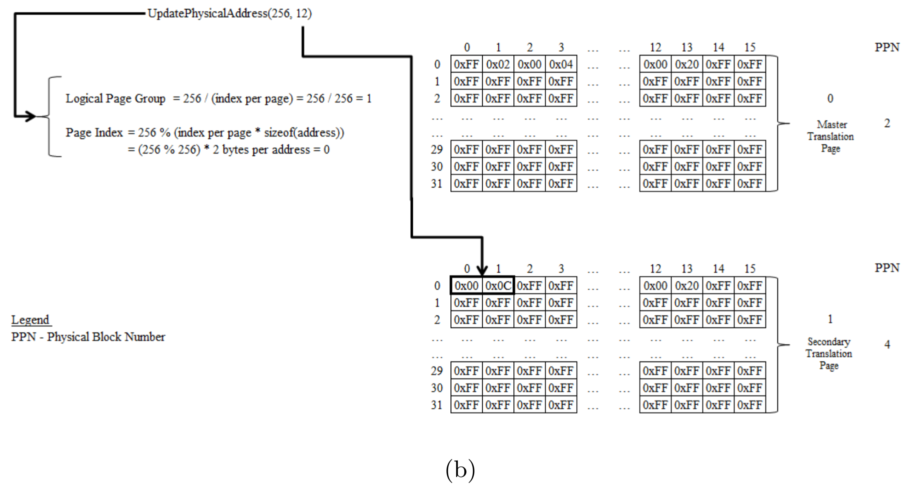
In the case where the address computed by Equation (5.20) is not in logical page group 0, the FTL must first locate the physical address of the corresponding STTP. Using a direct read from the MTTP, the STTP physical page address is resolved and the STTP is brought into a buffer for writing. The physical address of the logical page is then updated at the correct page index. Before the STTP can be written to flash memory, a new physical address is required for the STTP which is accomplished using the page exchange operation. The new physical page for the STTP is then written to the MTTP. The MTTP is brought in the buffer and the new address for STTP updated. Additionally, a new physical address is requested for the MTTP using the page exchange operation and the new physical page address is then written to index 0 (which corresponds to logical page 0 for the MTTP). Finally the MTTP is written back to flash at the new physical address and the pointer for the MTTP updated in SRAM.

Consider the example where the physical address for logical page 256 is to be updated with the new physical address 12. The FTL first calculates which logical page group (translation page) that the logical address is located in. In Figure 5.12a, action ① the logical page group is computed using Equation (5.20) resulting in a logical page group of 1, indicating that the logical page is not found in the master table translation page. The address of the STTP is then accessed with a direct read (Figure 5.12a action ②). The STTP is then brought into an available buffer on the memory device for updating. The page index is computed using Equation (5.21)(Figure 5.12a action ③) resulting in an address index of 0. The physical address for logical page 256 is then updated with the new physical address of 12 (Figure 5.12b).

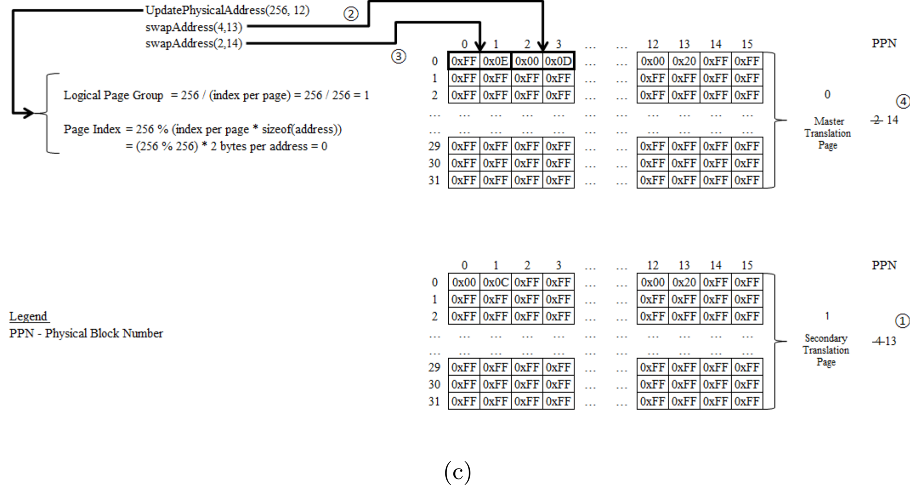
Once the address has been updated, a page exchange for the STTP must be done. In the example, the STTP was previously stored in physical page 4 and after the exchange a new physical page of 13 has been provided (Figure 5.12c action ①). The STTP is then written out to the new physical page provided. The physical page address for the STTP must then be updated in the MTTP which requires the MTTP to be brought into a buffer. Once buffered, the physical page address of the STTP in the MTTP is then updated to 13 (Figure 5.11c action ②). A page exchange for the MTTP



(a)



(b)



(c)

Figure 5.12: FlaReFTL Updating Secondary Translation Table

must be done before the MTTP is written back to flash. In the example, the MTTP was previously stored in physical page 2 and after the exchange a new physical page of 14 has been provided. Then physical page address of logical page 0 is then updated to 14 (Figure 5.12c action ③) at which time the MTTP can be written back to physical page 14 in flash (Figure 5.12c action ④).

The cost for updating an address in terms of byte transfer overhead in the STTP is computed with Equations (5.1), (5.3), (5.22) and (5.23) as

$$\begin{aligned} Update_{STTP} &= R_{STTP} + C_{load_{STTP}} + W_{LP}(2) + C_{store_{STTP}} \\ &\quad + C_{load_{MTTP}} + W_{STTP}(2) + W_{MTTP}(2) \\ &\quad + C_{store_{MTTP}} \qquad (5.26) \\ &= 4C + 5(C+2) \\ &= 9C + 10 \\ &= 36 + 10 = 46\,\mathrm{B} \qquad (5.27) \end{aligned}$$

where C_{load} is the cost to load the MTTP and STTP, W_{LP} is the cost for updating the physical address for the logical page, R_{STTP} is the cost to resolve the address of the STTP, W_{STTP} and W_{MTTP} are the costs of updating the physical addresses for the STTP and MTTP and C_{store} is the cost for writing the STTP and MTTP back to flash at the new physical page

addresses. From the analysis of Equation (5.26), updating a logical address in the STTP incurs a two load operations ($C_{load_{STTP}}$ and $C_{load_{MTTP}}$) and two store operation ($C_{store_{STTP}}$ and $C_{store_{MTTP}}$).

5.3 Consistency and Recovery with Zero-Overhead Logging

As embedded systems can suffer from unexpected resets or faults, it is critical that the FTL system used to store data is robust enough to withstand these failures while remaining consistent. As a number of pages are required to be written to the memory device during operations, it is possible that a failure may occur before completing a given operation. In this event the system is required to roll-back to the last known good state before the failure.

5.3.1 Keystoning

FlaReFTL uses a technique called *keystoning* which guarantees operation level consistency.

Definition 5.2. Keystoning is the act of writing a single page element, known as a *Keystone*, which allows the system to transition from one state to another in an atomic fashion.

The keystone page is the MTTP which maintains pointers to all other data pages in the system. All modifications to data pages or TTP that result in the pages being written to a new location in flash memory end up with a write to the MTTP. Similar to a logging file system but without the need for a separate logging operation, updates are recorded to the Dataflash without overwriting old data. Unlike other logging strategies, the system uses *Zero Overhead Logging* to manage data. A separate logging system is not maintained which would required additional read and write operations.

Definition 5.3. Zero Overhead Logging writes modified data pages o a previously erased area while leaving the original pages unchanged in storage with the ability to convert the logged pages to become live data without additional read/write overhead.

With FlaReFTL, new data is written out in a logging fashion, leaving the old data intact and then converting the data in place. In the event of a failure, the system will be able to recover be examining the operations written. New data is written to a location called the *write frontier*.

Definition 5.4. The write frontier is the location is memory where new data pages will be written. It denotes the start of a region of contiguous pages that have been previously erased. The write frontier advances through flash page addresses in a circular fashion.

In advance of the write frontier are free and erased pages where the system is able to write to without obstruction. The management of the free pages in front of the write frontier is managed by the FTL. The write frontier pointer is maintained by the FTL. Pages in advance of the write frontier have been erased and are immediately available for use. The allows the system to allocate new pages quickly without having to search for free resources. The size of the extent of free pages in front of the write frontier is controllable but the size of the frontier extent is set such that enough free pages will always be available for an operation to complete. This guarantees that all operations that mutate data can complete and not be blocked by the system waiting to reclaim pages that are *dirty*.

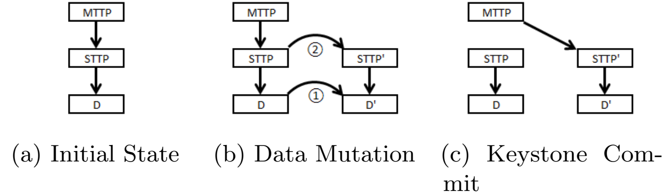
Definition 5.5. A dirty page is a page that has been previously written but the data in the page is no longer valid and is waiting to be erased by the system.

The assumption is made by the FTL that any page that is located in front of the write frontier extent and is not allocated to live data has previously been used and is dirty.

When the FTL is engaged in write operations, data and TTP pages will be written to new locations in advance of the write frontier. During the write operations, the system will maintain in flash memory, two categories of the pages; the original TTP pages that maintain the initial location of the data pages being modified and any modified pages containing updated information which may include data and TTP pages.

As an operation proceeds, the system will load, modify and write back to memory the pages at a position as required by the requested operation. Once the operation has completed, the TTPs will then be written out indicating the end of the operation. As all TTPs are linked through the MTTP, the write of the MTTP to flash will commit the changes, transitioning the system from the previous state to the current state.

Figure 5.13 demonstrates the order of operations for keystoning. In the initial state (Figure 5.13a), a data page (D) can be referenced through the MTTP and STTP. When the data page is mutated, the data page will be moved into a buffer, changed and then written back to a new physical location in flash in advance of the write frontier (Figure 5.13b action ①)



(a) Initial State (b) Data Mutation (c) Keystone Commit

Figure 5.13: FlaReFTL Updating Master Translation Table

resulting in two copies of the same page existing in memory. After the data page is updated, the corresponding STTP is buffered, modified with the new physical page number of the resource and the written to a new physical location in flash in advance of the write frontier (Figure 5.13b action ②) resulting in two copies of the same STTP existing in flash. From the view of the system, the original data has not been changed as the MTTP is still pointing at the unmodified STTP even though data has been written to the system. It is only after the MTTP has been buffered, modified with the new physical address of the STTP and written to a new location in flash in advance of the write frontier (Figure 5.13c) does the system change state from the original state of unmodified data to the new state with the modified data records.

In the event of a failure as write operations always happen to previously erased pages, the system with either see the original MTTP or the new, successfully written MTTP. Any new data pages are not accessible until the completion of the MTTP write.

5.3.2 Record Modification

To further expand on how FlaReFTL functions, consider the following example of inserting a record into an existing data set. When inserting a record in a page, the physical address of the logical page is determined from the MTTP and a STTP. New physical pages are allocated by the system. For a record insert, pages must be requested for the data page, and for any TTP that will be updated as all pages will be updated and cannot be overwritten.

Data in the MTTP

Consider the example in Figure 5.14 which demonstrates how pages are allocated for the case where a record in a data page D is being modified

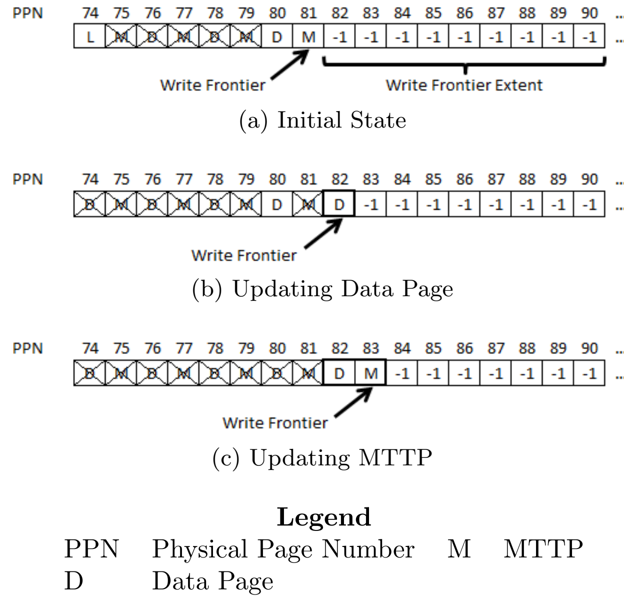
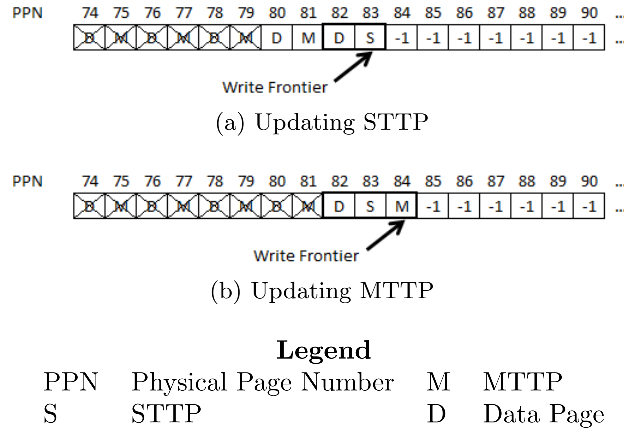


Figure 5.14: Updating Records in Data Page with MTTP

and the logical address of D is within the scope of pages managed by the MTTP. The initial state of the system is shown in Figure 5.14a. A physical page exchange is requested by the system for the data page (D). The D page is then written to flash at PPN 82 (Figure 5.14b) with the write frontier advancing to PPN 82. While data has been written out, the FTL is unable to reference the new data page as the current MTTP (M) is still active in PPN 81. To finish the write operation, the MTTP (M) must be updated with the new physical address (PPN 82) for D. After the MTTP is buffered and updated with the change, the old PPN 80 for MTTP is exchanged for a new PPN and the MTTP will be updated at page 83. The system completes the data modification operation by writing the MTTP to PPN 83 with the write frontier advancing to PPN 83. Once the MTTP is written, the FTL is now consistent and maintains valid pointers to the current D.

Data in the STTP

In the event that the data page D is not within the scope of pages managed by the MTTP, the FTL updates both the corresponding STTP followed by the MTTP. The write operation for D proceeds as in Figure 5.14b where the D page is then written to flash at PPN 82 and the write frontier



(a) Updating STTP

(b) Updating MTTP

Legend

PPN	Physical Page Number	M	MTTP
S	STTP	D	Data Page

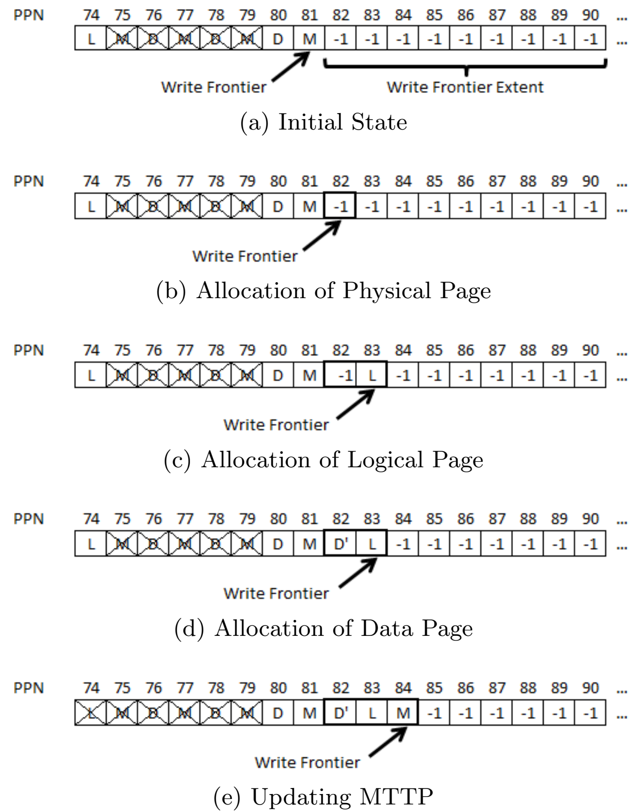
Figure 5.15: Updating Records in Data Page with STTP

advancing to PPN 82.

The system must update the new physical address (PPN 82) for D in the STTP. After the STTP is buffered and updated with the change, the old PPN for STTP is exchanged for a new PPN, and the STTP will be updated at page 83 (5.15a). To complete the write operation, the MTTP (M) must be updated with the new physical address (page 83) of the modified STTP. After the MTTP is buffered and updated with the change, the old PPN 80 for MTTP is exchanged for a new PPN and the MTTP will be updated at page 84. The system completes the data modification operation by writing the MTTP to PPN 84 (5.15b) with the write frontier advancing to PPN 84. Once the MTTP is written, the FTL is now consistent and maintains valid pointers to the current D.

Allocation of Logical Pages

Figure 5.16 demonstrates how the FTL allocates a new logical data (D′) page. When generating a new logical page, the FTL also allocates a physical page. The system first allocates a new physical page that will be bound to the logical page. A new PPN is requested for the new logical data page (PPN 82) (Figure 5.16b). The write frontier is advanced for PPN 82. While the data page is allocated by the system, nothing is written to the page until the logical binding occurs. The system then requests a new logical page from the LBP which triggers a page exchange for the LBP (PPN 74) and the LBP is written to a new location in memory (PPN 83) (Figure 5.16c). Once the logical page has been allocated to the system, this information is updated in



(a) Initial State

(b) Allocation of Physical Page

(c) Allocation of Logical Page

(d) Allocation of Data Page

(e) Updating MTTP

Legend

PPN	Physical Page Number	M	MTTP
L	LBP	D	Data Page

Figure 5.16: Allocation of a Logical Page

the OOB area of the data page and written to the flash memory at PPN 82 (Figure 5.16d). After the allocation of the logical page, the TTPs is updated with the new physical address (page 82) of the data page.

