1-1-2014

Study On Endurance Of Flash Memory Ssds

Mochan Shrestha
Wayne State University,

Study on Endurance of Flash Memory SSDs

by

Mochan Shrestha

Dissertation

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfilment of the requirements

for the degree of

Doctor of Philosophy

2014

Major: Computer Science

Approved by:

_____

Advisor Date

_____

_____

_____

DEDICATION

To my Family

Half the World Away

ACKNOWLEDGMENTS

I would like to express my deep gratitude to my PhD advisor Dr. Lihao Xu. I feel very lucky and privileged to have his guidance, help, knowledge, patience and support during the course of my PhD. I deeply appreciate his contributions in time and energy on me and I have benefited greatly from it.

I would like to thank Dr. Nathan Fisher, Dr. Song Jiang and Dr. Hongwei Zhang for serving on my prospectus and dissertation defense committee. I am very proud to have such wonderful people and researchers in my committee.

I would also like to thank my friend and lab mate Jianqiang Luo for always being willing to help, always willing to lend a ear and that wonderful and sharp analytical mind to our discussions.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

1.1 Problem Description

Flash memory is a low cost, solid state, non-volatile memory that can be electrically programmed and erased [15, p. 1-33]. Flash memory is widely used in electronic devices like cell phones, flash drives, embedded systems and increasingly in personal computers, laptops, servers and data centers where it is complementing [83], enhancing [46, 109] or displacing [98, 77] magnetic hard disks. Since flash memory doesn't have any mechanical components and only electronic components, it has the inherent advantages over electro-mechanical magnetic hard disks of shock resistance, low heat, power and noise levels, and low read latency. Flash memory thus provides rugged and reliable storage for mobile and embedded devices. For servers and data centers, it enables massive IO performance improvements, alleviating the CPU-storage bottleneck that severely hampers most system performance [7].

One of the important pillars of modern computing revolution has been the mass storage system. Revolutionary technologies like web search, massive databases and data centers would not be possible without storage technologies like Google File System[45], Hadoop etc that have been build upon the basic storage technology of inexpensive magnetic hard disks. Magnetic hard disk technology has enabled storage of massive amounts of data that can be quickly retrieved, processed and stored which has enabled revolutionary technologies to be built upon them.

The great benefits of the modern storage systems have only been possible because of the engineering around the limitations of the magnetic hard disk, and minimizing the

effects of its limitations and negative aspects while maximizing the utility of its positive aspects. Magnetic hard disks have always been thousands of times slower than the computer processing systems and a computer system can spend the majority of its time waiting for the IO system to respond. Magnetic hard disks are fragile and spin at high speeds; mechanical shocks can easily destroy them and are prone to sudden complete random failures. Storage systems based on them have to be built with high redundancy to insure against data loss. Nonetheless, despite its numerous limitations, we have been able to build impressive technologies based on this fundamental technology by engineering around its limitations.

NAND flash-memory SSDs [88, 57] have multiple orders of magnitude faster access times than magnetic hard disks because they are an all-electronic storage medium with no mechanical moving parts. This property gives flash memory the advantage of no delays from mechanical latency and thus, data requests can be satisfied in microseconds compared to milliseconds that it takes for magnetic hard disks. Additionally, flash memory is also available in capacities of hundreds of gigabytes at relatively low cost; a price point that no other types of all-electronic solid-state storage mediums are available in. Because of these desirable characteristics of flash memory, it promises to be the technology for the next generation of storage devices providing orders of magnitude superior performance.

Flash memory has already revolutionized storage in mobile devices like cell phones and rugged devices like factory equipment. The small size factor, the ruggedness and the high capacity for a low price has enabled flash memory to be the primary storage technology of the mobile revolution. For computer storage systems, the main advantage of flash memory is its very low access time which is multiple orders of magnitude faster than mechanical hard disks. Therefore, flash memory based storage systems could potentially be orders of magnitude faster, much more rugged against failures and data loss and only take up a fraction of the space and energy to operate than current magnetic hard disk based storage systems. Thus, it has the potential to fundamentally change and revolutionize modern

computer storage systems and such new storage systems could be the bedrock of many future revolutionary computer applications.

However, even with its many desirable properties, flash memory has not been able to be utilized to its full potential in storage systems because it possesses fundamentally different characteristics than magnetic hard disks [106]. Foremost, flash memory has not been a direct replacement for magnetic spindles in disks because using it efficiently requires considerable engineering around its unique characteristics. For example, data management methods are completely different in flash memory. Simple operations like updating data in flash memory needs a multitude of IO operations and supporting software architecture to perform the operations efficiently. So, flash memory's usage requires considerable architectural analysis, engineering and design before it can be used efficiently enough to provide the improved performance it promises.

The first fundamental difference between flash memory technology and magnetic disks is the lack of in-place updates in flash memory. This comes about because of the physics of how flash memory stores data. It uses a program operation to store data and when the data needs to be updated, it must first be erased before the new data can be reprogrammed in that spot. This is in contrast to magnetic hard disks where data can be simply overwritten at the same location. Additionally, to make a large number of storage cells in the chip, the sizes of the program and erase operations are different. Data is programmed in chunks called pages whereas data is erased in blocks comprising of 16-128 pages. This asymmetry in the size of programs and erases leads to the phenomenon called write amplification.

When an erase is performed on a block, it is likely that the block contains some valid data that cannot be discarded. To ensure that this valid data is not lost, it has to be read into memory before erasure and then after erasure written back. This step is called *block cleaning* or *block reclamation*. This internal copying back of data leads to a much larger number of IO operations than those issued by the user. The measure of the amount of

data copied back internally through this mechanism is called *write amplification* [53].

Another basic characteristic of flash memory is the limited programming cycles of flash memory cells wherein a flash memory cell can only be erased 10,000 - 1,000,000 times before it wears out and cannot reliably hold data [29]. Since some areas of the disks are written to more often than others, it could result in uneven wear and worn out areas, leading to a reduced functional lifespan of an SSD.

These are the fundamental properties of flash memory and any system using flash memory must deal with block cleaning, write amplification and uneven wear. In storage systems, the view of flash memory to the user is a linear array of updateable sectors. In order to provide this view of flash memory to the user, there is a layer of software that is used called the flash translation layer (FTL). It translates user requests to flash memory commands and manages the data stored. A common design for the FTL is to have data is written out of place, i.e. when data is rewritten, it is not written to the location where the data was previously stored but is written in a new location determined by some strategy. This design eliminates the need to erase blocks before every update but necessitates a background garbage collector erasing blocks to free up space. Thus, in addition to providing a disk interface to flash memory, the FTL also manages wear leveling and write amplification. It attempts to keep the wear of the blocks as even as possible and to minimize the number of extra system operations because of internal copybacks.

This raises the questions of how data can be stored and managed to reduce system usage. To do this, we need practical algorithms and strategies to manage wear leveling and write amplification. Theoretical bounds to what is actually possible for an FTL to achieve in terms of write amplification reduction also has to be investigated so we can understand the effectiveness of our practical algorithms. Though capacity and performance is often the primary attributes of flash memory systems, every flash memory system must manage wear leveling and write amplification. These are fundamental attributes of flash memory and will always need to be understood and efficiently performed to produce an efficient

and high performance storage system.

One useful property of flash memory is that any piece of stored data can be accessed with the same constant access time. Thus, we can store data at any location in the flash memory without it affecting the performance. Hence, we can utilize the placement of the data to reduce write amplification by grouping similar data together. We call the management of data locations in flash memory *layout management* and the algorithms that calculate the best location for each data sector *layout management algorithms*. We explore both the theoretical and practical sides of this. On the theoretical side, we explore the limits of reduction in write amplification and the absolute minimum write amplification possible using layout management. On the practical side, we present simple yet effective algorithms that reduce write amplification. We have implemented and tested these algorithms in our own event-based SSD simulator.

While the layout management is used to decide where to write data, the garbage collector decides which block to erase to reclaim data with the aim for both low write amplification and for wear leveling. We model and create mathematical equations that give the wear distributions for different configuration of the SSD, the wear leveling strategy and the workload distribution. This serves as a useful tool to study the properties of wear leveling against the different factors that affect it. We also give interesting practical algorithms that are very effective in performing wear leveling even when the garbage collector is tuned to reduce write amplification.

Multi-level flash memory cells are capable of storing data in more levels than zeros and ones. In such cells, electrons can be added to the cell to increase the value stored but cannot be removed to decrement the stored value. For such cells, we can use a coding scheme where data is encoded as flash cell increments. This coding scheme was introduced in [70] as floating codes. We give a new algorithm to construct floating codes that are optimal and show that these constructed codes form a sub-class of floating codes called update codes that have additional properties.

In this dissertation, we investigate algorithms in flash memory, both theoretically and practically, of the fundamental aspects of efficiently storing data in flash memory to enhance endurance through wear leveling and reducing write amplification. Using coding and layout management we attempt to minimize the amount of internal operations. Using mathematical models and practical algorithms, we study how to arrange data over time so that the wear is as even as possible. These form the foundation of every flash memory based storage system and every system design must tackle these aspects of flash memory. Our goal is that these investigations into the fundamentals can serve as a solid building block for future systems.

1.2 Contributions and Significance

The contributions of this dissertation can be roughly categorized as in the following four areas.

- We look at update codes that provide a method of constructing optimal floating codes. Floating codes encode data in multi-level flash memory so that data updates can be encoded as flash memory cell increments.
- We look at a theoretical view of write amplification in relation to layout management and its possible limits to its minimization through the analysis of offline workloads.
- Next, we present and analyze practical algorithms that can reduce write amplification and provide wear leveling that are simple and efficient yet effective.
- Finally, we provide a quantitative model of flash memory wear so that we can analyze the wear patterns of flash memory cells mathematically.