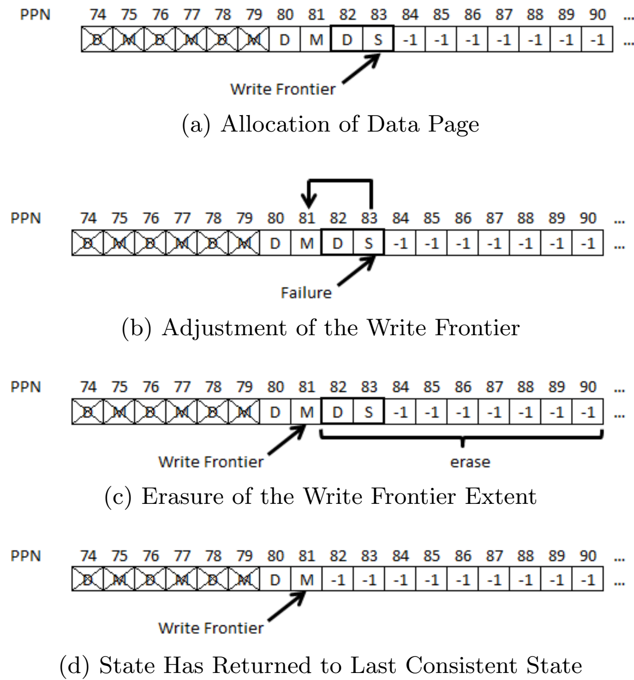
In this example, the logical page allocated is in the scope of the MTTP. If the page allocated was in the scope of the STTPs, the system would incur additional operations for the STTP exchanges. After the MTTP is buffered and updated with the change, the old PPN 81 for MTTP is exchanged for a new PPN and will be updated at page 84 (Figure 5.16e). The system completes the data modification operation by writing the MTTP to PPN 84 with the write frontier advancing to PPN 84. Once the MTTP is written, the FTL is now consistent and maintains valid pointers to the current D/ and the logical page is available for use.

5.3.3 Consistency and Recovery

When pages are written to memory, they are written in the order: data page, STTP and then MTTP. In this way, the MTTP correctly points to the most up to date version of any logical data pages and STTPs. Due to the order of writing out the blocks, if a failure occurs before writing the MTTP, the previous version of the MTTP still points to the previous data (Section 5.3.1) and will be used at restart. Only after the MTTP write has occurred will the system reflect the changes. The system now maintains the pointer to the new MTTP allowing the changes to come into view. All TTPs have an associated timestamp such that on a restart the most current master block can be located. By controlling how and when blocks are erased in relation to the write frontier, the system guarantees that live data pages being updated will never be erased until the current operation is complete. During the write process, the data from previous operations is consistent as it maintains a view of the previous MTTP. All changes to the system occur in previously erased memory locations and only with the write of the MTTP does the system transition to a new consistent state. On a restart or recovery, the most current master block can be located by scanning the OOB area of each page using direct reads. The availability of erased pages in the write frontier extend is a critical component to the design of the FTL.

Recoverability

Consider Figure 5.17, where a new record is being inserted into an existing data page where the updated data page is written into the free space at the write frontier, leaving the original data page in place. After the data pages



(a) Allocation of Data Page

(b) Adjustment of the Write Frontier

(c) Erasure of the Write Frontier Extent

(d) State Has Returned to Last Consistent State

Legend

PPN	Physical Page Number	M	MTTP
L	LBP	D	Data Page
S	STTP		

Figure 5.17: Recovery in the Event of a Failure

are updated, the STTP is written to the frontier (Figure 5.17a). If there is a failure at any point in the transaction, the FTL system will still see the consistent image of the data before any of the changes have occurred through links in the yet unchanged MTTP. On a failure, the FTL locates the latest copy to the MTTP (Figure 5.17b) and the write frontier to the last known previous MTTP. The FTL then proceeds to erase the write frontier extent removing any dirty pages that resulted from the failed operation (Figure 5.17c). A series of erase commands using Equation (5.1). The memory device supports erase units of both page and blocks (consecutive runs of 8 pages) level erases where all memory cells within the erase unit are restored to a value of 0xFF.

After the erase, the system is left in a consistent state from the last operation that successfully completed before the failure occurred (Figure 5.17d).

Recovery in the event of an unexpected system failure is greatly simplified due to the structure of FlaReFTL. On restart, while other systems must buffer and scan each page for consistency, FlaReFTL uses the direct read to access the relevant OOB information for each page scanning for the current MTTP and LBP based on timestamps. Due to the order of write operations, valid LBPs are pages with timestamps $\leq MTTP_{timestamp}$ as the MTTP will always be written last with respect to time. Once located, the position of the write frontier can be determined by the write position of the MTTP (and associated LBP) as it is the last page written for every write transaction. Once the last valid MTTP has been located, it contains consistent pointers to the rest of the data in the system. By the properties of write frontier extent, any pages within the extent can be considered dirty and the extent erased. For example, for a 4096 page device using a 4 MHz SPI bus speed, with a header of 13 bytes will only require data access time of 0.14 s to recover the file system to its last known consistent state.

Record Level and Page Level Consistency

Two levels of consistency are offered by FlaReFTL. Record level consistency ensures that for every record written to the logical page will be committed to the flash block. This can incur a larger number of erase/writes but guarantees that data records will not be lost in the event of a system failure. A looser level of consistency is also offered to improve energy consumption. Page level consistency reduces the number of erase/write operations by holding data in the Dataflash buffer until the page is full or the buffer needs to be flushed for other use. The data is held in the buffer as long as the device is powered. Page level consistency will not block the read

Table 5.1: Load, Store and Data Transfer Costs for FlaReFTL Operations

Operation	Load	Store	Data Transfer (B)
Logical Page Allocation/Return	1	1	18
Address Lookup (MTTP)	-	-	6
Address Lookup (STTP)	-	-	12
Address Update (MTTP)	1	1	20
Address Update (STTP)	2	2	46
Data Page Load	1	-	4
Data Page Store	-	1	4
Record Level Consistency Data Write where n is bytes written (excludes TTP update cost and assumes that page is loaded separately)	-	1	$4+n$
Page Level Consistency Data Write where n is bytes written(excludes TTP update cost and page is separately loaded and stored)	-	-	$4+n$

of data due to the use of direct reads and the data will be held in the buffer until it is evicted by the FTL if the buffer allocated to the data is required for another operation.

5.3.4 Operation Cost Summary

The cost in terms of data transfer to/from the Dataflash and the number of load and store operations is summarized in Table 5.1. For the values presented for record level consistency with data writes, the total cost involves loading the target page to the SRAM buffer, writing the data out to a new page for every records being writing and then updating the TTP with the new information. The worst case cost in terms of load and store operations where the page is located in the STTP is

$$\begin{aligned} Cost_{RLCW}(n) &= Load_{Data} + n \times Store_{Data} \\ &\quad +2n \times Load_{STTP} + 2n \times Store_{STTP} \\ &= (1+2n) \times Load_{Page} + 3n \times Store_{Page} \end{aligned} \tag{5.28}$$

where n is the number of records being written, $Load_{Data} and Load_{STTP}$ are the operations to load the corresponding flash page to an SRAM buffer,

$Store_{Data} and Store_{STTP}$ are the operations to store the corresponding SRAM buffer to a flash page, and $Cost_{RLCW}$ is the total load and store cost for a record level consistency write. $Load_{Page} and Load_{Page}$ are generalized storage operations.

Consider the example were a two byte record is written out 256 times using record level consistency. From Equation (5.28), the total cost for writing is

$$\begin{aligned} Cost_{RLCW}(256) &= (1 + 2 \times 256) \times Load_{Page} + 3 \times 256 \times Store_{Page} \\ &= 513 \times Load + 768 \times Store. \end{aligned} \tag{5.29}$$

Consecutive record level write operations will require 513 distinct page load operations in addition to 768 store operations. In addition, this operation will exchange 768 distinct pages from the pool of available pages, leading to increased levels of garbage collection as at then end, only one data page and two TTPs will be live.

For page level consistency with data writes from the values presented in Table 5.1, the total cost involves loading the target pages to the SRAM buffer, buffering the writes to the SRAM buffer and then writing the data out to a new page at the end of the sequence of writes. Once writing, the TTP with the new information. The worst case cost in terms of load and store operations where the page is located in the STTP is

$$\begin{aligned} Cost_{PLCW}(n) &= Load_{Data} + Store_{Data} \\ &\quad +2 \times Load_{STTP} + 2 \times Store_{STTP} \\ &= 3 \times Load_{Page} + 3 \times Store_{Page} \end{aligned} \tag{5.30}$$

where n is the number of records being written, $Load_{Data} and Load_{STTP}$ are the operations to load the corresponding flash page to an SRAM buffer, $Store_{Data} and Store_{STTP}$ are the operations to store the corresponding SRAM buffer to a flash page, and $Cost_{PLCW}$ is the total load and store cost for a page level consistency write. $Load_{Page} and Load_{Page}$ are generalized storage operations.

Consider the example were a two byte record is written out 256 times using page level consistency. From Equation (5.30), the total cost for writing is

$$Cost_{PLCW}(256) = 3 \times Load_{Page} + 3 \times Store_{Page}. \tag{5.31}$$

Thus the consecutive record level write operations will require only two distinct page load operations and two store operations for TTP modification

in addition to a single load and store for the data page. Page level consistency results in the same number of live pages as record level consistency writes. In contrast to record level consistency, this operation will only exchange three distinct pages from the pool of available pages, leading to decreased load on the garbage collector. This offers improved performance but sacrifices record level consistency in the event of a device fault.

5.3.5 In-Place Writes

FlaReFTL offers the user the ability to use masked overwriting to significantly reduce the number of erase cycles and extra data movement. As demonstrated in Chapter 4, with NOR flash, certain allowed rewrites can be accomplished without disturbing neighbouring cells. This allows data to be appended to a page without having to occur additional erase/write cycles which is particularly suited for logging applications (Section 4.4.1).

Similar to record level consistency presented in Table 5.1 and Equation (5.28), in-place writes will load the data page that is being appended to, but as it is using overwriting, the page will be written back to the existing location. As a result, no TTPs updates are required. Thus the cost for an in-place write operation is

$$\begin{aligned} Cost_{IPW}(n) &= Load_{Data} + n \times Store_{Data} \\ &= Load_{Data} + n \times Store_{Data} \end{aligned} \tag{5.32}$$

where $Cost_{IPW}$ is the total load and store cost for an in-place write with record level consistency write. In comparison to the record level consistency, in-place writes utilizing overwriting, requires 1/3 fewer store operations and only a single page load.

The savings in terms of page erases and energy consumption is significant compared to write operations that are written to fresh pages for every commit. This offers record level consistency without having to occur high levels erase operations.

Consider the example were a two byte record is written out 256 times using page level consistency. From Equation (5.32), the total cost for writing is

$$Cost_{IPW}(256) = Load_{Data} + 256 \times Store_{Data}. \tag{5.33}$$

In-place writes require a single page load operations and 2 store operations. In-place writes exchange zero pages, placing no additional load on the garbage

collector. This operation record level consistency without additional burden on the system.

A summary of the write methods is presented in Table 5.2 comparing the total operations for each method.

Table 5.2: Comparison of Load and Store Costs for FlaReFTL Write Operations

Write Operation	Cost (for n bytes)	Number of Pages Exchanged
Record Level Consistency	$(1+2n) \times Load + 3n \times Store$	$3n$
Page Level Consistency	$3 \times Load + 3 \times Store$	3
In-Place Writes	$Load + n \times Store$	0

5.4 Frontier Advance Wear Levelling and Garbage Collection

The *garbage collector* (GC) is a key component of the FTL, responsible for reclaiming dirty pages and returning them to a free and erased state. As discussed in Section 5.3.1, pages are allocated at the write frontier, which sweeps through memory pages linearly. A key challenge with all flash memory technologies is inconsistent wear across the device which leads to early device failure. With flash memory, *wear levelling* (WL) is critical to ensure that certain pages in the memory device do not experience accelerated, non-uniform wear compared to others, which impacts overall device performance. The wear levelling operations spread erase/writes across the entire device and thus extend the life of the device. With FlaReFTL, whenever a new page is allocated or an existing page is updated with new data, it is written to the page pointed to by the write frontier. Pages located in advance of the write frontier are clean as they have been guaranteed to have been erased by the wear leveller and garbage collector.

Unlike other systems, there is no separate garbage collection process to reclaim dirty pages. Instead, a dynamic wear levelling and cleaning strategy is used called *Frontier Advance Wear Levelling* (FAWL). It uses a greedy approach to keep the write frontier extent continually clean and available for use. Unlike other systems, the FTL does not need to track the individual erase cycles for pages in the system making it suitable for

resource constrained systems. FAWL prevents the need for non-deterministic GC and WL operations during a write cycle which can potentially impact performance. The FTL checks the number of available pages in the write frontier extent before a write operation and initiates a FAWL operation in a deterministic fashion if the current size of the write frontier extent is unable to support the operation without being interrupted. In advance of the write frontier, the algorithm attempts to keeps a minimum number of blocks free to guarantee that a write operation can fully complete. As data is written to the device, the write frontier advances into the clear space, ensuring that no old data has been inadvertently overwritten which decreases the size of the write extent. Located in front of the write frontier extent is the *sweep frontier extent* (SFE). Pages at the sweep frontier are inspected to see if they are live or not. Under operation, the sweep frontier advances through its extent, inspecting pages. Dirty pages are reclaimed, while live pages are moved from the sweep frontier to the write frontier to maintain a minimum size write frontier extent. The number of pages inspected is determined by the size of the extent. FAWL allows the write extent to fluctuate between a minimum and maximum size. The minimum size of the write extent is determined by the maximum number of pages that can be moved during one FAWL operation, in addition to the number number of control pages. Let $C(x)$ be the number of live pages that are equal to a given page type. The minimum write frontier extent is then calculated as

$$size_{WFE} \quad = \quad size_{SFE} + C(TTP) + C(BP) \tag{5.34}$$

where $C(TTP)$ is the count of table translation pages in the system, $C(BP)$ is the count of busy pages in the system and $size_{SFE}$ is the size of the sweep frontier extent. In the worst case, the wear levelling activity must completely copy all live data from the SFE as well as writing out a compete set of control pages. This produces a write amplification effect which provides an upper bound on system performance.

The state of the page is immediately determined from a direct read of the page OOB area. The LPN stored in the OOB is them compared to the PPN stored in the TTP. If the addresses are consistent, then the page is considered to be live, otherwise it will be treated as dirty. Live data pages will be copied to the write frontier as the sweep frontier advances through the page space, decreasing the size of the write extent. Use of the SRAM buffers prevents extra data from having to be transferred to the host and back to memory. If a page is dirty, the sweep frontier skips it and does not copy the page. As live pages are moved, each page's reverse logical pointer (from the OOB area) is recorded using a direct read and stored along with

the new physical address in host SRAM. Once all live blocks are moved, the information from the reverse logical pointers are used to update the STTPs and the MTTP. To complete the transaction, the master table translation page is written. If the transaction fails before completion, the blocks in front of the sweep frontier are undisturbed and accessible from the last known good keystone page. Upon a successful write of the master table translation page, the sweep frontier extent is erased extending the size of the write extent. This combined operation ensures the there are a minimum number of free and erased pages from the system but also ensures that pages are written in a uniform fashion across the device.

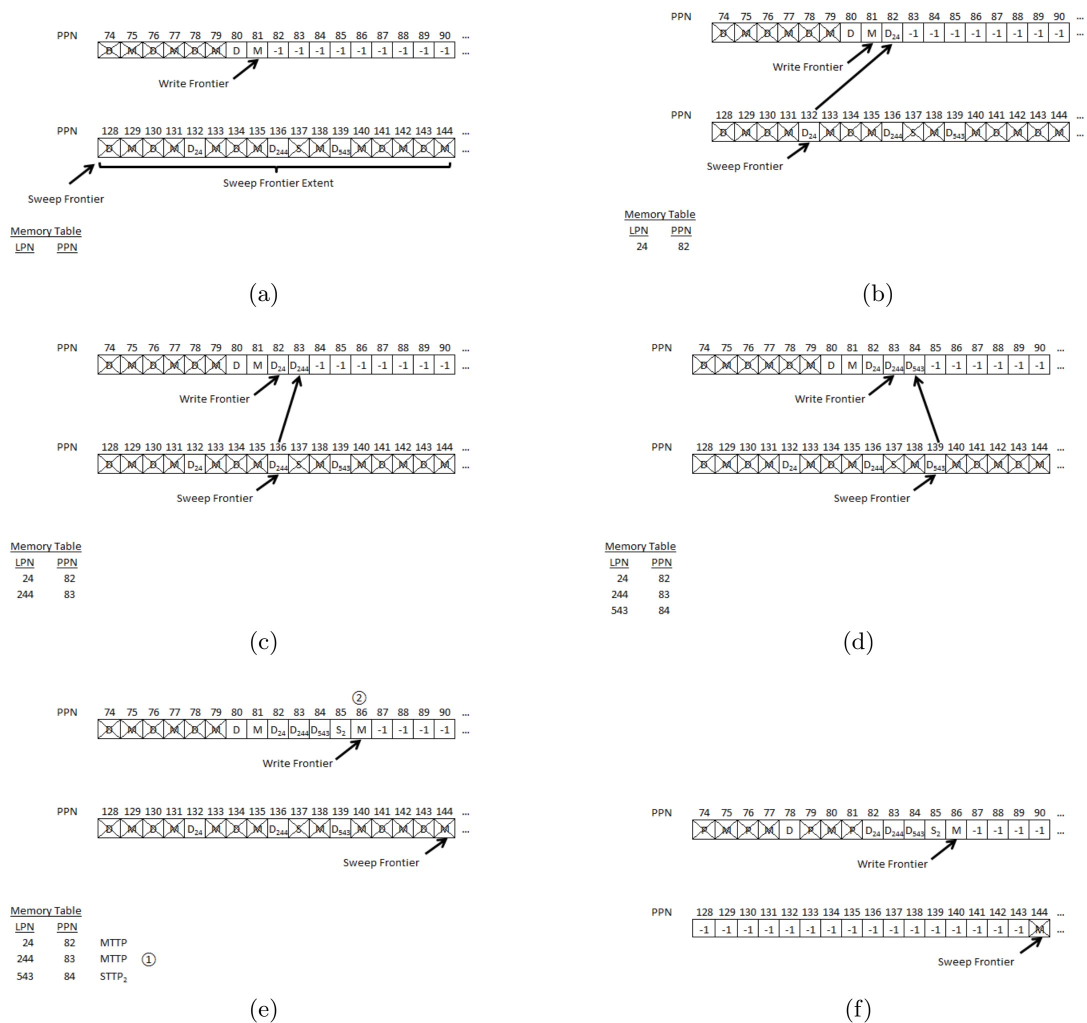


Figure 5.18: Frontier Advance Wear Levelling and Garbage Collection Operations

Consider the example in Figure 5.18 which demonstrates the frontier advance wear levelling operations. Initially the write frontier is located at LPN 81 and the sweep frontier is located at page 128 (Figure 5.18a). The sweep frontier extends forward from page 128 based on the required size. To start the FAWL activities, the FTL advances the sweep frontier into the sweep frontier extent (Figure 5.18b) and when a page is encountered, the system will determine how to move the page. The action is determined by the type of page encountered. If the page is a data page, using direct reads, the OOB area of the page will be scanned for the LPN number and resolved against the current PPN stored in the TTP. If the addresses match, the page is moved otherwise the page is skipped. If the page is a LBP, then the page's address is compared to the current LBP address pointer stored in SRAM and if the addresses are the same, the page is moved otherwise it is skipped. If the page is a TTP, the page is skipped as it will be readdressed at the end of the FAWL operations.

In this example, a data page 24 is encountered at PPN 132. The page is buffered, a new physical page requested from the system (PPN 82) and the data page written to the new location. In host SRAM, the FTL maintains a memory table that temporarily stores the mappings that will be used to update the TTPs after all data pages have been moved. The sweep frontier continues to traverse through the sweep frontier extent (Figure 5.18c) until it encounters the next live page which is data page 244 at PPN 136. The FTL repeats the page move operation moving data page 244 to PPN 83 and updating the memory table with the new page mapping for the data page. The sweep frontier continues advancing (Figure 5.18d) moving additional data pages. When the sweep frontier reaches the end of the extent, two copies of each live data page will exist.

Once the sweep frontier reaches the end of the sweep frontier extent, the FTL them proceeds to update the TTPs reflecting the new locations of the pages in the system. The FTL groups and sorts the page updates (Figure 5.18e action ①), as the system will update the TTP in decreasing order such that the MTTP will update last. The new PPNs are updated for each LPN and the required TTPs are written to new physical pages (Figure 5.18f). Finally the MTTP is updated with the new PPNs for moved STTPs and written out to a new physical location in memory. Once the MTTP has been written out, the FAWL operations will complete by updating the MTTP SRAM memory pointer and erase the sweep frontier extent (which is the region between the initial location of the sweep frontier extent and its new location).

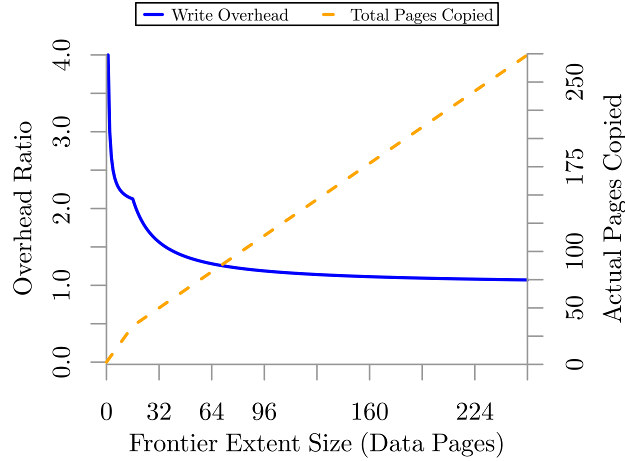


Figure 5.19: Write Amplification Overhead During FAWL

Frontier Size Considerations

Trades-offs are considered in the sizing of the write frontier extent, the trigger point and the size of the sweep frontier extent. In addition to data pages, a cost for writing out the TTP and potentially the LBP must be considered. This produces a write-amplification effect during the FAWL operation. For a small sweep frontier extent, the overhead of writing control pages can dominate. The maximum number of pages written during a FAWL is dependent on the number of valid data pages in the sweep frontier extent as

$$W_{pages} = min(C(D), C(TTP)) + C(D) + C(BP) \quad (5.35)$$
$$= min(C(D), 16) + C(D) + 1 \quad (5.36)$$

where W_{pages} is the count of pages copied, $C(D)$ is the number of data pages in the extent, $C(TTP)$ is the total number of TTP used in the system, and $C(BP)$ is the number of bit vector pages used in the system. For a small number of data pages in the sweep frontier extent, the potential overhead is high, but as the number of live pages increases the overhead decreases. Figure 5.19 shows the impact of the maximum number of pages in the sweep frontier extent and the overhead and total number of pages copied due to

write amplification. The Write Overhead demonstrates the relationship between the number of data pages versus the number of control pages written whereas Total Pages Copied indicates the worst case total write amplification for a given number of data pages. For an extent size less than the number of control pages, the overhead ratio (Write Overhead) is high as the actual number of pages written out is

$$W_{pages} = C(D) + C(TTP) + C(BP) \tag{5.37}$$

where W_{pages} is the total count of pages copied, $C(D)$ is the count of data pages, $C(TTP)$ is the number of TTP utilized to update the data pages and $C(BP)$ is the number of busy pages copied. The overhead in terms of additional pages writes is calculated as

$$OH_{FAWL} = \frac{W_{pages} - C(D)}{C(D)}. \tag{5.38}$$

In worst case, a TTP must be updated for each data page in addition to the LBP being updated leading to more control pages being written out than data pages. Thus it is desirable to maximize the count of data pages to reduce the overhead during FAWL activities.

From Equation (5.35), as the size of the extent and data pages increase above the count of control pages in the system, $C(D)$ starts to dominate leading to a decrease in overhead. This indicates that the size of the sweep frontier extent needs to be considered to minimize write amplification.

Other factors must be considered in the selection of the frontier extent size as the increasing the extent size reduced the total available pages for the user system. The system must ensure that a minimum number of pages are available for a restart such that the write frontier extent can be erased without encroaching on the sweep frontier. As the number of pages allocated by the system increases, the number of pages that will be written out during FAWL activities increases to the point where the wear levelling operations will continually write out more pages due to Equation (5.37). This will lead to average velocity of the write frontier being higher than the average velocity of the sweep frontier resulting in the write frontier encroaching on the sweep frontier. This will cause the number of available pages to fall below the minimum required for consistent operation and halt the system.

As a result, trade-offs must be considered in the selection of the size of the write frontier and sweep frontier. Selecting too small of write frontier will lead to the system encountering the scenario where the system cannot maintain a minimum write frontier size. Too large of a write frontier will

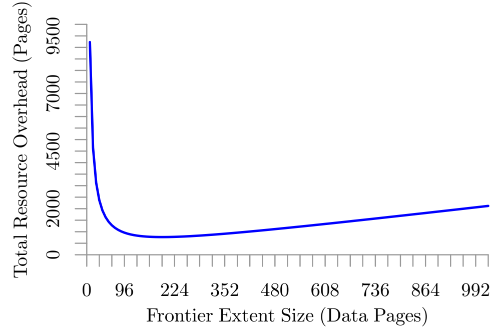


Figure 5.20: System Resource Overhead with Respect to Write Frontier Extent Size

increase the number of pages that must be held in reserve to support wear levelling operations, reducing the overall capacity. Additionally, a larger sweep frontier places an increased burden on system resources during wear levelling operations in terms of SRAM to maintain pointer tables.

From Equation (5.35), regardless of the size of the sweep frontier extent, the functional worst case is where the system is still able to maintain acceptable relative write and sweep frontier velocities. At this point, 17 of the total pages written out are assumed to be TTPs or BPs. The relationship is expressed as

$$C(D) + 17 \leq WFE_{size}. \tag{5.39}$$

The number of data pages in a given sweep must not exceed the threshold where the total number of pages written to the write frontier does not exceed the number of data pages. The system must be able to ensure that the data in the sweep frontier in addition to the control pages that can be written out to the write frontier without introducing a system fault. From Equation (5.39), the amount of reserved space required per operation is calculated as

$$ER(WFE_{size}) = \frac{17}{WFE_{size}} \tag{5.40}$$

where $ER(WFE_{size})$ is the size of the extent reserved pages required based on the size of the write frontier extent. This space is created by the continual

mixing of control pages with data pages. The minimum requirement leads to an overall assumption that for a given memory size, the minimum number of flash memory pages reserved must be able to support operations allowing the velocities of the write frontier and sweep frontier to be maintained at a consistent level. This is expressed as

$$\begin{aligned} SR_{min}(WFE_{size}) &= C(Pages) * ER(WFE_{size}) && (5.41) \\ &= C(Pages) * \frac{17}{WFE_{size}} && (5.42) \end{aligned}$$

where $SR_{min}(WFE_{size})$ is the minimum number of system wide reserved pages required. In addition to the minimum number of pages required to support FAWL operations, the system must also maintain a minimum write frontier extent size to allow for system restarts. This represents the write frontier extent size available. Thus to support FAWL operations, the write frontier extent fluctuates between WFE_{size} and $2 * WFE_{size}$. Combining this with Equation (5.41) a model for the relationship between the size of the write frontier extent and the total number of system reserved pages is given as

$$\begin{aligned} SR(WFE_{size}) &= C(Pages) * \text{Reserved} + 2 * WFE_{size} \\ &= C(Pages) * \frac{17}{WFE_{size}} + 2 * WFE_{size}. && (5.43) \end{aligned}$$

This relationship is shown in Figure 5.20 which plots the size of the write extent against the resulting number of pages that are required to be held in reserve for the system to maintain functionality. To maximize the number of pages the are available to the system, the minimum number of reserved pages is calculated as

$$SR'(WFE_{size}) = 2 - \frac{17 * C(Pages)}{WFE_{size}^2}. \quad (5.44)$$

The corresponding extent size is then calculated at the minimum by setting Equation (5.44) to zero and solving for WFE_{size}

$$\begin{aligned} 0 &= 2 - \frac{17 * C(Pages)}{WFE_{size}^2} \\ 2 &= \frac{17 * C(Pages)}{WFE_{size}^2} \\ WFE_{size} &= \sqrt{\frac{17 * C(Pages)}{2}}. && (5.45) \end{aligned}$$

For the memory device being used the number of pages is 4096. It then follows from Equation (5.45) that the extent size is

$$\begin{aligned} WFE_{size} &= \left\lceil \sqrt{\frac{17 * 4096}{2}} \right\rceil \\ &= 192. \end{aligned} \tag{5.46}$$

While this value presents a theoretical minimum feasible extent size, the value of the extent is required to be a power of 2 to permit wrapping as the sweep frontier extent sweeps past the end of memory. Hence, the minimum extent size is 256 pages. Substituting this value into Equation (5.43) the required system reserve in terms of pages is calculated as

$$\begin{aligned} SR(WFE_{size}) &= C(Pages) * \frac{17}{WFE_{size}} + 2 * WFE_{size} \\ &= 4096 * \frac{17}{256} + 2 * 256 \\ &= 784. \end{aligned} \tag{5.47}$$

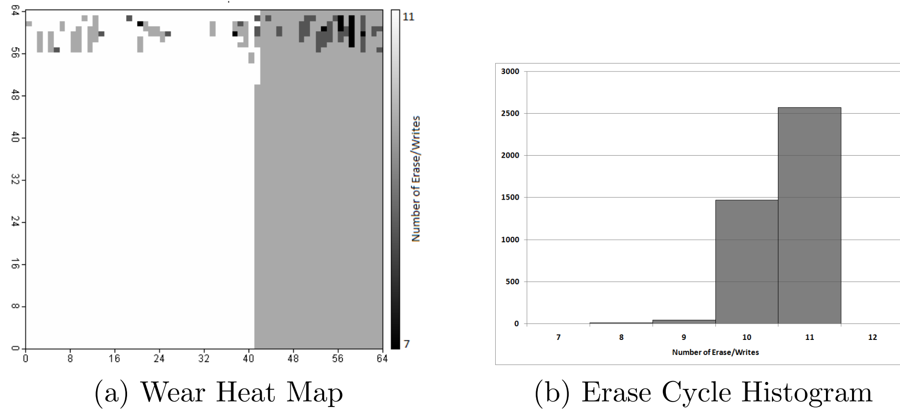
Using Equation (5.47), the minimum over provisioning required is calculated as

$$\begin{aligned} OP &= \frac{SR(WFE_{size})}{C(pages) - SR(WFE_{size})} \\ &= \frac{784}{4096 - 784} \\ &= 0.237 \end{aligned} \tag{5.48}$$

From this value it is seen that the minimum value for over provisioning is required to be 24%, leaving 76% of total capacity available for data. In practice, system performance degrades significantly as the number of live pages in the system approaches this limit. This is a result of the FAWL attempting to find available space in the system. System utilization should be limited below this value.

Uniform Wearing with FAWL

As a result of the reuse policy of FAWL, FlaReFTL presents extremely consistent levelling tendencies without the need for excessive overhead such as page erase counts as found in many other systems. Additionally, due to the page mapping scheme, complete block use is guaranteed before being



(a) Wear Heat Map (b) Erase Cycle Histogram

Figure 5.21: Results of the Wear Levelling Policy

sent for erasure. Under simulation, data records were created with a record length of 10 bytes, which is typical of what would be measured from a 12-bit device including a time stamp. To simulate a large collection period of 100 days where samples were taken every 15 minutes, 10,000 records were written to memory. The results of the wear levelling policy is seen in Figure 5.21 which presents a heat map and histogram of erase/write counts across the device. After over 40,000 erase/write cycles data has been uniformly worn across the entire device with a very small variance.

5.5 Conclusions

The Flash Resident FTL presents a unique and robust solution for resourced constrained systems that require a consistent solution for managing data on serial NOR Dataflash. This FTL offers a low static memory overhead FTL suitable for use on even the most resource constrained device. The FlaReFTL API is detailed in Appendix B.

With the FlaReFTL, all translation structures are in flash memory and never fully transferred to the host. In the event of a failure, translation structures will not be lost.

FlaReFTL offers the the following benefits:

- Minimal SRAM use as translation structures are stored in flash, allowing for use on the smallest of devices
- Consistent and efficient wear levelling and garbage collection without the overhead of tracking the number of physical page writes

- Support for three write modes
  - Page level consistent
  - Record level consistency
  - In-place record level consistency with overwriting
- Maintenance and translation lookups exploit direct reads for efficiency

FlaReFTL supports both page level consistency and record level consistency with in-place writes. FlaReFTL offers the ability for page level consistency where data can be held in the memory SRAM buffer off host until the page is full at which time it is flushed to storage. This reduces the number of writes considerably reducing the FAWL operations in addition to overall writes. It also allows resource constrained systems the ability to buffer data without having to commit a block of host SRAM to the FTL. The user must consider the trade-off that data may be lost in the event of a system failure.

Record level consistency with in-place writes offers significant advantages for resource constrained systems that require record level consistency specifically for append type operations. It allows devices to maintain record level consistency without having to occur additional erase and write operations from FAWL which can lower the write amplification for operations. Both the page level consistency and record level consistency with in-place writes significantly increase the service life of a memory device while still maintaining a functional FTL.

Chapter 6

Conclusion

> Rivers know this: there is no hurry. We shall get there some day.
>
> A.A. Milne - Winnie-the-Pooh (1882 - 1956)

6.1 Conclusions and Future Work

This work has provided an in depth analysis into data persistence issues using flash memory technologies and the underlying challenges to using flash memory with resource constrained embedded devices. It presents solutions to data management using a flash translation layer as an intermediate structure and strategies for write improvements with NOR flash. While the FTLs presented are viable for general purpose computing with NAND flash memories, none of the works in the current body of literature are suitable for resource constrained devices utilizing serial NOR flash. To address the short comings of current FTLs, the Flash Resident FTL (FlaReFTL) has been proposed as a low memory overhead FTL suitable for use on even the most resource constrained device. It is currently the only FTL available for serial NOR Dataflash compatible with 8-bit embedded systems.

6.2 Summary of Contributions

The key contributions of this work are:

- Write strategy improvements for Serial NOR Dataflash
- Minimal SRAM memory footprint flash memory management strategy
- A fault tolerant and robust flash translation layer for 8-bit embedded systems

- Consistent and recoverable data management system
- A deterministic, low overhead garbage collection and wear levelling algorithm
- Efficient buffer management through the use of direct reads
- In place writes for energy and device conservation

Through strategic use of serial NOR Dataflash SRAM buffers, unnecessary data transfers between the host and memory are limited. The FTL allows for low overhead page address resolution in addition to low overhead direct reads making the FTL suitable for use with resource constrained embedded systems. Additionally, the FTL offers different write consistency methods, allowing the end user to choose the required level of performance based on their use case. Page level consistency offers the user the ability to cache a complete page of data before writing without having to incur an additional host SRAM burden but risks data loss in the event of a system failure. Record level consistency utilizing overwriting offers the user the ability to increase the level of consistency for append type operations without incurring additional operations by the FTL due to FAWL.