1.2.1 Update Codes: A New Class of Floating Codes

When flash memory cells are erased, there is a large negative impact on the longevity and performance of the device. To defer and minimize these erasures, coding can be used to encode data updates as cell increments.

Our work provides a new algorithm for constructing optimal floating codes. Additionally, these constructed optimal floating codes form a new class of floating codes we call update codes. They are useful class because they can be thought of as a coding scheme to emulate a small group of cells with a high number of possible updates from a large number of flash cells that have a small number of updates. Additionally, the floating codes from update codes are isomorphic to their non-binary floating codes. Thus, finding optimal binary variable update codes gives us optimal floating codes in any l-ary variables and not just binary variables.

Specifically, we present update codes where an (n, q, k) update code uses an array of n cells with q levels to simulate k flash memory cells while maximizing the number of possible updates before erasure. Update codes are a subset of floating codes and have more structure and so, are easier to construct and additionally, the binary floating codes from update codes are isomorphic to non-binary floating codes.

We also investigate the poset (partially ordered sets) structure of the code and from this, we construct update codes for arbitrary q, k and $n \in \{k, k+1\}$, or arbitrary n, q and $k = 2$ that is optimal for single cell increments. We present an algorithm for constructing the update code and prove that the algorithm does produces a valid code. The corresponding floating code can be derived from the update code and is also optimal for single cell increments and has a deficiency of $O(qk)$, the best possible deficiency.

1.2.2 Minimization of Write Amplification

The excess internal copybacks from write amplification causes degradation in performance and longevity of the flash memory device. Here we explore the fundamental limits of write amplification reduction using data placement techniques called layout management algorithms to find the minimum value of write amplification possible.

A question regarding flash memory systems is that if we removed the abstracting layer of the FTL between the user and the flash memory, would it be possible to eliminate problems of SSDs like write amplification? The FTL is very restrictive in the information about the workload it can utilize and if the FTL is removed then all the workload information could be used to maximum efficiency. We use offline workloads simulating perfect knowledge and information about the workload, and explore the limits of write amplification reduction and minimization. This effectively looks to answer the question: is write amplification is an inherent property of flash memory, or an artifact of the flash translation abstraction, or an effect of inefficient implementation of the FTL?

First, we prove that zero write amplification for any workload and flash memory configuration is not possible and the worst case over-provisioning for zero write amplification is the trivial over-provisioning where we have as many blocks as sectors.

Next, we explore the question of finding the minimum write amplification when we have an offline workload and flash memory configuration. We present an algorithm for the estimation of the minimum write amplification based on our technique of decomposition of workloads. The estimation algorithm is useful because it provides a way to explore the inherent characteristics of flash memory and write amplification.

From our experimental results, we find that write amplification can be reduced to zero for moderate over-provisioning ratios of around 0.3-0.5. This hints at the notion that for high over-provisioning, write amplification is a property of flash memory but for lower over-provisioning, it is an artifact of the FTL abstraction. However, our work raises many

open questions and unsolved problems and so the answer is not fully solved but we do provide numerous algorithms and techniques for layout management which gives some idea on the minimization of write amplification.

1.2.3 Practical Designs and Algorithms for Reducing Write Amplification and Wear Leveling

Write amplification is a parasitic drain on flash memory performance and endurance. Reducing write amplification has positive effects on most aspects of flash memory like improving IO performance, wear and longevity.

Here we explore a method of reducing write amplification from data of varying volatility, for example cold and static data with low volatility, that create write amplification by being copied back multiple times over many garbage collection cycles. We use the technique of copying back different parts of the valid data in the block selected for erasing, to different copyback blocks depending on the number of times it has been previously copied back.

When data is being copied back blindly, a phenomenon called *sedimentation* occurs. Low volatility data settle at the low pages of a block evenly throughout the SSD and every copyback operation ends up copying back the low volatility data repeatedly. Our technique of separate and multiple copyback blocks eliminates sedimentation and decreases write amplification.

Our low overhead algorithm is very efficient and effective at moving data of different volatility to different blocks without requiring a-priori knowledge of the data or requiring predictive algorithms. No workload models or statistics have to be collected and calculated making this a very efficient and simple algorithm but it is effective as well. Though write amplification reduction and wear leveling can be conflicting goals, we provide a simple garbage collection algorithm such that we can achieve both wear leveling and write amplification reduction using copyback blocks.

Through simulation, we show that we can significantly reduce write amplification and improve wear leveling, and that these significantly improve SSD performance.

1.2.4 Quantitative Framework for Modeling Wear Leveling

To increase endurance and reliability, flash memory based devices employ certain wear leveling algorithms. But so far there hasn't been a rigorous mathematical tool to analyze and evaluate the effectiveness of various wear leveling algorithms.

We present a mathematical framework to model whole block wear leveling based on probabilistic models of workloads, wear leveling strategies and wear level states. From this framework, we derive equations for the distribution of wear levels and the effectiveness of wear leveling algorithms for various SSD configurations. Using these equations, we give a quantitative analysis and evaluation of baseline wear leveling algorithms for flash based devices.

Since the SSD abstracts the flash memory into an array of sectors and the user has no direct control on choosing specific blocks or pages to write data on, all blocks in the flash memory have an identical probabilistic wear distribution. Thus, by modeling a single block in the flash memory, we can effectively model the entire flash memory's wear patterns. This solves the problem of other models having to calculate probabilities for each block and the interaction probabilities among every other block. Thus, we can create a wear model that is easily applicable to any size of flash memory and that the size of the flash memory does not make the problem grow exponentially.

The quantitative framework gives an SSD designer a method to investigate wear leveling under various configurations and derive optimal design, and also serves as a starting point for modeling complex SSD systems.

We calculate the wear distributions as difference equations with workload, configurations and strategy as parameters in the equations. We numerically calculate a few wear distribution curves from our difference equations and investigate how the wear varies with

the various parameters.

1.3 Organization

In Chapter 2 we give the background on flash memory and SSDs and the related work and research on the topic. In Chapter 3 we present update codes and discuss the its constructions and its properties like optimality and relation to floating codes. In Chapter 4 we present the techniques we have developed to find the theoretical minimum possible of write amplification by using offline workloads. In Chapter 5 we present practical online algorithms that reduce write amplification and perform wear leveling. In Chapter 6 we provide the details of the design and implementation of the SSD simulator that we wrote to test and refine our practical algorithms and which was also used to produce the experimental results. In Chapter 7 we discuss modeling of wear and wear leveling strategies for flash memory and present our quantitative and mathematical model of wear leveling. In Chapter 8 we provide our conclusions and an overview of our visions and future work in this field.

CHAPTER 2

Background and Related Work

2.1 Flash Memory

Unlike hard disks which uses magnetic properties to store data, flash memory uses electron trapping in floating gate transistors. The key technology in flash memory is the floating gate transistor which is used to store the data bits in flash memory cells. The floating gate is a conductive island surrounded by an insulation layer and storing data bits into a cell amounts to trapping electrons inside the floating gate, which is called programming. To read the data in the cell, the electric field of the trapped electrons in the floating gate is detected which is called sensing. To reuse the cell, the electrons must be first discharged from the floating gate which is called erasing. To provide gigabytes of storage, flash memory contains billions of memory cells printed in chips along with the related circuit to sense, program and erase the electrons inside the floating gates. In this way, floating gates building blocks are used to construct the multi-gigabyte flash chips.

Even though the data storage on hard disks and flash disks are transparent to users, the characteristics of these two storage mediums are markedly different [6]. The main differences [43] are

1. *Read Latency, Throughput, Power Usage, Heat, Noise Levels and so on*

   This is the main advantage that flash memory has over magnetic hard disks. Flash memory does not have any mechanical parts and only circuits. Thus, read latency is lower. Power consumptions, heat and noise levels are also lower. Using efficient error correction, throughput of the disk drive can be high.

2. *Read/Write Asymmetry*

Magnetic Hard Disk	**Flash Memory SSDs**
Consists of sectors that are of size 512 to 1024 bytes	Consists of blocks that are of size 32 KB to 128 KB
Has two operations **Read Sector** and **Write Sector**	Has 3 operations **read block**, **write block** and **Erase block**. Some NAND allows writing to pages inside a block
Read and write cost is symmetrical	Read is fast, write is slower and erase is slowest.
Sectors or blocks do not wear out	After $10^5 - 10^6$ erase cycles, the blocks become unusable.

Table 2.1: Difference between Magnetic Hard Disks and Flash Memory SSDs

In flash memory, byte-level writes or programming can only occur in erased cells or pages. However, erasing must be in done at the block level. Blocks can be as large as 256 KB. It is a type of write once and bulk erase medium. It is also referred to as not being able to write in place.

As given in [19], the measured asymmetry is given below.

	Read	Write	Erase
Performance (μs)	348	919	1881

3. *Limited Programming Cycles*

   A flash cell can only be erased about 10,000 - 1,000,000 times before it wears out and cannot reliably hold charge anymore.

The differences between hard disk and flash memory is summarized in Table 2.1.

2.1.1 Variants of Flash Memory

NOR flash is an extension of EEPROM (electrically erasable and programmable ROM). It was initially designed as execute in place memory: meaning that code could be executed directly from NOR flash without having to have it copied to RAM. NOR flash is byte addressable and programmable but erasing and writing is very slow [91].

NAND flash is designed specifically for data storage. It has a higher erase cycle range and faster erase and write times. However, it is not byte addressable and is sequential access up to the given block size (512 bytes and over). This is in contrast to NOR flash which can be used in the same manner as SRAM. However, it should be noted that even magnetic hard disks have similar amount of sequential access. NAND flash is also manufactured with the knowledge that a certain portion of the blocks are bad and it is left to the software to manage the bad blocks and not use them. This makes the manufacturing simpler and more cost effective than having to make sure that each block is operational.