This work also highlights the benefits of using the technique of masked overwriting to reduce the number of page movements required when appending data records to a given physical page. This helps to extend the field life of the device by reducing the overall number of pages erases required in addition to reducing the cost of write operations. It allows devices that are using append type operations to utilize an FTL while no having to incur excessive write amplification and as well as energy savings.

6.3 Future Work

Future work will examine improvements to the frontier advance wear levelling algorithm with efforts focusing on reducing the impact of write amplification and degree of over provisioning required. As noted in this work, as the memory utilization increases, more time is dedicated to FAWL as well as increasing the number of erase operations incurred. A technique that may improve overall device performance and reduce unnecessary moves of live data will be investigated for the wear levelling operation. Instead of compacting and moving any live data page from the sweep frontier to the write frontier, the algorithm will assess the occupancy of a block and in conjunction with a coin flip algorithm, and determine if the page should be

moved. For pages that have a high percentage of live data pages, the sweep frontier may choose to skip the block entirely reducing the overhead from moving live data. Additionally, a coin flipping operation will be implemented to inject empty pages at the write frontier to maintain a reasonable number of free pages in a physical region and reduce the packing of live data pages. This technique may reduce overall write amplification effects as the amount of data in the system increases.

As observed in the analysis of the frontier advance wear levelling algorithm (Section 5.4), write amplification at high utilization leads to the over provisioning requirements. It is observed that the mixing of data with control pages leads to this effect. A possibility to reduce this effect would be to split data and control pages into two different memory devices. As the memory devices utilize the SPI bus, the only additional overhead to the system would be one assertion line for the new memory. The FTL would partition data and control page writes between the two devices while utilizing the same FTL operations on both devices. The would allow the sweep frontier and write frontier to maintain the same velocities thus eliminating the need for substantial over provisioning on a single memory device. Additionally, the FTL would gain increased performance as the second memory device offers buffers that can now be strictly reserved for address translation operations with the buffers on the first memory device exclusively reserved for data operations. This technique will allow FlaReFTL to scale to other sizes of memory devices as the control pages would not be required to be mixed with data pages. Improved memory optimization for wear levelling can also be realized but exploiting the flash SRAM memory buffers to store temporary page mappings, which would allow the FTL to run on the extremely memory constrained devices.

Additional improvements will be investigated with masked overwriting and bit vectors. The logical busy page structure may benefit from the use masked overwriting but will require modification of the FTL algorithm. Further improvement will be realized with the removal of the LBP from the system, by tracking logical page allocation in the OOB area with bit vectors and masked overwriting The use of bit vectors and masked overwriting will be expanded for use with other data structures and NOR memory candidates.

Finally, run-time improvements in the core FTL algorithm will be investigates through source code optimization. The FTL will be integrated into other key data managements projects in the 8-bit embedded device space. This will allow users to utilize FlaReFTL in their projects, eliminating the need to manage flash translation and wear levelling activities on devices involved in data collection activities.

Bibliography

[AA11] Daniel Allred and Gaurav Agarwal. Software and Hardware Design Challenges Due to the Dynamic Raw NAND Market. Technical report, "Texas Instruments", 2011. → pages 9, 76

[AB95] Tel Aviv (IL) Amir Ban. Flash File System. Patent, 04 1995. US 5404485. → pages 43

[AB99] Ramat Hasharon (IL) Amir Ban. Flash File System Optimized for Page-Mode Flash Technologies. Patent, August 1999. US 5937425. → pages 29, 55

[Ade15] Adesto Technologies. 16-Mbit DataFlash (with Extra 512-Kbits), 2.3V or 2.5V Minimum SPI Serial Flash Memory, July 2015. `http://www.adestotech.com/wp-content/uploads/doc8782.pdf`. → pages 18, 28, 83, 98

[AF02] D. Abramovitch and G. Franklin. Disk Drive Control: The Early Years. *Annual Reviews in Control*, 26(2):229–242, 2002. → pages 35

[AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A Survey. *Computer Networks*, 54(15):2787–2805, October 2010. → pages 1, 8, 10, 76

[Ake05] Johan Akerman. Toward a Universal Memory. *Science*, 308(5721):508–510, 2005. → pages 30

[ASSC02] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless Sensor Networks: A Survey. *Computer Networks*, 38(4):393–422, 2002. → pages 1, 10, 11, 76

[Atm04] Atmel Corporation. Atmel at45db161d datasheet, 2004. `http://www.atmel.com/Images/doc2224.pdf`. → pages 6, 18, 25, 28, 29, 31, 97

[Bar08] Frank Bartos. Life After Flash Memory. *Control Engineering*, 55(4):26, 2008. → pages 30

[BCDV09] Chiara Buratti, Andrea Conti, Davide Dardari, and Roberto Verdone. An Overview on Wireless Sensor Networks Technology and Evolution. *Sensors*, 9(9):6869–6896, 2009. → pages 10

[BCMV03] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti. Introduction to Flash Memory. *Proceedings of the IEEE*, 91:489–502, April 2003. → pages 16, 17, 18, 19, 20, 22, 23, 24, 31, 78

[BN92] R. L. Boylestad and L. Nashelsky. *Electronic Devices and Circuit Theory*. Prentice Hall, 5th edition, 1992. → pages 13

[BPC+07] Paolo Baronti, Prashant Pillai, Vince W. C. Chook, Stefano Chessa, Alberto Gotta, and Y. Fun Hu. Wireless Sensor Networks: A Survey on the State of the Art and the 802.15.4 and ZigBee Standards. *Computer Communications*, 30(7):1655–1695, 2007. → pages 1, 10, 76

[Bri97] Brian Matas and Christian de Suberbasaux. *Complete Coverage of DRAM, SRAM, EPROM,and Flash Memory ICs.* Integrated Circuit Engineering Corporation, 1997. → pages 22, 23, 24

[BsKGyL02] Anyang (KR) Bum-soo Kim and Seoul (KR) Gui-young Lee. Method of Driving Remapping in Flash Memory and Flash Memory Architecture Suitable Therefor. Patent, 04 2002. US 6381176. → pages 48

[BTB04] Richard Beckwith, Dan Teibel, and Pat Bowen. Report from the Field: Results from an Agricultural Wireless Sensor Network. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, LCN '04, pages 471–478, Washington, DC, USA, 2004. IEEE Computer Society. → pages 10

[CBCF94] P Cappelletti, R Bez, D Cantarelli, and L Fratin. Failure Mechanisms of Flash Cell in Program/Erase Cycling. In *Electron Devices Meeting, 1994. IEDM '94. Technical Digest., International*, pages 291–294. Ieee, Dec 1994. → pages 23

[CCS+07] Alberto Camilli, Carlos E. Cugnasca, Antonio M. Saraiva, Andr R. Hirakawa, and Pedro L. P. Corra. From Wireless

Sensors to Field Mapping: Anatomy of an Application for Precision Agriculture. *Computers and Electronics in Agriculture*, 58(1):25–36, 2007. → pages 10, 24

[CKK10] Tae-Sun Chung, Bum-Soo Kim, and Yong-Seok Kwon. Flash Memory Access Apparatus and Method. Patent, 02 2010. US 7664906. → pages 66

[CL10] Tyler Cossentine and Ramon Lawrence. Fast Sorting on Flash Memory Sensor Nodes. In *Proceedings of the Fourteenth International Database Engineering & Applications Symposium*, IDEAS '10, pages 105–113, New York, NY, USA, 2010. ACM. → pages 11

[CLC99] Mei-Ling Chiang, Paul C. H. Lee, and Ruei-Chuan Chang. Using Data Clustering to Improve Cleaning Performance for Flash Memory. *Software - Practice & Experience*, 29(3):267–290, March 1999. → pages 43, 44, 45

[CLP09] Hyun Jin Choi, Seung-Ho Lim, and Kyu Ho Park. JFTL: A Flash Translation Layer based on a Journal Remapping for Flash Memory. *ACM Transactions on Storage (TOS)*, 4:14:1–14:22, February 2009. → pages 48, 61, 175

[CLRL08] Tae-Sun Chung, Myungho Lee, Yeonseung Ryu, and Kangsun Lee. PORCE: An Efficient Power Off Recovery Scheme for Flash Memory. *Journal of Systems Architecture*, 54(10):935–943, 2008. → pages 66

[CMMS08] L. Crippa, R. Micheloni, I. Motta, and M. Sangalli. Nonvolatile Memories: NOR vs. NAND Architectures. In Rino Micheloni, Giovanni Campardo, and Piero Olivo, editors, *Memories in Wireless Systems*, Signals and Communication Technology, pages 29–53. Springer Berlin Heidelberg, 2008. → pages 16, 17, 20, 21, 22, 23, 24

[CP07] Tae-Sun Chung and Hyung-Seok Park. STAFF: A Flash Driver Algorithm Minimizing Block Erasures. *Journal of Systems Architecture*, 53(12):889–901, 2007. → pages 48, 63, 177, 183

[CPJK04] Tae-Sun Chung, Stein Park, Myung-Jin Jung, and Bumsoo Kim. STAFF: State Transition Applied Fast Flash Translation Layer. In Christian Mller-Schloer, Theo Ungerer, and Bernhard Bauer,

editors, *Organic and Pervasive Computing ARCS 2004*, volume 2981 of *Lecture Notes in Computer Science*, pages 199–212. Springer Berlin Heidelberg, 2004. → pages 63, 182, 183

[CPK11] Tae-Sun Chung, Dong-Joo Park, and Jongik Kim. LSTAFF*: An Efficient Flash Translation Layer for Large Block Flash Memory. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 589–594, New York, NY, USA, 2011. ACM. → pages 65, 183

[CPP+09] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. A Survey of Flash Translation Layer. *Journal of Systems Architecture*, 55(5-6):332–343, May 2009. → pages 25, 26, 29, 33, 42, 43, 44, 45, 46, 48, 52, 65

[CPRH05] Tae-Sun Chung, Dong-Joo Park, Yeonseung Ryu, and Sugwon Hong. LSTAFF: System Software for Large Block Flash Memory. In Doo-Kwon Baik, editor, *Systems Modeling and Simulation: Theory and Applications*, volume 3398 of *Lecture Notes in Computer Science*, pages 704–712. Springer Berlin Heidelberg, 2005. → pages 65, 182, 183

[DB07] Gary F. Derbenwick and Joe E. Brewer. *Alternative Memory Technologies*, pages 617–740. John Wiley & Sons, Inc., 2007. → pages 18

[Des10] Peter Desnoyers. Empirical Evaluation of NAND Flash Memory Performance. *ACM SIGOPS Operating Systems Review*, 44(1):50–54, 2010. → pages 25, 27

[DNH04] Hui Dai, Michael Neufeld, and Richard Han. ELF: An Efficient Log-structured Flash File System for Micro Sensor Nodes. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, pages 176–187, New York, NY, USA, 2004. ACM. → pages 1, 10, 11, 39, 40, 76

[DTV09] A. K. Dwivedi, M. K. Tiwari, and O. P. Vyas. Operating Systems for Tiny Networked Sensors: A Survey. *International Journal of Recent Trends in Engineering (IJRTE), Issue on Computer Science*, 1(2):152–157, May 2009. → pages 11

[DZ11] Yuhui Deng and Jipeng Zhou. Architectures and Optimization Methods of Flash Memory Based Storage Systems. *Journal of*

Systems Architecture, 57(2):214–227, February 2011. → pages 11, 18, 24, 27, 42, 43, 44, 45, 46, 48, 175

[ea90] Michael L. Kazar et al. DEcorum File System Architectural Overview. In *USENIX Technical Conference*, pages 151–164, 1990. → pages 37

[ea04] G. H. Koh et al. PRAM Process Technology. In *Integrated Circuit Design and Technology, 2004. ICICDT '04. International Conference on*, pages 53–57, 2004. → pages 30

[Ele05] Samsung Electronics. Samsung k9f5608x0d 32m x 8 nand flash memory datasheet, 2005. `http://www.alldatasheet.com/datasheet-pdf/pdf/129673/SAMSUNG/K9F5608X0D.html`. → pages 31

[Ele06] Samsung Electronics. *Samsung K9WAG08U1A 1G x 8 bit/2G x 16 bit NAND Flash Memory Datasheet*, 2006. `http://www.alldatasheet.com/datasheet-pdf/pdf/177488/SAMSUNG/K9WAG08U1A.html`. → pages 29

[Eva01] Dave Evans. The Internet of Things: How the Next Evolution of the Internet Is Changing Everything. Technical report, Cisco Internet Business Solutions Group, 2001. → pages 1, 76

[FCTL12] S. Fazackerley, A. Campbell, R. Trenholm, and R. Lawrence. A Holistic Framework For Water Sustainability And Education In Municipal Green Spaces. In *2012 25th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1–6, May 2012. → pages 10

[FG12] Daniel K. Fisher and Peter J. Gould. Open-Source Hardware Is a Low-Cost Alternative for Scientific Instrumentation and Research. *Modern Instrumentation*, 1(2):8–20, 2012. → pages 88

[FL10] Scott Fazackerley and Ramon Lawrence. Reducing Turfgrass Water Consumption using Sensor Nodes and an Adaptive Irrigation Controller. In *2010 IEEE Sensors Applications Symposium (SAS)*, pages 90–94, Limerick, Ireland, February 2010. → pages 10, 11

[FL11] S. Fazackerley and R. Lawrence. A Flash Resident File System for Embedded Sensor Networks. In *Electrical and Computer Engineering (CCECE), 2011 24th Canadian Conference on*, pages 1400–1405, May 2011. → pages iv, 7, 11, 28, 29, 72, 75, 99

[Gia99] Dominic Giampaolo. *Practical File System Design*. Morgan Kaufmann Publishers Inc., 1999. → pages 36, 37

[GIMA10] Daniel Giusto, Antonio Iera, Giacomo Morabito, and Luigi Atzori, editors. *The Internet of Things: 20th Tyrrhenian Workshop on Digital Communications*. Springer, 2010. → pages 1, 8, 10, 76

[Giu13] Giuseppe Burtini and Scott Fazackerley and Ramon Lawrence. Reducing Data Transfer for Charts on Adaptive Web Sites. *SAC'13 Proceedings of the 28th Annual ACM Symposium on Applied Computing*, March 2013. → pages 2

[GKU09] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: A Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '09, pages 229–240, New York, NY, USA, 2009. ACM. → pages 45, 46, 50, 51

[GLvB+03] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 1–11, New York, NY, USA, 2003. ACM. → pages 40

[GPSZ10] A Ghobakhlou, A Perera, P Sallis, and S Zandi. Modular Sensor Nodes for Environmental Data Monitoring. In *Proceeding of the Fourth International Conference on Sensing Technology*, pages 372–377, 2010. → pages 10

[GSS09] A Ghobakhlou, S Shanmuganthan, and P Sallis. Wireless Sensor Networks for Climate Data Management Systems. In *Proceeding of the 18th World IMACS/MODSIM Congress, Cairns, Australia*, July 2009. → pages 10

[GT05a] Eran Gal and Sivan Toledo. A Transactional Flash File System for Microcontrollers. In *ATEC '05: Proceedings of the Annual Conference on USENIX Annual Technical Conference*, pages 7–7, Berkeley, CA, USA, 2005. USENIX Association. → pages 11

[GT05b] Eran Gal and Sivan Toledo. Algorithms and Data Structures for Flash Memories. *ACM Comput. Surv.*, 37(2):138–163, 2005. → pages 11

[Hag87] R. Hagmann. Reimplementing the Cedar File System using Logging and Group Commit. *SIGOPS Oper. Syst. Rev.*, 21(5):155–162, November 1987. → pages 37

[Hea02] Steve Heath. *Embedded Systems Design.* Butterworth-Heinemann, Newton, MA, USA, 2nd edition, 2002. → pages 9

[HSW+00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System Architecture Directions for Networked Sensors. *ACM SIGARCH Computer Architecture News*, 28(5):93–104, November 2000. → pages 11

[HZW11] Ping Huang, Ke Zhou, and Chunling Wu. ShiftFlash: Make Flash-Based Storage More Resilient and Robust. *Performance Evaluation*, 68(11):1193–1206, 2011. Special Issue: Performance 2011. → pages 71

[IEE07] IEEE Computer Society. IEEE Std. 802.15.4-2007, August 2007. Available: http://standards.ieee.org/getieee802/download/802.15.4a-2007.pdf. → pages 10

[IFV11] Massimo Iaculo, Francesco Falanga, and Ornella Vitale. Introduction to SSD. In Giovanni Campardo, Federico Tiziani, and Massimo Iaculo, editors, *Memory Mass Storage*, pages 213–236. Springer Berlin Heidelberg, 2011. → pages 11

[Inc05] Micron Technology Inc. Micron p5q serial phase change memory (pcm) datasheet, 2005. `https://media.digikey.com/pdf/Data%20Sheets/Micron%20Technology%20Inc%20PDFs/P5Q%20PCM.pdf`. → pages 30, 31

[Inc06] Micron Technology Inc. Nand flash 101 introduction. Technical Note TN-29-19, Micron Technology, Inc., 2006. https://www.micron.com/~/media/documents/products/technical-note/nand-flash/tn2919_nand_101.pdf?la=en. → pages 26

[Inc08] Micron Technology Inc. Micron m25p05-a datasheet, 2008. https://www.micron.com/~/media/documents/products/data-sheet/nor-flash/serial-nor/m25p/m25p05a.pdf. → pages 18, 31

[JKJ+10] Dawoon Jung, Jeong-UK Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. Superblock FTL: A Superblock-Based Flash Translation Layer with a Hybrid Address Translation Scheme. *ACM Trans. Embed. Comput. Syst.*, 9(4):40:1–40:41, April 2010. → pages 24, 29, 45, 49, 61, 62

[JRO+11] V. Jelicic, T. Razov, D. Oletic, M. Kuri, and V. Bilas. MasliNET: A Wireless Sensor Network Based Environmental Monitoring System. In *MIPRO, 2011 Proceedings of the 34th International Convention*, pages 150–155, May 2011. → pages 10

[KC08] Se Jin Kwon and Tae-Sun Chung. An Efficient and Advanced Space-Management Technique for Flash Memory using Reallocation Blocks. *Consumer Electronics, IEEE Transactions on*, 54(2):631–638, May 2008. → pages 48, 61, 169

[KJKL06] Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A Superblock-Based Flash Translation Layer for NAND Flash Memory. In *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*, EMSOFT '06, pages 161–170, New York, NY, USA, 2006. ACM. → pages 48, 61, 62, 174

[KK04] K. Kim and G. H. Koh. Future memory technology including emerging new memories. In *Microelectronics, 2004. 24th International Conference on*, volume 1, pages 377–384 vol.1, May 2004. → pages 30

[KKC+10] Hunki Kwon, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Janus-FTL: Finding the Optimal Point on the

Spectrum Between Page and Block Mapping Schemes. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '10, pages 169–178, New York, NY, USA, 2010. ACM. → pages 71

[KKN+02] Jesung Kim, Jong Min Kim, S. H. Noh, Sang Lyul Min, and Yookun Cho. A Space-Efficient Flash Translation Layer for CompactFlash Systems . *Consumer Electronics, IEEE Transactions on*, 48(2):366–375, May 2002. → pages 48, 58, 174

[KLCB08] Jin Kyu Kim, Hyung Gyu Lee, Shinho Choi, and Kyoung Il Bahng. A PRAM and NAND Flash Hybrid Architecture for High-Performance Embedded Storage Subsystems. In *Proceedings of the 8th ACM international Conference on Embedded Software*, EMSOFT '08, pages 31–40, New York, NY, USA, 2008. ACM. → pages 30, 31, 49

[KNM95] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A Flash-Memory Based File System. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 13–13, Berkeley, CA, USA, 1995. USENIX Association. → pages 25, 39, 57, 58

[KRKC11] Se Jin Kwon, Arun Ranjitkar, Young-Bae Ko, and Tae-Sun Chung. *FTL Algorithms for NAND-Type Flash Memories*, volume 15. Springer Berlin - Heidelberg, March 2011. → pages 18, 29, 33, 48, 52, 54, 55, 57, 58, 61, 62, 65, 66, 69, 166, 169, 175, 183

[KS67] K. Kahng and S. M. Sze. A Floating Gate and its Application to Memory Devices. *IEEE Transactions on Electron Devices*, 14(9):629–629, 1967. → pages 15, 78

[Lai08] S K Lai. Flash Memories: Successes and Challenges. *IBM Journal of Research and Development*, 52(4):529–535, 2008. → pages 24

[LBP08] Chul Lee, Sung Hoon Baek, and Kyu Ho Park. A hybrid flash file system based on nor and nand flash memories for embedded devices. *IEEE Transactions on Computers*, 57:1002–1008, 2008. → pages 49

[Lev08] Adam Leventhal. Flash Storage Memory. *Commun. ACM*, 51(7):47–51, July 2008. → pages 24

[Lim11] Fujitsu Semiconductor Limited. *MB85RS64A 64K (8K x 8) Bit SPI Memory FRAM*, 2011. http://www.fujitsu.com/downloads/MICRO/fsa/pdf/products/memory/fram/MB85RS64A-DS501-00009-0v01-E.pdf. → pages 30

[LJKK08] Yong-Goo Lee, Dawoon Jung, Dongwon Kang, and Jin-Soo Kim. μ-FTL: A Memory-Efficient Flash Translation Layer Supporting Multiple Mapping Granularities. In *Proceedings of the 8th ACM International Conference on Embedded software*, EMSOFT '08, pages 21–30, New York, NY, USA, 2008. ACM. → pages 43

[LP06] Seung-Ho Lim and Kyu-Ho Park. An Efficient NAND Flash File System for Flash Memory Storage. *IEEE Transactions on Computers*, 55(7):906–912, July 2006. → pages 39

[LPC+07] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A Log Buffer-Based Flash Translation Layer using Fully-Associative Sector Translation. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(3), July 2007. → pages 48, 58, 61, 166, 174, 175

[LSKK08] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems. *ACM SIGOPS Operating Systems Review*, 42(6):36–42, October 2008. → pages 48, 61, 174, 175

[LSS+09] Juan A. Lapez, Fulgencio Soto, Pedro Snchez, Andrs Iborra, Juan Suardiaz, and Juan A. Vera. Development of a Sensor Node for Precision Horticulture. *Sensors*, 9(5):3240–3255, 2009. → pages 10

[LYL09] Hyun-Seob Lee, Hyun-Sik Yun, and Dong-Ho Lee. HFTL: Hybrid Flash Translation Layer based on Hot Data Identification for Flash Memory. *IEEE Transactions on Consumer Electronics*, 55(4):2005–2011, November 2009. → pages 49

[MCO08] Rino Micheloni, Giovanni Campardo, and Piero Olivo. *Memories in Wireless Systems*. Springer Publishing Company, Incorporated, 1st edition, 2008. → pages 4, 19

[MCP+02] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless Sensor Networks for Habitat Monitoring. In *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications*, WSNA '02, pages 88–97, New York, NY, USA, 2002. ACM. → pages 10

[MDC+09] Gaurav Mathur, Peter Desnoyers, Paul Chukiu, Deepak Ganesan, and Prashant Shenoy. Ultra-low power data storage for sensor networks. *ACM Transactions on Sensor Networks (TOSN)*, 5(4):1–34, 2009. → pages 11, 40

[MDGS06] Gaurav Mathur, Peter Desnoyers, Deepak Ganesan, and Prashant Shenoy. Ultra-low Power Data Storage for Sensor Networks. In *Proceedings of the 5th International Conference on Information Processing in Sensor Networks*, IPSN '06, pages 374–381, New York, NY, USA, 2006. ACM. → pages 11, 18, 19

[MFL11] Dongzhe Ma, Jianhua Feng, and Guoliang Li. LazyFTL: A Page-Level Flash Translation Layer Optimized for NAND Flash Memory. In *Proceedings of the 2011 International Conference on Management of Data*, SIGMOD '11, pages 1–12, New York, NY, USA, 2011. ACM. → pages 34, 46, 51

[MFM+08] Raul Morais, Miguel A. Fernandes, Samuel G. Matos, Carlos Serdio, P. J. S. G. Ferreira, and M. J. C. S. Reis. A ZigBee Multi-Powered Wireless Acquisition Device for Remote Sensing Applications in Precision Viticulture. *Computers and Electronics in Agriculture*, 62(2):94–106, 2008. → pages 10

[MGZ+09] A. Matese, S. F. Di Gennaro, A. Zaldei, L. Genesio, and F. P. Vaccari. A Wireless Sensor Network for Precision Viticulture: The NAV System. *Computers and Electronics in Agriculture*, 69(1):51–58, 2009. → pages 10

[MMR08a] R. Micheloni, A. Marelli, and R. Ravasio. NAND Flash Memories. In *Error Correction Codes for Non-Volatile Memories*, pages 85–101. Springer Netherlands, 2008. → pages 16, 20, 21, 22, 23

[MMR08b] R. Micheloni, A. Marelli, and R. Ravasio. NOR Flash Memories. In *Error Correction Codes for Non-Volatile Memories*, pages 61–83. Springer Netherlands, 2008. → pages 16, 19, 20, 23

[Mur15] Charles J. Murray. Why 8-bit MCUs Refuse to Go Away: New Peripherals are Paving the Way for the Continued Success of the 8-bit Microcontroller. *Design News*, 70(9):30, 2015. → pages 2, 77

[ODH+85] John K Ousterhout, Herve Dacosta, David Harrison, John A Kunze, Mike Kupfer, and James G Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. *Proceedings of the Tenth ACM Symposium on Operating Systems Principles SOSP 85*, pages 15–24, 1985. → pages 37

[PCK+08] Chanik Park, Wonmoon Cheon, Jeonguk Kang, Kangho Roh, Wonhee Cho, and Jin-Soo Kim. A Reconfigurable FTL (Flash Translation Layer) Architecture for NAND Flash-Fased Applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(4):38:1–38:23, August 2008. → pages 174

[PDD10] Dongchul Park, Biplob Debnath, and David Du. CFTL: A Convertible Flash Translation Layer Adaptive to Data Access Patterns. *ACM SIGMETRICS Performance Evaluation Review*, 38(1):365–366, June 2010. → pages 51

[PE08] F. J. Pierce and T. V. Elliott. Regional and On-Farm Wireless Sensor Networks for Agricultural Systems in Eastern Washington. *Computers and Electronics in Agriculture*, 61(1):32–43, 2008. → pages 10

[PH12] David A. Patterson and John L. Hennessy. *Computer Organization and Design - The Hardware / Software Interface (Revised 4th Edition)*. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, 2012. → pages 9, 10, 11, 33, 36, 43, 49, 50

[PK00] G. J. Pottie and W. J. Kaiser. Wireless Integrated Network Sensors. *Commun. ACM*, 43:51–58, May 2000. → pages 2, 11

[PKCH10] Jung Sik Park, Hi-Seok Kim, Ki-Seok Chung, and Tae Hee Han. PRAM and NAND Flash Hybrid Architecture Based on Hot Data Detection. In *Mechanical and Electronics Engineering*

(ICMEE), 2010 2nd International Conference on, volume 1, pages V1–93–V1–97, August 2010. → pages 30

[PPP11] Youngwoo Park, SungKyu Park, and KyuHo Park. Design of embedded database based on hybrid storage of pram and nand flash memory. In Jianliang Xu, Ge Yu, Shuigeng Zhou, and Rainer Unland, editors, *Database Systems for Adanced Applications*, volume 6637 of *Lecture Notes in Computer Science*, pages 254–263. Springer Berlin Heidelberg, 2011. → pages 30

[PRP+04] A. Pirovano, A. Redaelli, F. Pellizzer, F. Ottogalli, M. Tosi, D. Ielmini, A. L. Lacaita, and R. Bez. Reliability Study of phase-Change Nonvolatile Memories. *IEEE Transactions on Device and Materials Reliability*, 4(3):422–427, September 2004. → pages 31

[Ram12] Ramtron International Corporation. Fm24c04b 4kb serial 5v f-ram memory, 2012. `http://www.ramtron.com/files/datasheets/FM24C04B_ds.pdf`. → pages 30

[RGA+09] Philip W. Rundel, Eric A. Graham, Michael F. Allen, Jason C. Fisher, and Thomas C. Harmon. Environmental Sensor Networks in Ecological Research. *New Phytologist*, 182(3):589–607, 2009. → pages 10

[Riq09] Riquelme . Wireless Sensor Networks for Precision Horticulture in Southern Spain. *Computers and Electronics in Agriculture*, 68(1):25–35, 2009. → pages 10

[RO92] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-structured File System. *ACM Transactions on Computer Systems (TOCS)*, 10:26–52, February 1992. → pages 5, 37, 38, 39, 57, 58

[Ros02] George Rostky. Remembering the PROM Knights of Intel. *Electronic Engineering Times*, page 85, 2002. `http://www.eetimes.com/document.asp?doc_id=1144961`. → pages 17

[RR12] Michael E Ruiz and Richard Redmond. *Cyber Command and Control : A Military Doctrinal Perspective on Collaborative Situation Awareness for Decision Making*, pages 29–47. IGI Global, 2012. → pages 1, 8

[RSPS02] V. Raghunathan, C. Schurgers, Sung Park, and M. B. Srivastava. Energy-Aware Wireless Microsensor Networks. *IEEE Signal Processing Magazine*, 19(2):40–50, March 2002. → pages 11

[RSW05] Doron Rotem, Kurt Stockinger, and Kesheng Wu. Optimizing I/O Costs of Multi-dimensional Queries Using Bitmap Indices. In KimViborg Andersen, John Debenham, and Roland Wagner, editors, *Database and Expert Systems Applications*, volume 3588 of *Lecture Notes in Computer Science*, pages 220–229. Springer Berlin Heidelberg, 2005. → pages 89

[RUJ+11] Michael G Rodriguez, Luis E Ortiz Uriarte, Yi Jia, Kazutomo Yoshii, Robert Ross, and Peter H Beckman. Wireless Sensor Network for Data-Center Environmental Monitoring. *2011 Fifth International Conference on Sensing Technology*, 10(3):533–537, 2011. → pages 10

[SBMS93] Margo Seltzer, Keith Bostic, Marshall Kirk Mckusick, and Carl Staelin. An Implementation of a Log-structured File System for UNIX. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, pages 3–3, Berkeley, CA, USA, 1993. USENIX Association. → pages 38, 39

[SCKS08] M Sanvido, F R Chu, A Kulkarni, and R Selinger. NAND Flash Memory and Its Role in Storage Architectures. In *Proceedings of the IEEE*, volume 96-11, page 18641874. IEEE, 2008. → pages 9, 16, 18, 31, 76

[Sev14] Charles Severance. Massimo Banzi: Building Arduino. *Computer*, 47(1):11–12, January 2014. → pages 1, 76

[SFL16] W. Penson S. Fazackerley and R. Lawrence. Write Improvement Strategies for Serial NOR Dataflash Memory. In *2016 29th Annual IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, May 2016. → pages iv

[SKF+10] R. E. Simpson, M. Krbal, P. Fons, A. V. Kolobov, J. Tominaga, T. Uruga, and H. Tanida. Toward the Ultimate Limit of Phase Change in $Ge_2Sb_2Te_5$. *Nano Letters*, 10(2):414–419, 2010. PMID: 20041706. → pages 30

[SOP+04] Robert Szewczyk, Eric Osterweil, Joseph Polastre, Michael Hamilton, Alan Mainwaring, and Deborah Estrin. Habitat Monitoring with Sensor Networks. *Communications of the ACM*, 47(6):34–40, June 2004. → pages 10

[Sta13] William Stallings. *Computer Organization and Architecture: Designing for Performance*. Pearson Education, Inc, USA, 2013. → pages 13, 15, 25, 35

[TBHR15] Amy Teng, Adriana Blanco, Gerald Van Hoy, and Nolan Reilly. Market Share Analysis: Microcontrollers, Worldwide, 2014. Technical report, Gartner, Inc., May 2015. → pages 2, 77

[Tec15] EVERSPIN Technologies. *MR25H256 256Kb Serial SPI MRAM Datasheet*, 2015. https://www.everspin.com/getdatasheet/MR25H256. → pages 30

[TS99] Tokyo (JP) Takayuki Shinohara. Flash Memory Card with Block Memory Address Arrangement. Patent, 05 1999. US 5905993. → pages 49, 52

[WCS+07] T. Wark, P. Corke, P. Sikka, L. Klingbeil, Ying Guo, C. Crossman, P. Valencia, D. Swain, and G. Bishop-Hurley. Transforming Agriculture through Pervasive Wireless Sensor Networks. *Pervasive Computing, IEEE*, 6(2):50–57, April-June 2007. → pages 10

[Wit13] Clint Witchalls. The Internet of Things Business Index: A Quiet Revolution Gathers Pace. Technical report, The Economist Intelligence Unit Limited, 2013. → pages 1, 76

[WLQS11] Yi Wang, Duo Liu, Zhiwei Qin, and Zili Shao. An Endurance-Enhanced Flash Translation Layer via Reuse for NAND Flash Memory Storage Systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, March 2011. → pages 49

[WLW+10] Yi Wang, Duo Liu, Meng Wang, Zhiwei Qin, Zili Shao, and Yong Guan. RNFTL: A Reuse-Aware NAND Flash Translation Layer for Flash Memory. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '10, pages 163–172, New York, NY, USA, 2010. ACM. → pages 69, 70, 71, 182

[Woo01] David Woodhouse. JFFS: The Journaling Flash File System. In *Proceedings of the Ottawa Linux Symposium*, 2001. → pages 39

[Wu10] Chin-Hsien Wu. A Self-Adjusting Flash Translation Layer for Resource-Limited Embedded Systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 9(4):31:1–31:26, April 2010. → pages 61, 71, 176

[ZBT09] Aviad Zuck, Ohad Barzilay, and Sivan Toledo. NANDFS: A Flexible Flash File System for RAM-constrained Systems. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, pages 285–294, New York, NY, USA, 2009. ACM. → pages 19

[ZSI11] Cliff Zitlaw (Spansion Inc.). The Future of NOR Flash Memory, 2011. `http://eetimes.com/design/memory-design/4215634/The-Future-of-NOR-flash-memory`. → pages 4, 9, 18, 76

[ZYLK+05] Demetrios Zeinalipour-Yazti, Song Lin, Vana Kalogeraki, Dimitrios Gunopulos, and Walid A. Najjar. Microhash: An Efficient Index Structure for Flash-based Sensor Devices. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, pages 3–3, Berkeley, CA, USA, 2005. USENIX Association. → pages 18, 19, 40

[ZYWY09] Yiming Zhou, Xianglong Yang, Liren Wang, and Yibin Ying. A Wireless Design of Low-Cost Irrigation System Using ZigBee Technology. In *International Conference on Networks Security, Wireless Communications and Trusted Computing (NSWCTC '09)*, pages 572–575, Los Alamitos, CA, USA, 2009. IEEE Computer Society. → pages 33, 42

Index

E^2PROM, 18

active, 42

battery backed-up SRAM, NVS-RAM, 14
Block, 57
BlockBasedLogging, 173
blocks, 26

chip select, CS, 98
consistency, recovery, 132

DFTL, 46
dirty page, 127

EEPROM, 8, 18
EEPROM,Electrically Erasable Programmable ROM, 17
erase before write, 33

Ferroelectric RAM, FRAM, 30
Flash Aware File Systems, 34
flash translation layer, FTL, 34
Fowler Nordheim tunnelling, FN tunnelling, 17
free page, 42

garbage collection, GC, 42
GC, garbage collector, 138

hybrid, 48

Internet of Things, 1, 8, 82
invalid page, 42

keystone, keystoning, 126

large block device, 29
LazyFTL, 46
live page, 42
logical block number, LBN, 46
logical page number, LPN, 46

microcontroller, 9
MISO, 98
MOSFET, 13
MOSI, 98
MRAM, magnetoresistive, 30
Multi-Level-Cell, MLC, 24

NOR flash, Channel Hot Electron injection, 22

Out of Bounds Area, OOB, 49
Out-of-bounds area, OOB, 25

pages, 25
Physical Block Number, PBN, 26
physical page number, PPN, 47
power off recovery, POR, 42

read cost, write cost, 29

SCLK, 98
Single-Level-Cell, SLC, 24
SPI, 98
SSD, 50
sweep frontier, sweep frontier extent, SFE, 139

switch, 60
System-On-Chip,SOC, 10

thrashing, 45

valid, 42

wear, 29
wear levelling, WL, 29
wireless sensor network, WSN, 10
WL, wear levelling, 138
write frontier, 126

zero overhead logging, logging, 126

Appendix

Appendix A

FTL Algorithms

A.1 FTL Schemes

The following section details further variations on FTL schemes for use with NAND flash.