To lower the cost per bit of NAND flash memory, more chip area is allocated for memory cells and the amount of overhead circuitry reduced by organizing the memory cells into pages and blocks. A page is the smallest read and write unit comprised of 2^{15} to 2^{16} adjacent cells connected to the same word line. A block is the smallest erase unit comprised of 32 to 128 pages sharing a common source line connection. The cell, page and block structures in flash memory are illustrated in Figure 2.1 where a row represents a page and a collection of pages between source lines represent a block. Thus, pages and blocks are the fundamental structure of flash memory where read and programs operations occur in pages while erase operations occur in blocks.

2.1.1.1 Multilevel flash memory

Multi-level flash memory can store more values than 0 and 1 in each cell e.g. [100], [99]. Since it is not possible to remove charge from flash cells, the programming of flash cells is only in 1 direction (i.e. the cell values can only be increased). Since this involves careful electron placement into cells, the reliability of programming can be low. In this case, error correcting needs to be employed to reliably use multilevel flash memory. One of the drawbacks of multilevel flash memory is that is has erase cycle limits of only 10,000 compared to 100,000 - 1,000,000 for single level flash memory since the window has to be

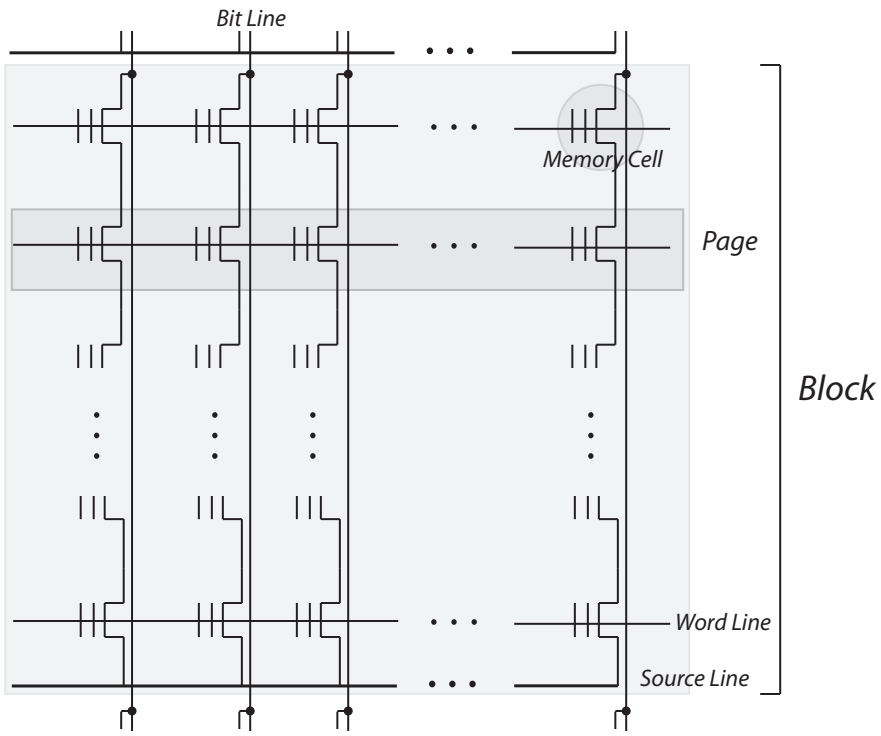


Figure 2.1: Cell, Page and Block in Flash Memory

expanded for the additional bit values.

This type of memory is called write asymmetric memory (WAM). If the cell is a q level cell, it takes values of $0, 1, \ldots, q-1$ and programming can only change the value from a lower to a higher state.

MLC flash cells have their own hardware characteristics [33] and is an active research field for better designs and materials. However, the following observations have been made about MLC flash

1. Reduced reliability because of higher electric fields required for read, write and erase.

2. Reduced read speed

3. Longer programming cycles and increased disturbs during programming.

2.1.1.2 Flash Packages

Flash packages are commercial products that contain flash memory and a micro-controller for operating and communicating with flash memory inside it. It consists of one or more flash dies stacked together for data storage and numerous pins to transmit and receive commands and data. Flash packages can fully perform read, program and erase operations without external control and are designed to be easily integrated into boards to add storage functionality. Thus, products using flash memory for data storage build on commercial flash packages, one such being SSDs.

2.2 Solid State Disks (SSDs)

A solid state disk (SSD) is usually NAND flash memory combined with a DRAM buffer and a controller. The controller manages the flash and the buffer, and provides a communication interface to be used with an existing interface standard.

To be used as a disk drive in systems, the IO interfaces require the disk to operate as a linear array of addressable sectors and assume an in-place update capability where old data is overwritten by newer data at the same location. However, in-place updates are extremely inefficient for flash memory and thus, for efficiency it is emulated in the software layer running on the SSD controller CPU known as the flash translation layer (FTL). To emulate in-place updates, an out-of-place mapping system is used where each sector is mapped to a physical page via a mapping table. When sector updates occur, they are written to a brand new page and the mapping tables updated so that the sector address now points to the newly written page. In this way, the software FTL and hardware components of SSD together make it possible to use flash memory in computer storage systems.

The FTL also serves many other functions besides emulating in-place updates. The FTL also translates IO requests to flash memory commands, performs garbage collection

of invalid data known as block cleaning, and wear leveling to maximize the life of the drive. FTL is thus, a complex layer between the computer system and flash memory chips simplifying its usage.

2.2.1 Wear Leveling and Efficient Reclamation

Though flash memory can be used as a ROM device where the operating system and other read only files are stored, we will look at flash memory as a general purpose main memory storage. The direct method of storing data (i.e. data for block 1 is stored in flash block 1) is not efficient and presents the problem of wear-out of cells as certain blocks are written to more often than others in a typical computer use [42].

Furthermore, if we are to use flash memory as we use magnetic hard disks by mapping the virtual blocks to flash blocks, we would have the following problems:

1. Once a block has been written then any update to the block would require reading the block, erasing the block and then writing back to the block. This would result in a very slow performance and high energy usage.

2. Since some files are updated more often than others, for example log files and tables for directory structures, this would lead to wearing out of certain parts of flash memory. This would then lead to a low lifetime of the memory.

The counter the above two problems, we will need to adopt the following two techniques:

2.2.1.1 Efficient Reclamation

As flash memory is used over time, some data gets obsolete or the user requests that it be deleted from the flash memory. Thus with time, the available memory is now used up as obsolete or invalid data is still stored in the flash memory and thus, some blocks in the flash memory need to be erased to obtain clean pages for new data. For most flash file

systems, this is done by a garbage collector. Since erases are time and power consuming, and reduces the lifetime of the cell and techniques must be used to minimize the number of erasures.

On top of erasing, reclamation also involves the following:

- Valid pages of the block must be copied somewhere else before the block is erased. Thus, the garbage collector must choose the reclamation victim properly so that data isn't just being copied around needlessly.
- The data structures for the file system that point to the block must be updated and the block identified as an erased block for reuse as free space.

2.2.1.2 Wear Leveling

Wear leveling [49] refers to prolonging the life of flash memory by avoiding erasing one block of flash much more than other blocks. Thus, all blocks in the flash memory will have been erased an equal number of times and the wear of the flash memory is equal over the entire flash disk.

The count for the number of times a cell has been erased can be written onto the header of the block. However, most garbage collection methods do not use erase counts and rely on random or sequential placement of data that algorithmically promotes wear leveling.

2.2.2 Block Mapping

Instead of directly using flash memory blocks as is, to address wear leveling and inefficient erasing, a block mapping technique [43] can be used. The host sees a *virtual sector number* which are mapped to flash blocks and pages. When new data is written, it does not overwrite the old block but writes to some other flash block and page and updates the virtual sector number to reflect the new physical block and page location. We will call

the virtual sector number to flash block and page map the *direct map* and the physical page to virtual block as the *inverse map*.

Sooner of later, we will run out of empty blocks and have to start garbage collecting. Note that block mapping also helps in wear leveling since multiple writes to the same virtual block is written to a different flash block.

As sizes of flash memory grows, it has become inefficient to store large direct and inverse maps in RAM. One technique is to do block level mapping. However note that this is also inefficient if only a few bytes are updated in each block. Then, a lot of bytes have to be read and written to a new block. To alleviate this problem, we use caching and another solution is to use log-structured based file system. Another issue is that the size of the block in NAND memory is increasing to make it more cost effective to use large values. Thus, as technology of flash changes what was once the optimal method may no longer be optimal and a new optimal solution may have to be proposed.

2.3 Related Work

A mathematical model for wear leveling problem [44] was presented in [5]. The limitation of their approach is the lack of the ability to mark blocks as invalid and to do garbage collection. Research in wear leveling algorithms have explored reverse engineering algorithms in commercial SSDs [8], reducing wear leveling operational cost by summarizing block information [72] and designs for moving rarely updated static data proactively [21], through dual [18] or multiple [62] queues. Studies on flash endurance have put forward the notion that current estimates of cell endurance are inaccurate [29] and that current models of endurance are too simplistic [123] where recovery periods could increase endurance [96]. Thus, there is a need for an analytical wear leveling model that incorporates modern SSD characteristics that can be used to assess proposed wear leveling algorithms and also address the newer models of endurance.

Analytical models for estimating and calculating write amplification under simple assumptions are given in [53, 12] and its limits [52] and the impact of garbage collection on it [58]. Current models suffer from state explosion where calculating write amplification for practical sized SSDs take impossibly long. There is a need for a simple model of write amplification that can be calculated for large SSDs which can also be easily manipulated to test new designs and workloads.

Performance modeling for flash memory has been done by restricting the workloads. Poisson based workloads on a single chip SSD analysis has been done in [13]. Other performance modeling research has been on real time flash systems, and workload experiments and metrics. Real time garbage collection analysis for flash memory was given in [20] but it assumes that the CPU polls and waits for flash operations to complete. This is especially restrictive for multi-die SSDs which can have concurrent SSD operations. Models that are aware of the CPU execution and device IO operation was proposed in [76] but does not consider the characteristics of flash memory like block erase and read-/write asymmetry. Real time file system design has been proposed in [61] where the uncertainty of garage collection delays is prevented by replication. Workloads and how it affects the SSD system including metrics [9], experimental results on real life workloads [103] and performance [23] have also been explored. Thus, firstly there is a need for real time flash memory model with modern SSDs assumptions and an extension to deal with parallel flash operations. Secondly, the work then has to be extended for QoS assumptions since the worst case delays by garbage collection operations severely under-estimate the performance capabilities of flash memory SSDs.

Simulation of SSDs have been primarily focused on augmenting DiskSim [10], the disk system simulator. By adding an SSD model to Disksim, simulations can now be performed for the storage system where a magnetic hard disk is replaced by an SSD. An object oriented framework for modeling the internal structures of a flash memory is given in [82]. Beyond just modeling the SSD, the various tradeoffs of flash design elements

including data placement and parallelism, wear leveling and workloads is given in [1]. Another simulation study was done exploring the bottlenecks, organizations and facets of SSD architecture in [32, 31]. All the simulators created for flash memory suffer a design deficiency that is carried over from the design of hard disk system simulators like DiskSim in that the simulator cannot reorganize, reorder and parallelize executions of the IO operations. Disksim sends an event to the disk model and waits for the completion of the event before proceeding to the next event. However, due to the highly parallel nature of SSDs, events are started and completed out of order and executed in parallel to each other. The internal garbage collection and wear leveling mechanisms further send their own IO requests which run alongside the user driven IO operations. Thus, there is a need for a simulator that is able to capture the parallelism of the SSDs and the independent nature of the controller in the SSD for accurate and insightful simulation.

2.3.1 Practical Designs and Approaches

While our approach is to use mathematical models and simulation to validate new algorithms and designs, there is a lot of research that simply make use of commercial SSDs to build storage systems like data center based applications of SSDs [17], low power clusters [2], hybrid methods with magnetic hard disks [115, 24, 83]. Custom made SSDs have been created to test flash properties and algorithms like endurance [8]. For specific applications like databases [86] there has been research into new designs for page logging [85] and transactional flash [105], for file systems new designs like DFS [71] or existing Linux file systems [118, 94]. Flash memory can substantially improve performance in all computing applications and specialized designs and algorithms for various applications are limitless. As capacities of SSDs have become large there has been a need for efficient new data structures and algorithms to manage data and several versions of B-trees have been proposed [89, 75].

2.3.2 Coding Techniques

Floating codes were first introduced in [70] as a generalization of the WOM model [108], and an optimality bound and constructions of optimal codes for arbitrary n, q, $k = 2$ and $l = 2$ were given. These floating codes were generalized for arbitrary k in [113] with a deficiency of $O(k^2q)$ and later improved upon in [122, 92] to a deficiency of $O(qk \log k)$. Floating codes for n, q, k and $l = 2$ called indexed codes were given in [65] which are asymptotically optimal but imposes some restrictions on n, q, k. Further bounds and more example codes for some values of n, q, k were given in [64]. Floating codes called covering codes for $l > 2$ were given in [65]. Less restrictive forms of floating codes were studied in [34, 26, 35].

A more general problem of coding for flash memory has been studied in [67, 63, 66, 68] and have led to related codes like rank modulation [69], buffer codes [122, 64] and WOM codes [120, 121, 16].

2.3.3 FTL Designs

Commercial SSDs have their own FTL designs to interface between the flash memory and the storage system, and thus, is a heavily patented area. Patents have been filed for refinements to the block reclamation methods [37], data organization [51], garbage collector [101], configurability of the controller [22] and many others. Other patents include new methods for caching data and managing the cache [36, 4], using flash memory as a cache[93], power failure behaviors and techniques [39], dealing with issues of concurrency in data operations [41], design techniques for managing operations like a data pipeline [38], error correcting for failing regions of flash memory [40], using reference cells for monitoring errors from charge leakage [97], managing errors in multi-level flash memories [116], using LDPC codes for error detection and correction [125] and many others.

CHAPTER 3

Update Codes

3.1 Introduction

Flash memory is a widely used non-volatile storage technology that operates by trapping charges inside a floating gate. An array of millions to billions of flash memory cells can be put on a chip to enable storage of large amounts of data. Additionally, the cells can be used in a variety of different ways so that data storage can be optimized with different applications in mind.

Charge can be added incrementally to a cell but to release the stored charge, an erase operation has to be performed. Repeated erasures break down the cell insulation and thus, after erasing a cell certain number of times, it is no longer able to hold charge and store data. Additionally, erasing can only happen simultaneously on a large group of cells called a block and any valid data in the block needs to be copied somewhere else. On top of that, compared to other flash memory operations, the erase operation is a few orders of magnitude slower. Thus, erasing has to be avoided as much as possible in order to increase the longevity and performance of flash memory devices.

To reduce and defer block erasures, a coding scheme called floating codes have been proposed [70]. In this coding scheme, an array of variables and its updates is stored as various states of flash memory cells and variable updates are encoded as cell increments. Thus, each time data is updated it does not require erasing the flash memory cells but only requires cells to be incremented.

Here, we present update codes which are closely related to the floating codes. The general idea of update codes is that we want to encode a block of flash cells so that it can

emulate a smaller number of cells but with higher update levels and with a set number of total updates possible across all the update cells. In other words, suppose we use a block of n flash memory cells with q levels, we use update codes to represent these cells as k update cells on which we can perform a total of t updates where $k \leq n$ and $t \geq q$.

The main practical importance of updates codes is that they are equivalent to a subset of floating codes. Thus, if we find an update code, we also have an equivalent floating code. The benefit of going through update codes is that it allows for more structure and thus a more convenient way of finding codes. Additionally, non-binary floating codes for $l > 2$ can be easily constructed from binary floating codes derived from update codes giving a large number of optimal codes by solving the binary floating code problem.

We first explore floating and update codes in terms of posets (partially ordered sets) where each update of a cell or variable provides a natural ordering for a poset. The cell increments produce a cell poset, the cell updates an update poset and the variable updates produce a variable poset, which have different poset structures. In Section 3.3, we describe properties of the posets and the fundamental structural differences between the posets. Mainly, we show that two cell state vectors can only have one cover while variable state vectors have no such restriction. This property underlines the fundamental challenge to constructing floating codes because each variable state vector has to be represented by multiple cell state vectors. However, update state vectors also have at most a single cover and are very close in structure to cell state vectors and thus, update codes are simpler to construct.

In Section 3.4, we give a construction for a t_1-optimal update code. We call a code t_1-optimal if it can do $(n-k+1)(q-1)$ variable updates each with single cell increments (the fundamental limit of floating codes). We first give an algorithm for the floating cell construction and then prove that the algorithm produces a valid floating code. In Section 3.5, we prove that non-binary floating code ($l > 2$) can be constructed from binary floating codes and provide an algorithm for it. Thus, the problem of finding floating codes reduces

to just finding binary floating codes that are derived from update codes.

This construction is a novel way of creating update and floating codes and achieves the optimal bound for single updates for arbitrary q, k and $n \in \{k, k+1\}$, or arbitrary n, q and $k = 2$. The deficiency of the code, which is a measure of closeness to optimality, is $O(kq)$, the best possible deficiency achievable. The best known floating code for arbitrary n, k, q has deficiency $O(qk \log k)$ [122, 92] but note that our code covers a smaller set of parameters while having optimal deficiency.

3.2 Floating and Update Codes

3.2.1 Flash Memory Architecture

Data is stored in flash memory cells where each cell can be at q different levels $\{0, 1, \ldots, q-1\}$. Each cell can be incremented up to level $q-1$ but then cannot be decremented. The only possible way to reuse these cells, once the data stored in them are no longer needed, is to erase a block comprising of hundreds of thousands of these cells. After a block erasure, all the cells in the block have level 0.

3.2.2 Flash Memory Representation

Given n cells which have q levels, we call the vector

$$[c_0, c_1, \ldots, c_{n-1}]$$

the cell state vector and $C^{n,q}$ the space of all state space vectors of length n and q levels.