A.1.1 Block Based Logging Schemes

FAST

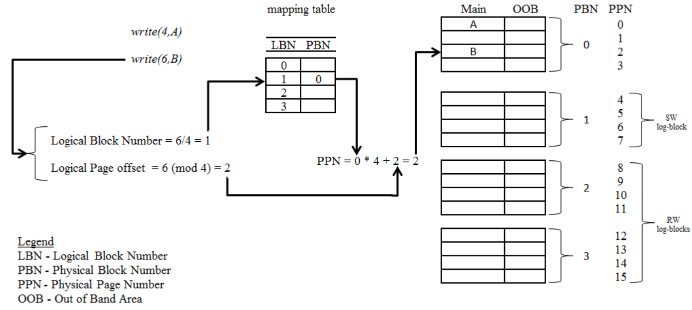
To address the noted limitations of BAST, Lee et al. have proposed a fully associated sector translation (FAST) [LPC+07] scheme. The primary difference between BAST and FAST is that unlike BAST where there is a one to many mapping relationship between data blocks and log blocks, FAST allows for a many to many mapping strategy; in essence it is a fully associative strategy where one log block can be shared between many target data blocks. Additionally, unlike previous strategies, not every live block is associated with a log block. This strategy attempts to reduce log block thrashing and the total number of erasures. It presents a significant improvement over BAST due to the fully associative scheme. Additionally, it segregates data into random write *(RW)* and sequential write *(SW)* logs based on access patterns to take advantage of switch types merges. When a log block is written strictly with in-place updates, it can be directly converted to a data block thus reducing the degree of copying required. With sequential writes, the chances of a switch merge versus a full merge are significantly increased which improves overall performance as data access patterns tend to have a high degree of sequential writes [KRKC11]. This results in a lower number of potential operations in terms of erase cycles.

The system maintains one log block that is solely used for in-place updates (SW log-block) as well as a series of RW log blocks that are shared between all blocks across the entire device. Sequential updates to previously allocated pages will utilize the SW log under the precondition that the page under update has a logical page offset of zero (being the first entry in the SW log) or that the page to be updated has a logical offset that is greater than the

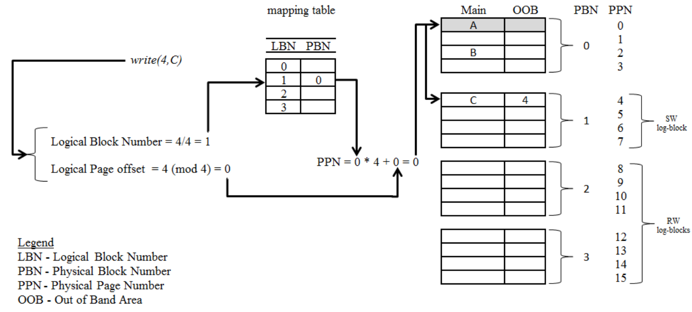
last previously entered page. If the page to be updated in the log cannot satisfy either of the constraints, then it cannot be written to the SW log and will be placed in the next available entry in the RW log. When the SW log is full, it will be merged with the target block and converted to a live data block. While the SW log block is mapped temporarily to a single data block, the RW log blocks are shared between all blocks on the device in an effort to increase block utilization before being invalided and erased. The RW log functions in the same fashion as presented with FMAX where pages are entered sequentially, in an out-of-place fashion utilizing the OOB area to store the logical page address.

Consider the following example in Figure A.1 where each block contains 4 pages and the system has two RW log blocks where the operations *write(4,A)*, *write(6,B)*, *write(4,C)* and *write(6,D)* proceed. The logical block number and page number are calculated using Equations (3.1) and (3.2) respectively using four pages per block. For the first two write operations (Figure A.1a), the target pages do not contain any data so the writes proceed directly on the target pages in physical block 0. For the third operation of *write(4,C)* the target page is already allocated so the update must proceed in either the SW or RW log-block. As the logical offset of the target page as computed by Equation (3.2) has previously been utilized but nothing has previously been written into the SW log-block, the page will be written in-place in the SW log-block with the LPN being recorded in the OOB area. The original target page in physical block 0 will be invalided to indicate that a more current version of the page exists in the log. After the write to the SW log-block, the next valid entry must have a logical page offset of 1 as it is the next sequential page to be written. For the fourth operation of *write(6,D)* the target page has already been allocated so it will also be written to either the SW or RW log-block. As the logical page has an offset of 2, it does not satisfy the write preconditions for the SW log-block. The next page that can be written into the SW log-block must have an offset of 1. As a result, this page cannot be written into the SW log-block. The operation will proceed to write the updated page into the RW log-block. In order to track the pages that exist, a RW log-block mapping table is maintained in SRAM that maintains page mappings between logical page numbers and physical page numbers. The page will be logged in physical page 8 and the RW log-block mapping table is updated accordingly. In the OOB area of the logged page (physical page 8), the logical page number is recorded and the original target page is invalidated.

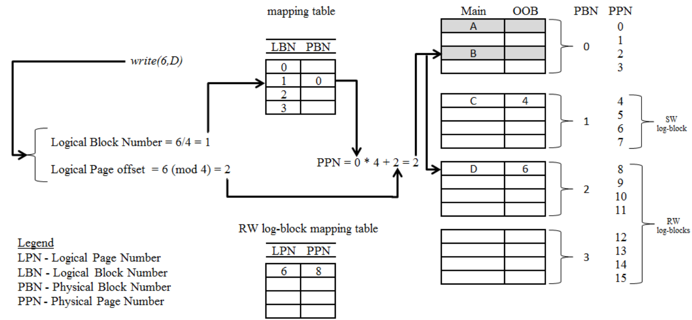
Once the SW or the RW log-blocks are completely filled, they will be merged with the original target pages and new log blocks selected. For the



(a) FAST writing to Unallocated Pages



(b) FAST writing to the SW log-block



(c) FAST writing to the RW log-block

Figure A.1: The FAST FTL Scheme

RW log-block merge this may involve merging numerous pages due to the many-to-one relationship between data blocks and the RW log-block. The worst case scenario is that every page is mapped to a different block which will cause a chain of erase operations [KRKC11]. For the SW log-block merge, as the data is written sequentially for a specific block, FAST will convert the page using the same strategy as with BAST to being a live block through a switch merge where the log block becomes the live data as all data has been written in-place in the log block.

The significant difference between BAST and FAST is that FAST maintains sector level mapping for logs which are stored in SRAM. For systems with significant SRAM resources, this additional overhead may be a reasonable trade-off with performance compared to BAST, but for memory constrained systems, this significantly reduces its affordability. Similar to BAST, FAST maintains a dedicated block-level table in flash and a map-block directory in SRAM. Sector mapping is strictly maintained in SRAM and must be rebuilt on remount and does not consider any recovery options. Other limitations in terms of performance exist with the SW log-block. As there is only one SW log-block allocated for the system at a given time, a sequential operation may start to utilize the SW log-block. If another series of sequential writes were to start they would be blocked from using the SW log-block and be forced to utilize the RW log-block, impacting the overall performance of the FTL algorithm. As a result, the FAST scheme may suffer a performance degradation depending on the specific data access patterns as noted previously.

EAST

Kwon and Chung present EAST [KC08] in response to the poor space utilization encountered with the FAST scheme. EAST attempts to address this issue through the use of state transition tracking at the log block level allowing for more efficient allocation of blocks. They claim significant performance increases over FAST. Utilizing strategies from both FAST and BAST, the EAST algorithm allows a 1 to n mapping between target blocks and log blocks. It is fundamentally a block mapping scheme and hence uses Equations (3.1) and (3.2) to calculate the logical block number and physical page offset for any logical page. The differentiating factor with EAST is that it supports both in-place (FAST) and out-of-place (BAST) updates to the log blocks in an effort to increase log-block utilization. Based on this technique, Kwon et al. [KRKC11] claim that a log block will be fully utilized before allowing a merge to proceed. Unlike with FAST, one notable difference from

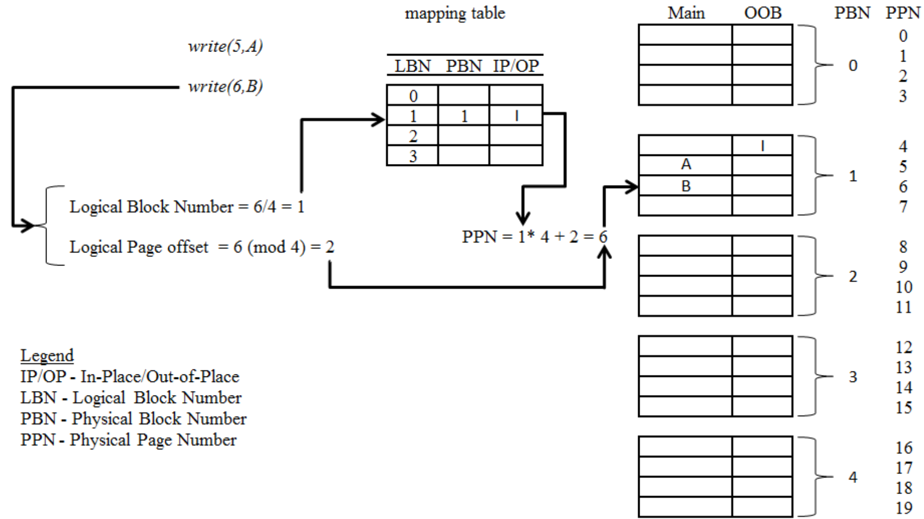
previous methods is the addition of an in-place/out-of-place state encoded in the block mapping table. This is used to determine if the target block is using in-place updates or has been converted to a log using out-of-place updates and is intended to increase block utilization before erasure.

Initially, a physical block will have associated with it a given number of log blocks. Writes to the target page will proceed in an in-place fashion as long as the target pages have not been previously allocated. When a page is to be written that has previously been allocated, the target page is converted to support out of place updates and becomes a log type block. Additional log blocks are allocated to support overflow before a merge operation is required. Essentially, the technique supports in-place updates on the chance that data will be written in a sequential fashion but failing that, converts to a logging system similar to FMAX with the exception that logged data is no longer restricted to a single block.

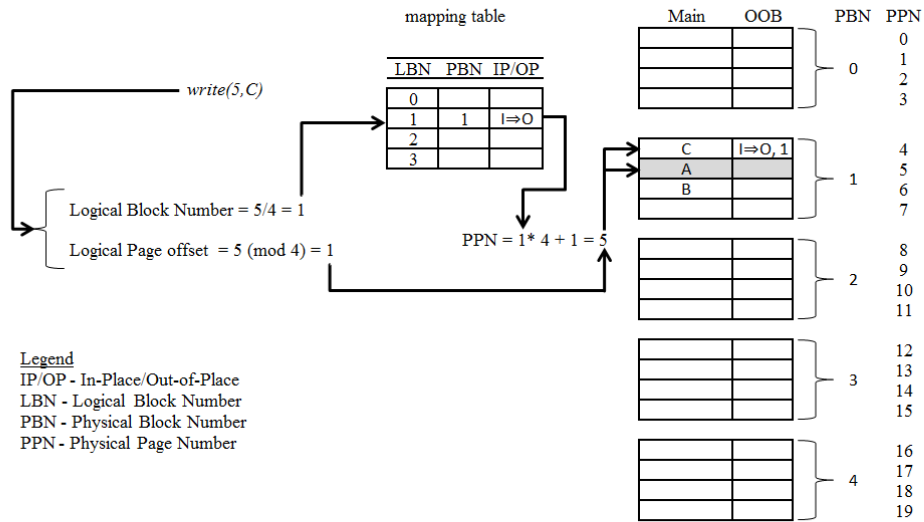
Initially, log blocks start out strictly with the in-place structure where pages are updated at their corresponding logical offset allowing for fast access and updates. If a page in the log is to be overwritten, the state of the entire block is changed such that it will only support out-of-place updates and functions in a logging fashion similar to BAST. Once a block has made a transition from in-place to out-of-place, it will stay as an out-of-place block until it is full and needs to be merged with the original page. This allows log pages to be fully utilized before being sent for erasure.

Consider the following example in Figure A.2 where each block contains 4 pages where the operations *write(5,A)*, *write(6,B)*, *write(5,C)*, *write(6,D)* and *write(5,E)* proceed. The logical block number and page number are calculated using Equations (3.1) and (3.2) respectively using four pages per block. For the first two write operations of *write(5,A)* and *write(6,B)* (Figure A.2a), the target pages do not contain any data so the writes proceed directly on the target pages in physical block 1 as allocated by the block mapping table. Additionally, logical block 1 is marked as being in-place as well as the in-place state being recorded in the OOB area of the physical block.

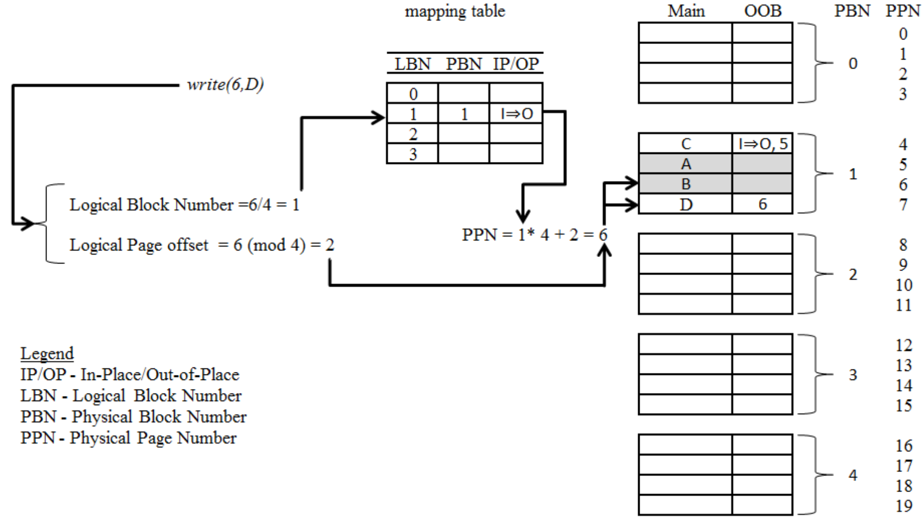
When the third operation (*write(5,C)*) proceeds, the system will determine that the page has previously been allocated and cannot be written in-place. As a result, the block will be converted to support only out-of-place updates as shown in Figure A.2b. The state of the block will be changed in both the block mapping table and in the OOB area of the physical block. The target page will be invalidated and the algorithm will start scanning at the start of the block looking for the first unallocated page. When one is encountered, the data will be written to the page and the logical page number recorded



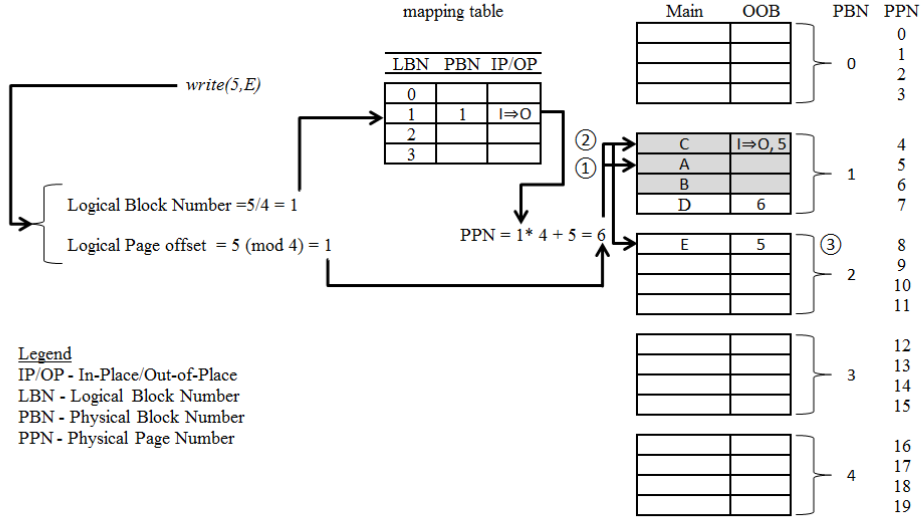
(a) EAST Writing to Unallocated Pages



(b) In-Place to Out-of-Place Conversion



(c) EAST Writing Utilizing Out-of-Place Writes



(d) EAST Adding a Log Block

Figure A.2: The EAST FTL Scheme

in the OOB area. In this case as the first page is available, *write(5,C)* will proceed to store data at that location.