Given $\boldsymbol{c} = [c_0, c_1, \ldots c_{n-1}], \hat{\boldsymbol{c}} = [\hat{c}_0, \hat{c}_1, \ldots, \hat{c}_{n-1}] \in C^{n,q}$ we say that $\boldsymbol{c} \leq \hat{\boldsymbol{c}}$ if $\forall i \in \{0, 1, \ldots, n-1\}$,

$$c_i \leq \hat{c}_i$$

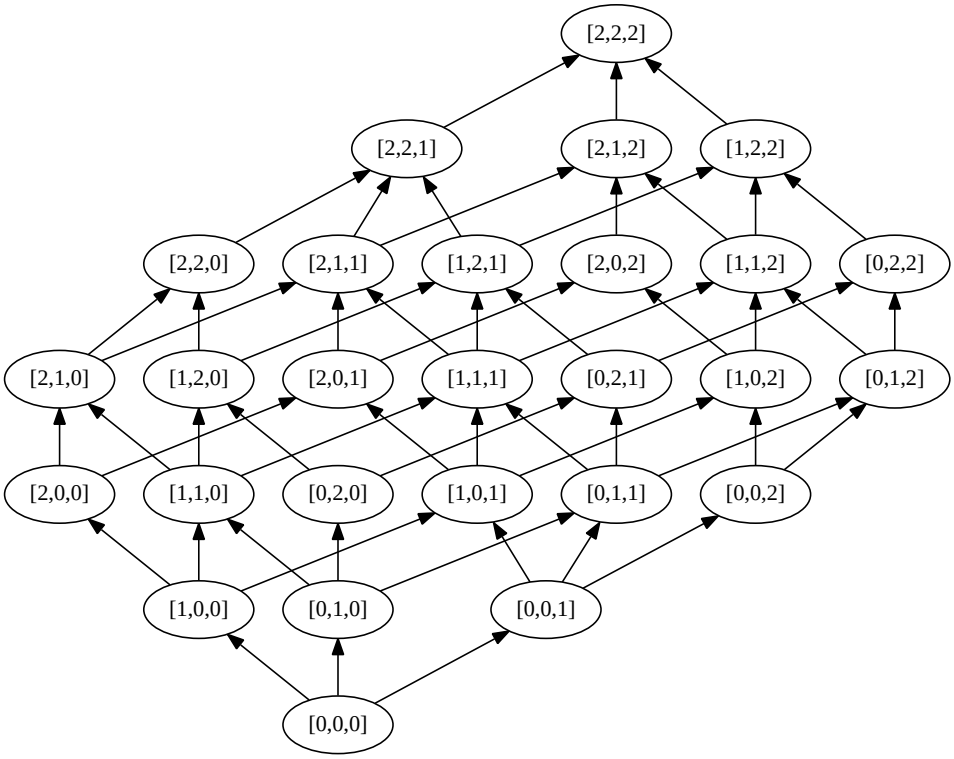


Figure 3.1: Cell Poset $n = 3, q = 3$

i.e. at each co-ordinate of $\boldsymbol{c}$, the value at that co-ordinate is less than that of the value at the co-ordinate at $\hat{\boldsymbol{c}}$. With the above relation, the set of cell-state vectors forms a partially ordered set (poset) (the partially ordered set is of the product topology of n products of the ordered space $\{0, 1, \ldots, q-1\}$).

By our assumption, the smallest change that can be made to a cell state vector is by incrementing the value of a single cell. The states that are possible after γ increments will be called the γth generation. The cell poset and its generations are shown in Figure 3.1 for $n = 3$ and $q = 3$.

Let G be the function that gives the generation of the cell state vector $G : C^{n,q} \to \mathbb{Z}^+$ which is the weight of the cell state vector

$$w(\boldsymbol{c}) = G(\boldsymbol{c}) = \sum_{i=0}^{n-1} c_i$$

From the above poset, we have that

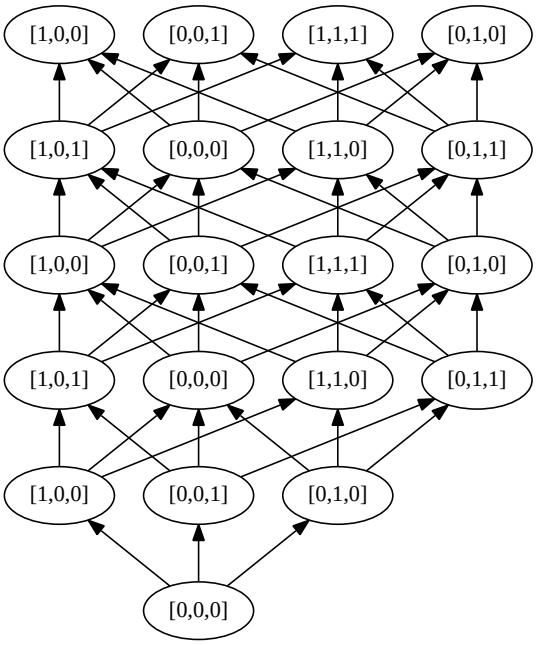


Figure 3.2: Zero Start Variable Poset $k = 3, l = 2$

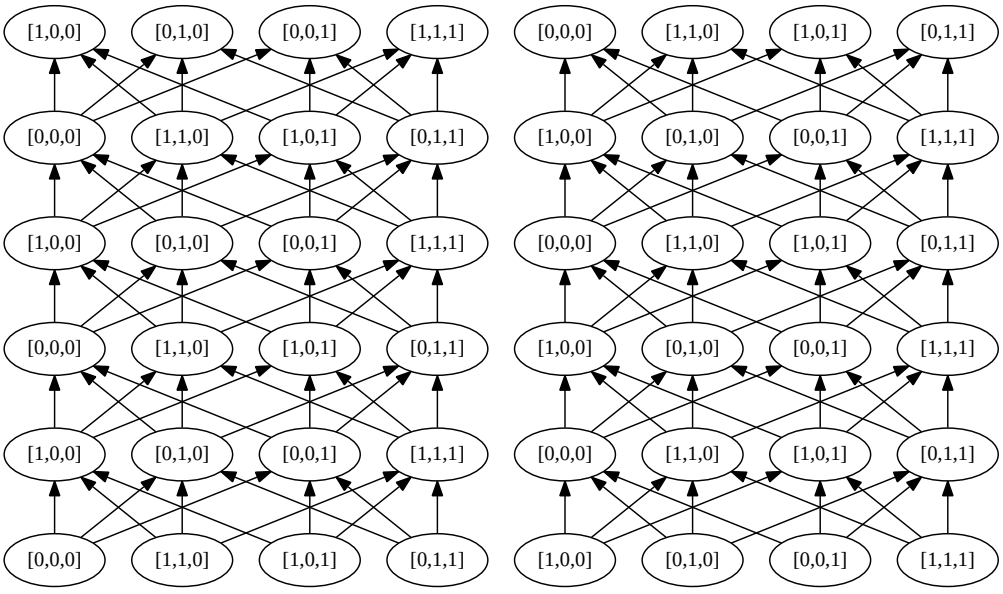


Figure 3.3: Full Start Variable Poset $k = 3, l = 2$

1. the number of vertices of the poset is q^n
2. given n cells with the maximum value of $q-1$, a cell can be updated $n(q-1)$ times from the all zero cell state vector to the all $(q-1)$ cell state vector.

3.2.2.1 Edge Characteristics

Given a cell state vector $\boldsymbol{c}$, let $\phi_i(\boldsymbol{c})$ denote the number of cells in $\boldsymbol{c}$ whose value is i. For any cell state vector, the maximum number of incoming and outgoing edges are n. When a cell in a vector is $q-1$, it cannot be further incremented and thus results in one less outgoing edge. When a cell is 0, it could not have been incremented from -1 and thus has one less incoming edge. Thus,, for each cell state vector $\boldsymbol{c}$,

1. the number of outgoing edges is $n - \phi_{q-1}(\boldsymbol{c})$
2. the number of incoming edges is $n - \phi_0(\boldsymbol{c})$

3.2.3 Data Variable Representation

Given k variables which can store l values (i.e. any value from the set $\{0, 1, \ldots, l-1\}$) and we call the vector

$$[v_0, v_1, \ldots, v_{k-1}]$$

the variable vector and $V^{k,l}$ the space of all variable vectors of length k each containing l values.

We assume that at each update step just one variable gets updated. Thus, if $\boldsymbol{v} = [v_0, v_1, \ldots, v_{k-1}]$ was updated to the variable vector $\hat{\boldsymbol{v}} = [\hat{v}_0, \hat{v}_1, \ldots, \hat{v}_{k-1}]$ and the updated variable was for variable i, then $\forall j \in \{0, 1, \ldots, k\} \backslash \{i\}$,

$$v_j = \hat{v}_j$$

As with the notions of generations with cell state vectors, we can impose a natural

poset on the variable state vectors as well. We call the set of variable state vectors available after γ updates to be the γth generation. However, several generations will share the same variable state vectors and so we distinguish between them by indexing to the number of variable updates.

However, there are two possible interpretations to the variable vector depending on the starting conditions, do we start with the all-zero variable vector in 0th generation or do we start with the full set of possible values for k variables taking any of the l variable values in the 0th generation? For the first, we call it the full start poset and the latter the zero start poset. Here, we focus on the zero start poset.

3.2.3.1 Full Start Poset

If the 0th generation has all the possible variable state vectors, then the variable poset forms a hypercube of k dimensions of alphabet l. For this poset, we have

1. number of vertices v_d is k^l
2. number of edges is $v_d k(l-1)$; one data can change to any of the l values except the old value, each vertex having exactly $k(l-1)$ edges.

3.2.3.2 Zero Start Poset

We assume that we start with the all zero vector in the 0th generation and each subsequent generation updates only one variable. The 1st generation has variable state vectors that are non-zero in one location.

3.2.3.2.1 Binary Variables Suppose that $l = 2$ (the variables are binary) and at generation i, we have a variable vector $[v_0, v_1, \ldots, v_{k-1}]$, then the $i+1$st generation does not have that variable vector. This is illustrated in Figure 3.2 and the above fact can be restated as that any two variable vectors in the same generation differ in at least two positions.

3.2.4 Update Cells Representation

Since update cells are emulated flash memory cells, the update cell representation is similar to that of the flash memory cells. Given k update cells, we denote the update state vector as

$$[u_1, u_2, \ldots, u_{k-1}]$$

and since they allow for t updates, which have a $t+1$ levels of $\{0, 1, \ldots, t\}$.

The first difference from flash memory cells is that at most only one u_i can have value maximum value t. If an update cell reaches its maximum value, then we cannot update it further. This is in contrast to flash memory cells where after some of the flash memory cells reach the maximum value of $q-1$, we can still increment other flash memory cells.

The second difference is that the height of the poset is $t+1$ since we limit the total number of updates to t. Thus, after t updates, the update cells cannot be further incremented.

3.2.4.1 Conversion to Variable Poset

To convert an update poset to a zero start variable poset, all we need to do is take the modulo of the update cells. Suppose we have a update poset with k update cells and to transform it to a variable poset of k variables with l levels, we set the new variables as

$$[u_1 \mod l,\ u_2 \mod l,\ \ldots\ ,\ u_{k-1} \mod l]$$

The update poset for $k = 3$ and $t = 3$ is shown in Figure 3.4. If we took modulo 2 of the update state vectors in the poset, we would get the first four generations of the variable poset in Figure 3.2.

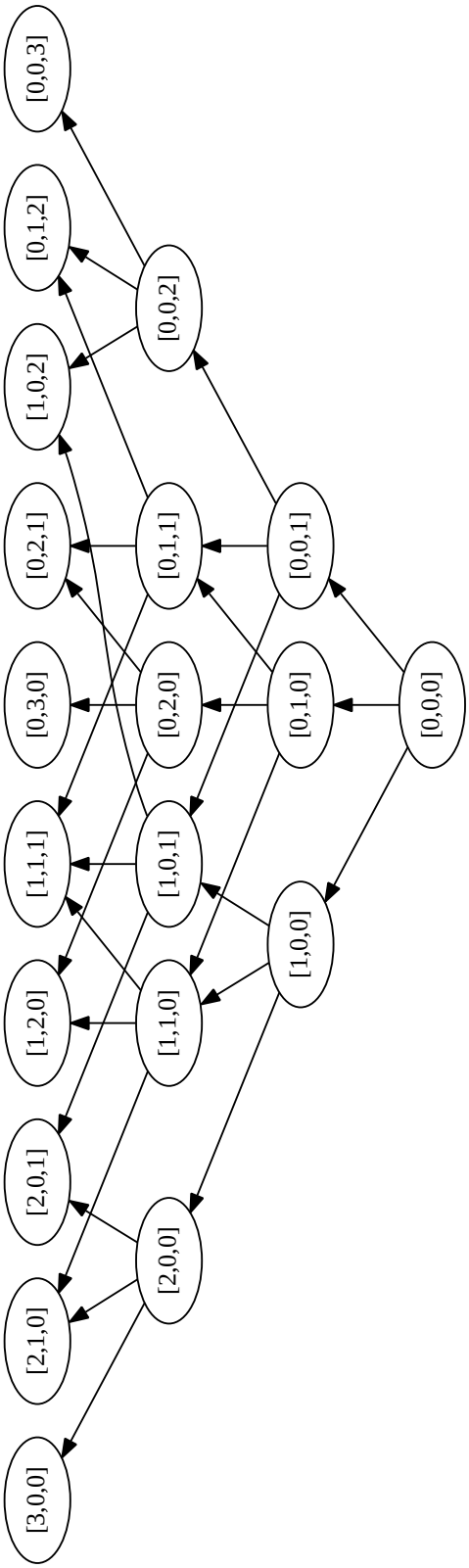


Figure 3.4: Update Poset for $k = 3$ and $t = 3$

3.2.5 Floating Codes Mappings

A floating code is essentially a mapping between a given variable state poset and a given cell state poset. In this mapping, a cell state vector represents some variable state vector. Since the same variable state vectors are repeated in many generations, the new variable state vector is indexed to the generation in the variable poset. When a variable is updated, to reflect the new variable vector, a new cell state vector higher than the current one is used to represent the new updated variable state variable.

Thus, a sequence of updates happens on a *chain* in the partially ordered set. As a consequence, we can state that for a floating code, every chain in the variable poset must be mapped to some chain in the cell poset.

Next, we look at the two mapping, the decode and update mappings that are used to describe a floating code.

3.2.5.1 Decode and Encode Mapping

The decode mapping D maps cell state vectors to variable state vectors, i.e.

$$D : C^{n,q} \to V^{k,l}$$

The mapping need not be injective but must be surjective. It needs to be surjective as every possible variable state vector must be represented by a cell state vector. It may not injective because multiple cell state vectors might be decoded as the same variable vector. The decode mapping function D might also not be a function because variable state vectors of different generations might be mapped to the same cell state vector.

We call the inverse of this mapping the *encode mapping*

$$E : V^{k,l} \to C^{n,q}$$

which gives us the cell state vectors that store a particular variable state vector. Note that this mapping need not be surjective since there could be cell state vectors that are unused.

3.2.5.2 Update Function

Given the current cell state vector $\boldsymbol{c}$ and desired variable state vector $\boldsymbol{v}$, the update function

$$U : C^{n,q} \times V^{k,l} \rightarrow C^{n,q}$$

gives the cell state vector $\hat{\boldsymbol{c}}$ that reflects the desired variable state vector. However, the cell state vectors can be updated such that the new cell state vector is greater, i.e, $[\hat{c}_0, \hat{c}_1, \ldots \hat{c}_{n-1}] > [c_0, c_1, \ldots, c_{n-1}]$. If the above is not possible, then the cell is unusable until it has been erased.

In the poset form, the update function is canonical from the decode and encode mapping. Given the current cell state vector $\boldsymbol{c}$, the update function finds the cell state $\hat{\boldsymbol{c}}$ vector that contains the variable state vector and that $\boldsymbol{c} < \hat{\boldsymbol{c}}$, and that can be reached by the least number of updates. In other words, given the current generation γ of $\boldsymbol{c}$, we would like to find $\hat{\boldsymbol{c}}$ in generation $\hat{\gamma}$ that decodes to the desired variable state vector $\boldsymbol{v}$ and $\boldsymbol{c} < \hat{\boldsymbol{c}}$ with the constraint that it minimizes the difference in the generations $\hat{\gamma} - \gamma$.

In notation, the canonical update function can be defined as

$$U(\boldsymbol{c}, \boldsymbol{v}) = \arg\min_{\substack{\boldsymbol{c} < \hat{\boldsymbol{c}} \\ D(\hat{c}) = \boldsymbol{v}}} G(\hat{\boldsymbol{c}})$$

3.2.6 Update Code Mappings

Similar to the floating codes, an update code is a mapping between an update poset and a cell state poset. Each update state vector must be mapped to one or more cell state vectors with a decode mapping and cell state vectors to none or one update state vector

by the encode mapping. Similarly, the update function can be inferred from the encode and decode mappings.

In Section 3.2.4.1, we described how an update poset can be converted to a variable poset. Similarly, we can also convert an update code to a floating code by replacing the update state vectors by the corresponding variable state vectors.

3.2.7 Floating Codes Optimality

Since the cell poset allows for $n(q-1)$ updates, the largest possible number of variable updates is $n(q-1)$ and serves as a loose upper bound.

However, the last generation in the cell poset has only one element whereas the variable poset usually has more than that (when $k > 1$) and so $n(q-1)$ updates are only possible when $k = 1$.

Let t be the greatest number of updates possible before a cell becomes unusable and requires erasure. From above,

$$t \leq n(q-1)$$

We can further tighten the bound observing that each cell state vector must allow for $k(l-1)$ variable updates. To do the update in one cell increment, we must have $k(l-1)$ outgoing edges in the cell state vector. All the updates might be possible when a cell state vector has n cells and when $n \geq k(l-1)$. However, if any of the cells reach state $q-1$, further updating of the cell is not possible and we lose an outgoing edge. Thus, from here, we can see that only $(n-k(l-1)+1)(q-1)$ variable updates are actually possible with a single cell increment because after $(n-k(l-1)+1)(q-1)$ updates, the number of cells that can be updatable is less than $k(l-1)$. Then, we then have to update using more than one increment and if we use two increments for each update, we can do further $\left\lfloor \frac{[k(l-1)-1](q-1)}{2} \right\rfloor$ increments.

If $n < k(l-1)-1$, we have to start off doing each variable update with more than one

cell state increment. There are $n(q-1)$ cell state increments possible and each variable increment using two cell state updates, we have $\left\lfloor \frac{n(q-1)}{2} \right\rfloor$ updates possible.

This above is the upper bound as given in [70]

$$t \leq \begin{cases} [n-k(l-1)+1](q-1) & \\ + \left\lfloor \frac{[k(l-1)-1](q-1)}{2} \right\rfloor & n > k(l-1)-1 \\ \left\lfloor \frac{n(q-1)}{2} \right\rfloor & n < k(l-1)-1 \end{cases} \tag{3.1}$$

As defined in [113], the *deficiency* δ is

$$\delta = n(q-1) - t$$

and from the definition of the deficiency we have that $\delta > 0$. Thus, finding the least upper bound on t is finding the greatest lower bound on δ. For binary variables ($l = 2$), the best known bound on deficiency is $O(k(q-1))$ [92], derived from the upper bound for t given in Equation 3.1. For $l > 2$, the lower bound for the deficiency is $O(k(q-1)(l-1))$ (assuming $n > k(l-1)-1$).

3.2.7.1 t_1 Optimality

As given in Equation 3.1, the number of variable updates possible is determined first by unit increments and then, by multiple increments. Let t_1 be the number of possible variable updates with a unit increment. We here assume $n > k(l-1)+1$ otherwise $t_1 = 0$. We call a floating code t_1-optimal if

$$t = [n-k(l-1)+1](q-1)$$

and when $k = 2$, $t = (n-k+1)(q-1)$. Basically, a floating code is a t_1-optimal code when it is possible to update variables using single cell increments until we reach the generation

which has cell state vectors which doesn't have enough updateable cells. A t_1-optimal code also has deficiency

$$O(k(q-1)(l-1))$$

as

$$\delta = n(q-1) - t = k(l-1)(q-1)$$

For a t_1-optimal code to be a fully optimal code, we also have to be able to update variables in the last $[k(l-1)-1](q-1)$ generations using two generations in the cell state poset for one variable update.