For the forth operation (*write(6,D)*) as the block has been converted to supporting only out-of-place updates, the algorithm scans forward locating the target page, invalidates the page and then continues to scan for the next available page (Figure A.2c). In this case, physical page 7 is the next available page that can accommodate data. For the last write operation, the algorithm determines that there are no pages left in the first page. Thus, it will attempt to determine if it can allocate another physical block to be added as an out-of-place log block. The number of blocks allocated to a data target block as log blocks is a precalculated performance based parameter as discussed below. If the system is able to allocate a free data block (also referred to as an "reallocation"block), the write will proceed. In Figure A.2d, the system allocates physical block 2 as a log block. The algorithm will invalidate the target page and then proceed to write the data to the first available page in the new block. If the algorithm determines that a block can not be allocated for the log, it will proceed with a merge operation, compacting all data in the original target and log block to a new data block and erase the invalid pages.

Kwon and Chung compared the performance of EAST with FAST with a variety of access patterns and on average, it performed better due to the convertibility of log blocks. No performance comparison was done against BAST or FMAX. Under most circumstances, a single page re-write will force a log block to convert to support out-of-place updates and essentially become the BAST type scheme. The uniquely differentiating feature of EAST is that the number of logs blocks per target block is configurable depending on the host specifications. To improve performance, the authors choose the number of logs blocks such that the time to erase one log block is less than the time it takes to scan all the pages in the log. This ensures that the scan time for out-of-order updates will be bounded by the time it would take to create a new log block using the FAST scheme. In terms of log block conversion, the merge operation operates similar to BAST with the conversion being completed when the log block is full and cannot support any further out-of-place writes. EAST supports the functionality of a switch merge where the log page is the valid live copy of the entire page.

While it is targeted at small block memory devices, the authors assert that this is a transportable system to large block devices. This claim presents technical challenges as noted in Section 2.2.2, as large block devices can only support sequential updates within a block. As a result, only the out-of-place updates would be performed, which renders EAST into the same operational

state as BAST with the only difference being the number of log blocks associated with each target block.

Other challenges exist with EAST, particularly for memory constrained devices. EAST tracks the number of erases per block in SRAM which introduces significant overhead. Based on erase count, it determines which block to use next as a log block. When a new block is allocated, it must scan the erase count list to find the block with the lowest erase count which introduces significant overhead. Additionally, the mapping table is also stored in SRAM. Depending on the size of the flash memory and the number of logs, this presents a feasibility barrier for memory constrained devices. Kwon and Chung claim that EAST offers significant advantages over previous methods; while the system can adjust the number of log blocks such that the timing trade-off between scanning and block erasure is balanced, it still imposes significant limitations, as the FTL must be maintained in SRAM as well as the erase block count. Additionally, no performance comparison was made with BAST which essentially offers the same access patterns for non-sequential updates and does not offer any recovery options [PCK+08].

LAST

Lee et al. [LSKK08] propose *LAST*, an FTL that is more suited for general purpose computing than for resource constrained devices. In general, previous works have shown that a general-purpose FTL does not offer good overall performance. They make the claim that it is appropriate to have an FTL for a specific target and data set. LAST attempts to address the issues that were found with BAST [KKN+02], FAST [LPC+07] and Superblock FTL [KJKL06] (Section 3.4.4) to improve overall performance in terms of merge costs, access patterns and the separation of hot and cold data performance issues. It is a general purpose FTL that essentially combines three different FTLs to address three different access patterns that may be encountered. The key feature is that it examines a request and determines if it is a sequential or random type pattern. Its key differentiating attribute is the division of the log block area into sequential and random access patterns. Once it determines if it is a sequential or random write, it will redirect the data to the specific area for each type of data. Sequential data is handled in a similar fashion as with BAST. Sequential writes to a log will be efficiently switched to data block for the minimal cost using a switch-merge. Random data will go to the random log buffer where a log block can be associated with many data blocks, similar to FAST.

LAST also attempts to divide data into hot and cold sections in the

random block area which helps to reduce the cost of a full merge. Locality is inferred from the size of the write. If it is a small write it is considered to have high temporal locality. Conversely, large writes are considered to have low temporal correlation [LSKK08]. In reality, the size of the partition depends largely on how the host operating system has divided or partitioned data and can influence the hotness of data. The authors note that this performance can be impacted by a preset level threshold that determines the difference between hot and cold data. They note that if the threshold values are tuned incorrectly, LAST may suffer significant performance degradation. Lee et al. [LSKK08] note that under small size random writes such as may be encountered with a resource constrained device, log block utilization is poor leading to a high number of merges. This log block thrashing [LPC+07] increases the overall cost of merges due to number of reads, writes and erasures.

Overall they claim that by exploiting temporal locality as well as hot/cold correlations in data they can reduce the cost of the full merge improving the overall performance of the flash translation layer. They also assert that their features can reduce the number of erase operations based on the partitioning of data. The performance increases realized by FAST are due to the reduction in erasures; which is claimed to be up to 54% improvement over BAST, FAST and Superblock FTL [KRKC11] (Section 3.4.4).

JFTL

JFTL [CLP09] is designed to improve the performance of journal-based file systems such as ext3 by exploiting a journal remapping technique. The unique feature of JFTL is that all data is written into a new region as an out-of-place update. In place of using data (D) and update (U) blocks, JFTL maintains a home location for the data and then a journal space where all updates are made. While a block mapped strategy, data is journaled on a page level. This prevents the significant numbers of overwrites and eliminates redundant data. It is suitable for append type operations. With journaling file systems, data written to the main data section is first written to a log which prevents overwrites. Additionally, it adds support for atomic type operations, redundancy and fault tolerance. Instead of rewriting mutated data from the logs to flash memory, JFTL converts or remaps journal (log) pages to become data pages, thus effectively reducing the need for read/write operations as well as eliminating the need for redundant data [DZ11]. As a result, the journaling file system will always be left in a consistent state.

Like other FTLs, JFTL includes an erase strategy. As pages become

obsolete, they are marked for deletion. Unlike previous works, JFTL attempts to mark pages that will soon be erased in advance of the erase option thus eliminating the need for an unnecessary move. The authors claim that this strategy reduces the number of erase operations incurred through merge operations. JFTL presents a recovery and remount strategy in the event of a system failure. On rebuild, JFTL scans the entire data section to look for transactions that are incomplete and performs journal remapping within each mapping table. This operation repeats until it finds the end of the journal which can be either good or corrupt. Once the system has rebuilt the journal remapping tables, the system is reset. The recovery speed is dependent on the size of the journal. It is asserted that the recovery speed is faster than with a standard FTL, but the authors fail to indicate which FTL it is being compared against and how other FTLs perform recovery operations.

While JFTL offers significant performance enhancements, it is designed to use with a journaling file system and a system with significant SRAM resources. While journal based strategies may prove to be an overall useful strategy for flash memories, integration with a journal based file systems renders it an unsuitable choice for use in a memory-constrained embedded system.

SAFTL

SAFTL [Wu10] is a self adjusting FTL with a smaller memory requirement than BAST and a low erase strategy, but it has a complicated four stage collection strategy to select victim pages. This strategy uses both coarse-grained and fine-grained memory mapping strategies that are maintained in SRAM. The unique contribution is the ability to switch between mapping schemes depending on the type of writes. For large chunks of contiguous data, a coarse-grained mapping structure is used while a fine-grained hash map is used for fast access with a self adjusting mechanism to control the number of fine-grained slots available.

On average, it performs better than BAST on page reads due to the fine-grained translation scheme. In terms of overall address translation performance, it performs better than BAST but in some cases its performance was worse due to swapping of the fine-grained translation. This requirement leads to a complicated and involved reload policy with a high data transfer overhead limiting its suitability for resource constrained devices as well as a large SRAM memory footprint. While SAFTL demonstrates good performance for huge scale flash memories, it is infeasible for small memory

devices due to the limited bandwidth channel between the host and memory. The authors also fail to address issues of performance in recoverability.

A.1.2 State Based FTLs

The follow section discussions further variations on state based FTLs.

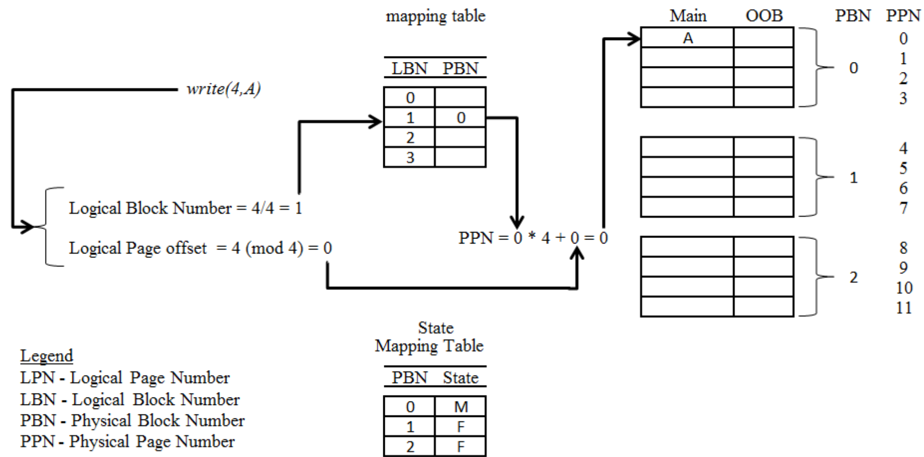
STAFF

The STAFF algorithm [CP07] (Section 3.4.5) utilizes partial page programming to reduce block erasures in addition to increasing block utilization via a state machine that encodes the state of each page in the out of band area of a given block. STAFF allows the encoding of free, obsolete, modified in-place, complete in-place, and modified out-of-place states in the OOB area of a block allowing for pages to be modified within a block without the block having to first be erased.

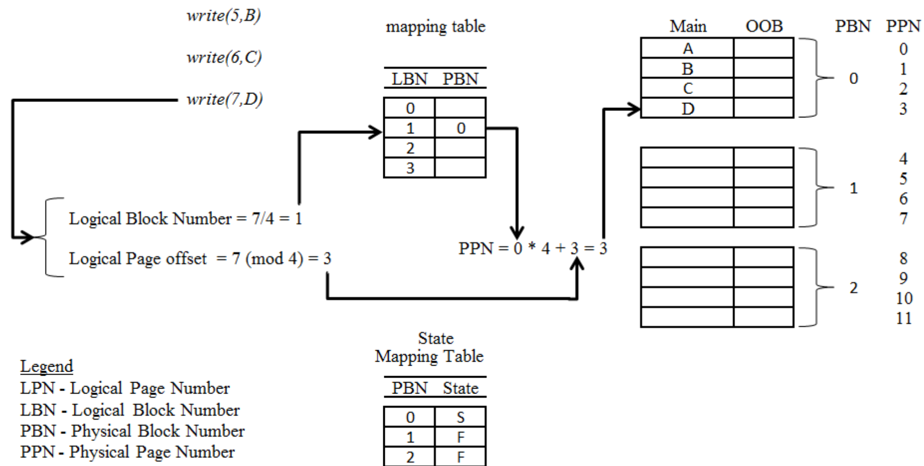
Consider the two following examples with the first following the state changes for in-place operations and conversions (Figure A.3) and the second following the state changes for out-of-place operations (Figure A.4), where each block contains 4 pages. The logical block number and page number are calculated using Equations (3.1) and (3.2) respectively using four pages per block. For both cases, consider that the system contains only free blocks (F state).

To demonstrate the in-place operations, consider the operations *write(4,A)*, *write(5,B)*, *write(6,C)*, *write(7,D)*, *write(4,E)*, *write(5,F)*, *write(6,G)*, *write(7,H)*. When the first operation *write(4,A)* is processed, the physical block and logical page offset are calculated (Figure A.3a). The system resolves the state of the block from the state mapping table, determines that the block is in a free state and proceeds to write the directly to the calculated logical page offset. The state of the block is changed from F to M state in the state mapping table to indicate that the block contains only in-place data. For the next three operations of *write(5,B)*, *write(6,C)*, *write(7,D)*, the process proceeds in a similar fashion as all blocks can be written in-place (Figure A.3b). After the last write operation, the block is completely filled and can no longer have data added to it. Since all the pages contain live data and are in-place, the state of the block is changed from M state to S state in the page mapping table. This allows the system to correctly index data on reads directly and not have to perform a linear scan to find target pages.

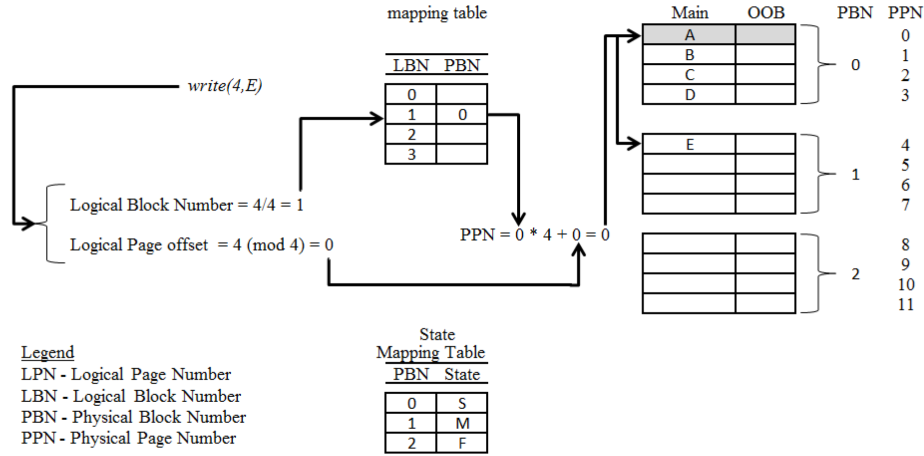
For the next operation of *write(4,E)*, the system checks the state of the



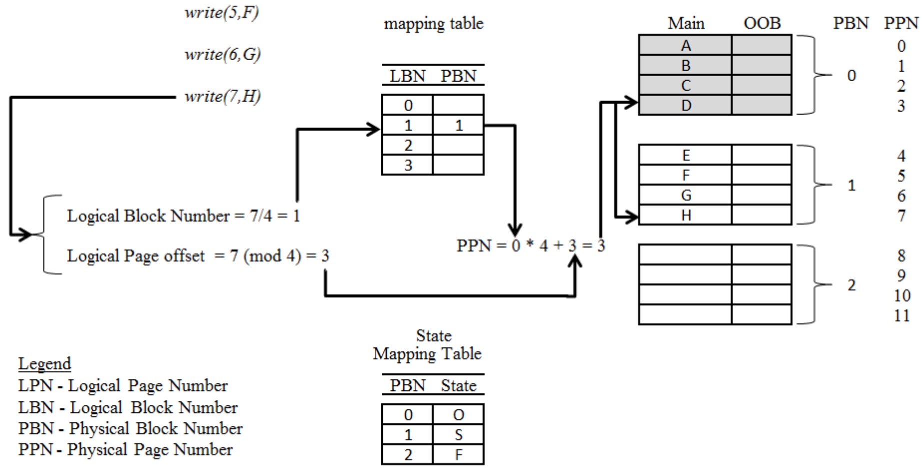
(a) STAFF writing to Unallocated Pages (M-state)



(b) STAFF Completing an In-Place Block (S-State)

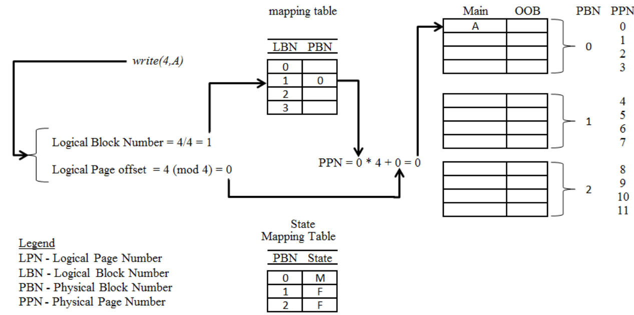


(c) STAFF Allocating an In-Place Log Block (M-State)

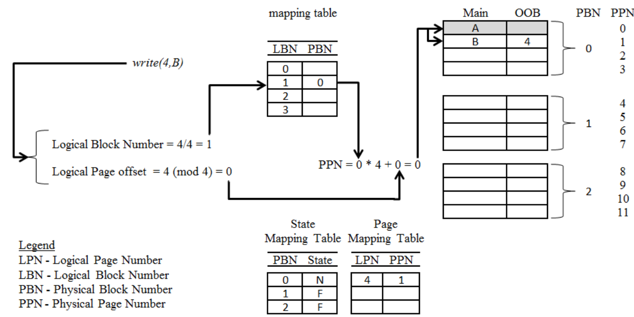


(d) STAFF Releasing a Block (O-State)

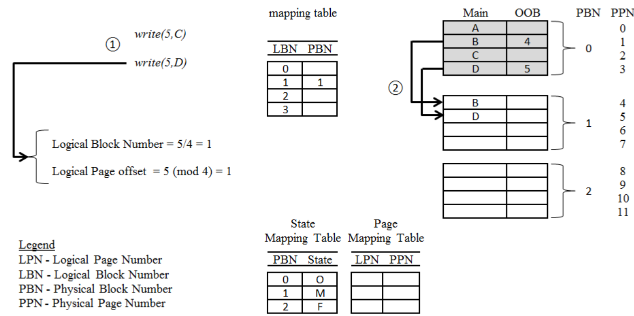
Figure A.3: The STAFF FTL Scheme for In-Place Operations



(a) STAFF writing to Unallocated Pages (M-state)



(b) STAFF Converting to an Out-of-Place Block (N-State)



(c) STAFF Allocating an In-Place Log Block (M-State) For N-State Block

Figure A.4: The STAFF FTL Scheme for Out-of-Place Operations

physical block and determines that it is full thus it needs to allocate a new block for the buffer (Figure A.3c). Unlike other FTL schemes, STAFF does not need to delineate between log and data blocks due to the variable state of each block and thus refers to added blocks as buffers. In this case, physical block 1 is allocated and as the block in the F state, the system can directly write the target page at the correct physical offset and invalidate the original target page in physical block 0. The state of the new block in the state mapping table is changed from F to M to indicate that it now contains in-place updates. The state of the original target page is unchanged as it still is complete-in-place with live data. It will remain in the state until all pages in the block become invalid (Figure A.3d). As the operations *write(5,F)*, *write(6,G)*, *write(7,H)* proceed, the same process continues where the data is invalidated in the target page and the data can be written in an in-place fashion to the buffer block. Once the buffer block is completely full using in-place updates, its state transitions from M to S in the state mapping table. It is now the complete in-place live version of the block. As a result of this, the block mapping table entry for logical block 1 is updated to point to physical block 2. Finally, as the original target block contains only invalid pages, the state of the block is changed in the block mapping table from S to O to indicate that it is obsolete and can be erased by the system. As a result of the state transition, the buffered block which is the shadow for the original target block is able to supersede the original data block when it becomes invalid without having to go through a merge type operation.

To demonstrate the out-of-place operations, the operations *write(4,A)*, *write(4,B)*, *write(5,C)*, and *write(5,D)* proceed. When the first operation *write(4,A)* is processed, the physical block and logical page offset are calculated (Figure A.4a). The system resolves the state of the block from the state mapping table, determines that the block is in a free state and proceeds to write the block directly to the calculated logical page offset. The state of the block is changed from F to M state in the state mapping table to indicate that the block contains only in-place data for the first page write. For the next operation of *write(4,B)* (Figure A.4b), the page will attempt to write the first logical page in block one as the state of the block is M, but cannot as the page is already occupied. As a result, the state of the page will be changed from M state to N state, allowing the block to be updated in an out-of-place fashion. The state of the page will be updated in the state mapping table and the system will then perform a linear scan of the physical block from logical page offset of 0 looking for the first empty page. Once an empty page is located, B will be written to the page and the logical page number will be updated in the OOB area. In order to facilitate better access

to N state blocks, STAFF maintains a page mapping table in SRAM for all N state blocks that maps logical page numbers to physical page numbers.

As the block has now been converted to the N state the operations will proceed in an out-of-place fashion and be written into the next available free pages with the last write being the most current version of logical page 5 as shown in Figure A.4c ①. As a result, the page at the logical page offset of 4 (the last free page filled) will contain the correct logical page number in the OOB area. After the N state block is filled the system will allocate a new buffer block (block 1) and copy live pages from the original target block into the new block (Figure A.4c ②). The state of block 1 is then updated in the state mapping table from F state to M state as the valid pages were entered into the new block at the correct logical offsets thus allowing the page to be in order. Additionally, the state of the original target block is then changed to O state as it can be erased and the physical block number for logical block 0 is updated from 0 to 1 to reflect the new mapping.

LSTAFF and LSTAFF*

LSTAFF [CPRH05] was first proposed by Chung et al. in 2004 as a large block extension of the early presentation of STAFF [CPJK04]. The scheme uses the same state machine proposed by STAFF to track the state of each block. They observed that with large block devices, the page size was often larger than that of the sector size dictated by the host operating system. LSTAFF assumes that it will be operating under a host operating system.

Unlike small block devices where logical sector size maps approximately to page size, with large block devices, device data is typically written in 1 kB to 4 kB chunks; this represents multiple logical sectors. Large block devices also have significant write constraints where pages within a single block must be written in sequential order. Most FTLs do not satisfy this constraint and will not work with large block devices. Additionally, this feature of large block flash will prevent other log block-based strategies from using out of place updates such as RNFTL [WLW+10].

LSTAFF uses a three level mapping scheme to allow operating system sectors to be mapped to logical pages on device. They store the mapping area in the OOB area as well as the state information for each block. As a result of the three level mapping, a degree of parallelism could be achieved on a read or write basis. As multiple sectors are stored or read from a single page in a large block device, the LSTAFF scheme exploits this parallelism. The authors offer a cost estimation analysis to show the potential gain from using this technique with large block devices.

LSTAFF* [CPK11] is an extension of LSTAFF [CPJK04, CPRH05, CP07] which has been further optimized for large block flash memory. It assumes that the host operating system will now have a many to one mapping between the logical operating system pages and physical device paged as is the case with large block devices. Similar to LSTAFF, LSTAFF* assumes that it will be operating under a host operating system.

One unique feature that LSTAFF and LSTAFF* offers is parallelism in terms of page reads and writes due to the large block mapping. As reads or writes will generally require the entire large block to be brought into the SRAM buffer, physically adjacent pages to the target page will be simultaneously buffered. If the adjacent pages are spatially or temporally adjacent, the host system may achieve additional benefit. While the page level parallelism offers significant increases for enterprise level systems, it offers little value for memory and bandwidth constrained devices that may not be able to buffer a single large block page due to limited memory. For a memory limited device, multiple pages would need to be stored in SRAM then flushed to the device. This introduces significant overhead in terms of space and transfer time thus making this strategy infeasible.

Similar to STAFF, LSTAFF* uses a state machine to track the state of cache pages, which presents significant overhead. Further, it offers no advantage for devices that do support out of place updates. Linear probing is required to determine and change state tables. Additionally the in-order write restriction reduces overall utilization This also increases complexity when merging log data blocks to enforce in-place rights.

LSTAFF* maintains a cache for data, thus exploiting a feature of large block flash memory that allows for the outputting of random data from within a page. This is achieved through the use of the memories internal buffer. LSTAFF* will then gather and reorganize pages based on state to improve performance. While Kwon et al.[KRKC11] attribute this buffering feature to the initial LSTAFF paper [CPRH05] in 2005, the buffering and random access concept was only presented in the LSTAFF* paper [CPK11] in 2011.

LSTAFF and LSTAFF* were compared under simulation to STAFF and FAST but only consider performance in terms of the number of writes. The strategy is not suitable for embedded devices due to the overall memory constraints. They did not consider the overhead of the state machine, as well as the increased complexity of the algorithm, or the impact on wear levelling, garbage collection and block utilization

A.2 Comparison of FTLs

Table A.2 summarizes the presented FTLs and highlights the type of mapping scheme used for each, as well as the key contribution of each algorithm in addition to any special requirements.

Algorithm	Type	Target Memory	Special Requirements	Key Contributions
DAC	Fully Associative Page Level Scheme	NAND		Data(D) and Update(U) block separation; seminal work that presents update blocks to minimize erase count
DFTL	Fully Associative Page Level Scheme	NAND	Enterprise SSD	Caching of mapping pages in memory as opposed to maintaining entire page map in SRAM.
CFTL	Fully Associative Page Level Scheme	NAND		Caching of mapping pages in memory as opposed to maintaining entire page map in SRAM. Tracking of hot/cold data to control access patterns.
LazyFTL	Fully Associative Page Level Scheme	NAND	Enterprise SSD	No merges required due to page mapping scheme, lazy page mapping replacement strategy for cached pages in SRAM using LRU.
Mitsubishi	Block Level Scheme	NAND		Seminal block level scheme; logical page number stored in OOB area. Introduced merging operations.

ANAND	Block Level Scheme with Page Based Logging	NAND	Device must support non-sequential writes at the block level	Separation of physical blocks into data and log blocks to minimize erase operations.
FMAX	Block Level Scheme with Page Based Logging	NAND	Device must support non-sequential write at the block level	Separation of physical blocks into data and log blocks to minimize erase operations. Treats log block as a linear log to which reduces the number of merges.
BAST	Block Level Scheme with Block Based Logging	NAND	Targeted at Compact Flash market. Device must support non-sequential write at the block level	Introduces a one-to-many between log and data blocks which reduces the number of log blocks required in the system. Uses block level mapping for data blocks but page level mapping for log blocks. Introduces map blocks that are specifically used to store mapping information and are brought into memory using an on-demand caching model.
FAST	Block Level Scheme with Block Based Logging	NAND		One log block can be shared between many data blocks. Separate logs for random and sequential writes.

EAST	Block Level Scheme with Block Based Logging	NAND	Device must support non-sequential write at the block level	Combines attributes of BAST and FAST, supporting both in-place and out-of-place updates in log to improve log block utilization. Lowest erase count victim block selection.
LAST	Block Level Scheme with Block Based Logging	NAND	For general purpose computing due to large resource requirements.	Combines FAST, BAST and Superblock FTL schemes and dynamically selects based on data access patterns. Separation of hot/cold data.
JFTL	Block Level Scheme with Page Based Journaling	NAND	For using with journaling file system and large SRAM.	All updates are written to journal using page level mapping which reduces the number of overwrites before being written to data section. Journaled pages are converted to data pages to minimizing read/write operations.
SAFTL	Self Adjusting Mapping Scheme	NAND	Large SRAM requirement	Suitable for large memories with variable write characteristics.
Superblock FTL	Block Set Scheme	NAND	Designed for Large Block Devices	Combined adjacent blocks into larger Superblocks. Using page level mapping within Superblock which is stored in OOB area. Uses FAST type scheme for mapping Superblocks.

STAFF	State Based FTL utilizing a Block Level Mapping Scheme	NAND	Device must support non-sequential writes at the block level and partial re-writes. Large SRAM requirement as both state maps and block maps are maintained in memory. Targeted for general purpose computing	Individual page states are encoded in OOB area of block allowing for more efficient use of block. State block mapping is maintained in SRAM. Improved performance over other algorithms due to low cost swapping and merge operations.
LSTAFF	State Based FTL utilizing a Block Level Mapping Scheme	NAND	Designed for Large Block Devices were flash page may be larger that operating system sector.	Three level mapping scheme stored in OOB area along with state information. High degree of parallelism possible for adjacent operating system sector writes. Supports Large Block Devices with sequential write constraints.

LSTAFF*	State Based FTL utilizing a Block Level Mapping Scheme	NAND	same as LSTAFF	Improves performance of LSTAFF as it maintains a data cache though the internal memory buffer.
PORCE	Recovery Scheme for Block Based FTLs		Will only work with FTLs that use in-place updated	Recovery strategy for Block Based FTLs.
Reuse-Aware NAND FTL	Block Reuse Strategy	NAND	Device must support non-sequential writes at the block level	Minimizes erase operations by analyzing the utilization dirty blocks and will reuse dirty blocks with low utilization as log blocks.
JanusFTL	Self Adjusting Mapping Scheme (Hybrid)	NAND	Targeted for SSD. Device must support non-sequential writes at the block level	Allows for dirty page reuse. Self adjusts to specific workload patterns.

ShiftFlash	Data Consistency Scheme	NAND	Targeted for SSD. Requires multi-chip storage device for inter-chip wear levelling and garbage collection.	Allows for high level roll-back through the use of snapshots. Enforces sequential writes through buffering in SRAM
AVR		NOR	AVR32 core only	Pre-compiled for AVR32 32-bit core only. Required large SRAM memory footprint. Closed source.

Table A.1: Summary of FTL Schemes for Flash Memory

Appendix B

The FlaReFTL Interface

The following section details energy public interface for the FTL.

B.1 FTL Instances

Listing B.1 shows the main instance structure for the management of a single instance of the FTL. When an instance of the FTL is created, the values are populated. The `masterBlock` is used to track physical location of the active master translation page. The `timestamp` parameter is used to track and stamp write operations to the translation pages and is used for recovery operations. The `leveller` parameter is used to manage and access the wear levelling module of the code. The `writeMode` parameter controls overwriting operations.

Listing B.1: Control Structure for Flare FTL.

```
typedef struct FlareFTL{
volatile block_address_t      masterBlock;
volatile flare_timestamp_t    timestamp;
volatile wearLeveler_t        wearLeveler;
volatile flare_byte_t         writeMode;
}flare_FTL_t;
```

An instance of the `flare_FTL_t` struct is created in code by the user and is required to be passed to all operations of the FTL.

B.1.1 System Control Functions

The following functions are public and are used for the mounting, unmounting and creation of system.

function FLARE_MOUNT(volatile flare_FTL_t *instance)

Mount the drive and create the cache buffer for FTL. Creates the cache buffer for FTL if it is needed depending on strategy and rebuilds and mounts FTL. On a successful mount, the instance variable is populated with the pointers required for operation.

return The status of the mounting operation.

end function

function FLARE_UNMOUNT(volatile flare_FTL_t *instance)
Free all resources of the FTL. Flushes any buffered data and commits keystone pages, leaving a consistent system.
return The status of the unmounting operation.
end function

function FLARE_CREATEFTL(volatile flare_FTL_t *instance)
Initializes a serial NOR Dataflash device with Flare FTL. Formats the device and creates initial management pages. On successful creation, the instance variable is populated with pointers for the logical busy page, physical busy page and master table translation page.
return The status of the creation operation.
end function

B.1.2 Page Management Functions

The following functions are public and are used for the allocation and destruction of logical pages in the system. This operation allocates both a physical and logical page for the user. Updates metadata in OOB area of a page with the correct LPN.

function FLARE_GETPAGE(volatile flare_FTL_t *instance, flare_logical_page_t *lpn)
Allocates a logical and physical page in the system and populates the lpn parameter with its logical address as an integer value.
return The status of the allocation.
end function

function FLARE_RETURNPAGE(volatile flare_FTL_t *instance, flare_logical_page_t logical_page_number)
Returns a logical page to the available pool of pages.
return the status of the return operation.
end function

B.1.3 Read Functions

The following functions are public and read a series of bytes from a given location.

function FLARE_READPAGE(volatile flare_FTL_t *instance, flare_logical_page_t logical_page_number, void *buffer)
Reads the entire page at the given logical address into the user supplied buffer.
return The status of the read page operation.
end function

function FLARE_READBYTES(volatile flare_FTL_t *instance, flare_logical_page_t logical_page_number, void *buffer, flare_offset_t offset, flare_length_t length)
Reads a series of bytes from logical page at logical address with a given offset and length into the user supplied buffer.
return The status of the read bytes operation.

end function

Private Read Functions

The following functions are private and are used by the FTL.

function LOADLOGICALPAGE(int logicalAddress, char bufferNumber)
Loads a logical page into specified buffer in preparation for writing.
end function

B.1.4 Write Functions

The following functions are public and are used for writing data to a logical page in the system.

function WRITEBYTES(int logicalAddress, char * data, int length, int offset, char commitLevel)
Writes byte* of size length to offset in logical page to the specified buffer.
The *commitLevel* will determine if the page will be written with record level consistency or page level consistency. If using record level consistency, the logical page stored in the buffer will be immediately flushed to the flash block. If using page level consistency, the page will not be flushed until it is full or a pageWrite operation is specified.
end function

Private Write Functions

The following functions are private and used by the FTL.

function WRITEPAGE(int logicalAddress, char bufferNumber, char writeMode)
Writes a page out from the specified buffer to flash block at the corresponding logical addresses. The *writeMode* parameter is used to control if the logical page will be written an a new location in memory or use the proposed low energy append-in-place write.
end function

B.1.5 Buffer Management Functions

The following functions are private and are used to manage the buffers on the serial NOR Dataflash device.

function BM_REQUESTBUFFER(void)
  Allocates an on device buffer for use by the system (used strictly for writing data)
  return The buffer number allocated.
end function

function BM_RETURNBUFFER(char bufferNumber)
  Returns the specified buffer number to the buffer pool for reuse.
end function
function BM_INSTANTIATE(void)
  Initializes the buffer manager.
end function

The buffer manager is responsible for managing the state of the SRAM memory buffers on the serial NOR Dataflash. The buffer manager abstract the allocation of the buffers from the FTL.

Listing B.2: Control Structure for Flare FTL.

```
typedef struct buffer_page{
        flare_page_type_t          type;
                        /** type of page that is stored */
        flare_timestamp_t          timestamp;
                        /** the timestamp of the page */
        flare_logical_page_t       lpn;
                        /** the lpn of the page, if need be */
        flare_physical_page_t      ppn;
                        /** the ppn of the page, if needed */
        bm_buffer_status_t         status;
                        /**< The status of the buffer */
} bm_buffer_page_t;
```

In the buffer manager, each buffer maintains information about the page that has been stored in it (Listing B.1.5. When a page is buffered, the type of page that is being stored is recorded as well as the timestamp of when the page was last written which is generated by the FTL. Additionally, the physical and logical page numbers are recorded as well as the status of the page in the buffer. The buffer also maintains a timestamp for each page in terms of when it was last used and is controlled by the buffer manager and used for buffer selection. Each buffer is permitted to have the following states:

buffer_active The data in the buffer is the same as what is stored in the corresponding page in flash.

buffer_active_written The data in the buffer has been changed from from what originally loaded.

buffer_flushed The data in the buffer has been flushed to flash but is still active.

buffer_invalid The state of the data in the buffer is uncertain.

The consumer of the buffer is responsible for updating the status and timestamp values for each individual buffer page when conducting operations.