3.2.8 Update Codes Optimality

For update codes, we want to maximize t, the total number of possible updates on k update cells. The bound given in Equation 3.1 for floating codes also applies to update codes because each update state vector has to have k outgoing edges for the k possible updates.

An obvious solution for update codes is to divide the n blocks into k groups equally so that $\lfloor \frac{n}{k} \rfloor$ flash cells are used for one variable giving us

$$t = \left\lfloor \frac{n}{k} \right\rfloor (q-1) \tag{3.2}$$

However, since all update codes are floating codes, the bound on t from the theory of floating codes is

$$t \leq [n-k+1](q-1) + \left\lfloor \frac{[k-1](q-1)}{2} \right\rfloor \tag{3.3}$$

As an example, taking $n = 12$ and $k = 4$, the bound in Equation 3.2 gives us $t = 3(q-1)$ while Equation 3.3 gives us $t \leq 10(q-1)$, more than three times the number of updates possible by using update codes.

Here, we are interested in constructing t_1-optimal floating codes for optimal deficiency

so we will look at the equivalent definition for update codes. We call an update code t_1-optimal if it can encode the first

$$(n-k+1)(q-1)$$

update cell increments each with one single flash memory cell increment. Then, if we find a t_1-optimal update code, we have a t_1-optimal binary floating code with optimal deficiency.

We do not have to worry about optimality for floating codes for $l > 2$ because the binary t_1-optimal floating codes we create from update codes are isomorphic to t_1-optimal $l > 2$ floating codes which we show in Section 3.5. Thus, if we find an update code, we can derive a binary floating code which can then be used for $l > 2$ floating codes.

3.3 Properties of the Posets

3.3.1 Cell Poset

Let γ denote the generation, i.e. γ updates or increments have so far been made. Let S_γ denote the set of cell state vectors that are of the γth generation or the set of all cell state vectors possible after γ increments.

The all zero cell state vector is the only vector in generation 0. The last generation has all the all-$(q-1)$ vector and that can be reached after $n(q-1)$ updates. Thus, there are $n(q-1)+1$ possible generations.

Let $(c_0, c_1, \ldots, c_{n-1})$ be a cell state vector in S_γ. We have the following properties:

1. The sum of the increments is γ

$$\sum_{i=0}^{n-1} c_i = \gamma$$

2. All the values in the cells are less than q, i.e., $\forall i \in \{0, 1, \ldots, n-1\}$

$$c_i < q$$

3.3.1.1 Counts of States Per Generation

Let $N_s(n, q, \gamma)$ denote the number of states in the cell state poset at generation γ where cells state vectors are n cells wide and each cell can take a maximum value of $q-1$. Since the total must sum up to all the states of the poset, we have

$$\sum_{\gamma=0}^{nq-1} N_s(n, q, \gamma) = q^n$$

We would like to know the number of states in each generation. We look at an recursive definition. For the base case we have that

$$N_s(1, q, \gamma) = \begin{cases} 1 & 0 \leq \gamma < q \\ 0 & \text{otherwise} \end{cases}$$

since if $n = 1$, each generation can only have 1 state and the recursive equation we have

$$N_s(k, q, \gamma) = \sum_{i=0}^{q-1} N_s(k-1, q, \gamma - i)$$

where we look at cell state vectors of length $k-1$ and count all the possible vectors that sum to γ is if we added all cells from values of 0 to $q-1$.

This distribution follows the distributions of "s" [48] and for large values of n, it can be approximated by a normal distribution [48].

3.3.1.2 Properties of the Cell State Vectors

We first define the notion of a *vertex cover*.

Definition 3.3.1. *Let $\boldsymbol{c_1}$ be a vertex in a poset P. We say the vertex $\boldsymbol{c}_2 \in P$ covers $\boldsymbol{c}_1$ ($\boldsymbol{c}_1 \sqsubset \boldsymbol{c}_2$) if*

1. $\boldsymbol{c}_1 < \boldsymbol{c}_2$
2. $\forall \boldsymbol{c}_3$ *such that* $\boldsymbol{c}_1 < \boldsymbol{c}_3$, *we have* $\boldsymbol{c}_2 < \boldsymbol{c}_3$ *or that* $\boldsymbol{c}_2 || \boldsymbol{c}_3$, *i.e., they are incomparable.*

In our case, the cover of a cell state vector is the set of vectors that are in immediate next generation that can be reached after a single update.

Theorem 3.3.2. *Let $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ be two cell state vectors in the same generation, i.e., $w(\boldsymbol{c}_1) = w(\boldsymbol{c}_2)$. Then, at most one cell state vector covers both $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$.*

Proof. Suppose $\boldsymbol{c}_3 = [c_0, c_1, \ldots, c_{n-1}]$ covers both $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ and that $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ were incremented at i and j to reach $\boldsymbol{c}_3$ respectively. Then,

$$\boldsymbol{c}_1 = [c_0, c_1, \ldots, c_{i-1}, c_i - 1, c_{i+1}, \ldots, c_{n-1}]$$

and

$$\boldsymbol{c}_2 = [c_0, c_j, \ldots, c_{j-1}, c_j - 1, c_{j+1}, \ldots, c_{n-1}]$$

Let $\boldsymbol{c}_4$ be the cell state vector obtained by incrementing $\boldsymbol{c}_1$ at location k where $k \neq i$. Then, $\boldsymbol{c}_4$ covers $\boldsymbol{c}_1$ but can never cover $\boldsymbol{c}_2$ because $\boldsymbol{c}_{4,i} = c_i - 1$ and $\boldsymbol{c}_{2,i} = c_i$ and $\boldsymbol{c}_{2,i} > \boldsymbol{c}_{4,i}$. Thus, any other cell state vector that covers $\boldsymbol{c}_1$ cannot cover $\boldsymbol{c}_2$ and thus, $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ have just one cover. □

Lemma 3.3.3. *Let $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ be two cell state vectors in the same generation. Then, they must vary at more than one location.*

Proof. If they are in the same generation, then $G(\boldsymbol{c}_1) = G(\boldsymbol{c}_2)$, and thus,

$$\sum_{i=0}^{n-1} c_{1,i} = \sum_{i=0}^{n-1} c_{2,i}$$

but this is not possible if they vary in only one location. □

Lemma 3.3.4. *Let $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ be two cell state vectors in the same generation. Then, $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ have a common cover $\boldsymbol{c}_3$ if and only if they must vary by 1 at two locations.*

Proof. Suppose that $\boldsymbol{c}_3$ is a common cover of $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$, then $\boldsymbol{c}_3$ varies with $\boldsymbol{c}_1$ at only one location by 1 and similarly, $\boldsymbol{c}_3$ varies with $\boldsymbol{c}_2$ at only one location by 1. $\boldsymbol{c}_3$ cannot vary with $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ at the same location and thus, $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ vary at two locations by 1.

Suppose that $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ vary by 1 at two locations, say locations i and j. Then, $\boldsymbol{c}_{1,i} = \boldsymbol{c}_{2,i} + 1$ and $\boldsymbol{c}_{1,j} + 1 = \boldsymbol{c}_{2,j}$, or $\boldsymbol{c}_{1,i} + 1 = \boldsymbol{c}_{2,i}$ and $\boldsymbol{c}_{1,j} = \boldsymbol{c}_{2,j} + 1$. Either way, let $\boldsymbol{c}_{3,i} = \max(\boldsymbol{c}_{1,i}, \boldsymbol{c}_{2,i})$ and $\boldsymbol{c}_{3,j} = \max(\boldsymbol{c}_{1,j}, \boldsymbol{c}_{2,j})$ and $\boldsymbol{c}_{3,k} = \boldsymbol{c}_{1,k} = \boldsymbol{c}_{2,k}$ for $k = \{1, \ldots, n\} \backslash \{i, j\}$ and thus, $\boldsymbol{c}_3$ covers both $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$. □

We next define the *root* which is simply the opposite of a cover. If $\boldsymbol{c}_1$ covers $\boldsymbol{c}_2$, then $\boldsymbol{c}_2$ is a root of $\boldsymbol{c}_1$. We use this to show that if two vertices share a cover, then they must also share a root.

Definition 3.3.5. *Let $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ be vertices in the poset P. We say that $\boldsymbol{c}_1$ is a root of $\boldsymbol{c}_2$ if*

1. $\boldsymbol{c}_1 < \boldsymbol{c}_2$
2. $\forall \boldsymbol{c}_3 \in P$ *such that* $\boldsymbol{c}_3 < \boldsymbol{c}_2$, *we either have* $\boldsymbol{c}_3 < \boldsymbol{c}_1$ *or that* $\boldsymbol{c}_3 || \boldsymbol{c}_1$ *($\boldsymbol{c}_3$ or $\boldsymbol{c}_1$ are incomparable).*

Theorem 3.3.6. *Let $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ be cell state vectors in the same generation. Suppose that $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ share a common cover $\hat{\boldsymbol{c}}$ ($\boldsymbol{c}_1 \sqsubset \hat{\boldsymbol{c}}$ and $\boldsymbol{c}_2 \sqsubset \hat{\boldsymbol{c}}$). Then, there must be a common root $\boldsymbol{r}$ between $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$. ($\boldsymbol{r} \sqsubset \boldsymbol{c}_1$ and $\boldsymbol{r} \sqsubset \boldsymbol{c}_2$). Furthermore, there can only be at most one such root.*

Proof. By Lemma 3.3.4, since $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ have a common cover, we have that $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ vary at two locations. Let these two locations be i and j and assuming ith location is larger in $\boldsymbol{c}_1$, we have the following

$$\boldsymbol{c}_{1,i} = \boldsymbol{c}_{2,i} + 1 \qquad \boldsymbol{c}_{1,j} = \boldsymbol{c}_{2,j} - 1$$

Let $\boldsymbol{r}$ be the cell state vector identical to $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ in all locations except i and j. Let $\boldsymbol{r}_i = \boldsymbol{c}_{2,i}$ and $\boldsymbol{r}_j = \boldsymbol{c}_{1,j}$. Then, $\boldsymbol{r}$ is one generation below $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ (i.e., $G(\boldsymbol{r}) = G(\boldsymbol{c}_1) - 1$). Also, $\boldsymbol{r}$ is a root of both $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ because incrementing by 1 at the ith location leads to $\boldsymbol{c}_1$ and incrementing by 1 at the jth location leads to $\boldsymbol{c}_2$. Thus, $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ have a common root.

Now, let us assume that we have two common roots $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$. But, this would make $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$ have two common covers $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$. By Theorem 3.3.2, this is not possible and hence, we have a contradiction. Thus, there can only be one common root. □

Theorem 3.3.7. *Let $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ be cell state vectors in the same generation. Then, $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ have a common cover if and only if they have a common root.*

Proof. We showed that they have a common root if they have a common cover in Theorem 3.3.6.

We need to show that if $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ have a common root, then they have a common cover. Let $\boldsymbol{r}$ be the common root and suppose that $\boldsymbol{c}_1$ is the result of incrementing $\boldsymbol{r}$ at the ith location and $\boldsymbol{c}_2$ is the result of incrementing $\boldsymbol{r}$ at the jth location. Then,

$$\boldsymbol{c}_{1,i} = \boldsymbol{r}_i + 1 \qquad \boldsymbol{c}_{2,j} = \boldsymbol{r}_j + 1$$

Let $\hat{\boldsymbol{c}}$ be the vector which is exactly the same as r except at positions i and j. We let

$$\hat{\boldsymbol{c}}_i = \boldsymbol{r}_i + 1 \qquad \hat{\boldsymbol{c}}_j = \boldsymbol{r}_j + 1$$

Now, $\hat{\boldsymbol{c}}$ is two generations higher than $\boldsymbol{r}$ and one generation higher than $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$. Note that $\hat{\boldsymbol{c}}$ covers $\boldsymbol{c}_1$ because it is $\boldsymbol{c}_1$ incremented at jth location by 1 and it also covers $\boldsymbol{c}_2$ because it is $\boldsymbol{c}_2$ incremented at ith location by 1. Thus, $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ have a common cover if they have a common root. □

3.3.2 Update Poset

The properties of update posets are almost identical to that of cell posets. An update code of parameters k update cells and t updates will be the first $t+1$ generations of the cell poset where $n = k$ and $q = t + 1$. Thus, all the properties regarding vertex counts, covers and roots carry over from cell posets to update posets.

3.3.3 Variable Poset

The variable state poset is assumed to have t generations, the maximum number of updates afforded by the cell state poset using a given floating code. As with the upper bound on the number of updates, the number of generations cannot be more than $n(q-1)+1$.

3.3.3.1 Counts of States Per Generation

We have to examine the full-start and zero-start variable state poset differently. We use the notation $N_v(k, l, \gamma)$ to denote the number of states in the γth generation with the variable state poset with k variables whose value is at most $l - 1$.

3.3.3.1.1 Full-Start Variable State Poset Each generation has all the possible variables states. Since we have k variables with l levels, each generation has l^k states.

3.3.3.1.2 Zero-Start Variable State Poset For $\gamma \geq k$ and $l > 2$, the zero-start variable state poset the counts of generations are identical to full-start variable state vectors, i.e. l^k states. For $\gamma \geq k$ and $l = 2$, the counts of generations are half, i.e. 2^{k-1}.

Now, for $\gamma < k$ and $l > 2$, the first generation has the all-zero variable state vector only. The next generation has all the vectors that are non-zero in one location and excludes the all-zero vector, i.e. $k(l-1)$ variables state vectors. The generation after includes all the variables that can be non-zero in at least 2 locations and includes the all zero vector.

Thus, we have

$$N_v(k,l,\gamma) = \begin{cases} 1 & \gamma = 0 \\ k(l-1) & \gamma = 1 \\ \sum_{i=0}^{\min(\gamma,k)} \binom{k}{i}(l-1)^i & \gamma > 1 \end{cases}$$

Now, for the case of $\gamma < k$ and $l = 2$ binary variables, the variable state vectors in the current generation is not present in the previous or next generation, and thus we have to classify by even or odd generation. Thus, we have

$$N_v(k,l,\gamma) = \begin{cases} \sum_{i=0,i\ even}^{\min(\gamma,k)} \binom{k}{i}(l-1)^i & \gamma \text{ is even} \\ \sum_{i=1,i\ odd}^{\min(\gamma,k)} \binom{k}{i}(l-1)^i & \gamma \text{ is odd} \end{cases}$$

3.3.3.2 Properties of the Variable State Vectors

Unlike the cell state vectors, a variable state vector is covered by all the variable state vectors that denotes a change in a variable.

Theorem 3.3.8. *Let $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ be two variable state vectors in the same generation. Then, there are at most* $\max(l-2, 2)$ *variable state vectors that cover both of them. If $l = 2$ (the variables are binary), they are either none or two variable state vectors that cover both of them.*

Proof. If $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ differ in more than 2 variables, they cannot be covered by the same variable state vector since we can only update one variable at a time. Now, suppose $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ differ in exactly two variables at positions i and j, i.e., $\boldsymbol{v}_{1,i} \neq \boldsymbol{v}_{2,i}$, $\boldsymbol{v}_{1,j} \neq \boldsymbol{v}_{2,j}$ and $\boldsymbol{v}_{1,s} = \boldsymbol{v}_{2,s}$ for $s = \{0, 1, \ldots, k-1\} \backslash \{i, j\}$.

Let $\boldsymbol{v}_3$ be the vector such that $\boldsymbol{v}_{3,i} = \boldsymbol{v}_{2,i}$ and $\boldsymbol{v}_{3,j} = \boldsymbol{v}_{1,j}$ and equal to $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ at all other locations. Here, $\boldsymbol{v}_3$ covers both $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$, setting $\boldsymbol{v}_{1,i} = \boldsymbol{v}_{3,i}$ gives us $\boldsymbol{v}_3$ and setting $\boldsymbol{v}_{2,j} = \boldsymbol{v}_{3,j}$ gives us $\boldsymbol{v}_3$.Similarly, $\boldsymbol{v}_4$ which has $\boldsymbol{v}_{4,i} = \boldsymbol{v}_{1,i}$ and $\boldsymbol{v}_{4,j} = \boldsymbol{v}_{2,j}$ and equal to $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ everywhere else also covers both $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$. Thus, when $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ differ in exactly

two locations, they are covered by two variable state vectors.

Now, suppose $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ vary in one location i only. We have to consider two cases, one when the variables are binary ($l = 2$) and one when $l > 2$.

First assume that $l = 2$ and that the variables are binary. If this is a zero start variable poset, two such variable state vectors cannot be in the same generation since they have to differ in at least two positions. If not a zero-start poset and the variables binary with $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ differing at location i only, $\boldsymbol{v}_{1,i}$ is either 0 or 1 and $\boldsymbol{v}_{2,i}$ is the other. Since updating ith variable changes $\boldsymbol{v}_1$ to $\boldsymbol{v}_2$ and vice verse, their next generation versions cover each other but the next generation of $\boldsymbol{v}_1$ does not cover itself. Thus, there are no covers for binary variables when they differ at one location only.

When $l > 2$, both $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ can be updated to a value that is not equal to either $\boldsymbol{v}_{1,i}$ and $\boldsymbol{v}_{2,i}$ to form variable state vectors that covers both $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ and there are $l-2$ such covers. □

Lemma 3.3.9. *Let $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ be two variable state vectors in the same generation. Then, $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ have a common cover $\boldsymbol{v}_3$ if and only if $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ differ in at most two locations.*

Proof. Suppose $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ differ in only one location. Then, in the next generation, both the location can be set to the same value to cover both $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$. Now, if they differ in two locations, they can be set to each other's value at the differing locations in the next generation to get a vector that covers both $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$. Now, if $\boldsymbol{v}_3$ covers both $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$, then they can differ at at most locations because each generation only allows an update of one variable. □

3.4 Constructing Update Codes

Given a cell state poset P and a variable state poset V, an update code $\mathcal{F}$ can be constructed by finding a decode mapping D. After the update code is constructed, the corresponding binary floating code can be derived from the update code by taking modulo

two of all the update state vector entries.

When constructing floating codes, we have the following problems and challenging aspects:

1. the cell poset and variable poset have different shapes; the cell state poset is "round" in shape and the variable state poset "rectangular with a triangular bottom"

2. as from lemma 3.3.2 and 3.3.8, the structure of the covers are different; two cell state vectors can only have at most one cover whereas two variable state vectors can have more than one cover. For $l = 2$, it can have two covers from Theorem 3.3.8.

When we create update codes, the challenges become slightly different into the following:

1. though update and cell posets are similar, the n, q cell poset and k, t update poset parameters are different. Additionally, as update codes are only have $t + 1$ generations, the update posets are "triangular" in shape.

2. the cover structures are similar and two update state vectors can also only have a single shared cover.

In this section, we present a method of constructing update codes which is equivalent to constructing floating codes.

3.4.1 t_1-Optimal Update Code

We next present the construction of t_1 optimal update code class which has the following parameters for n, q and k:

1. arbitrary q and k and $n = k + 1$ or $n = k$

2. arbitrary n, q and $k = 2$.

By the definition of t_1 optimality for update codes, we should be able to do $(n-k+1)(q-1)$ update cell increments using unit cell increments. When $n = k$, we should be able to do $(q-1)$ update cell increments using single increments and when $n = k+1$, we should be able to do $2(q-1)$ update cell increments using single cell increments.

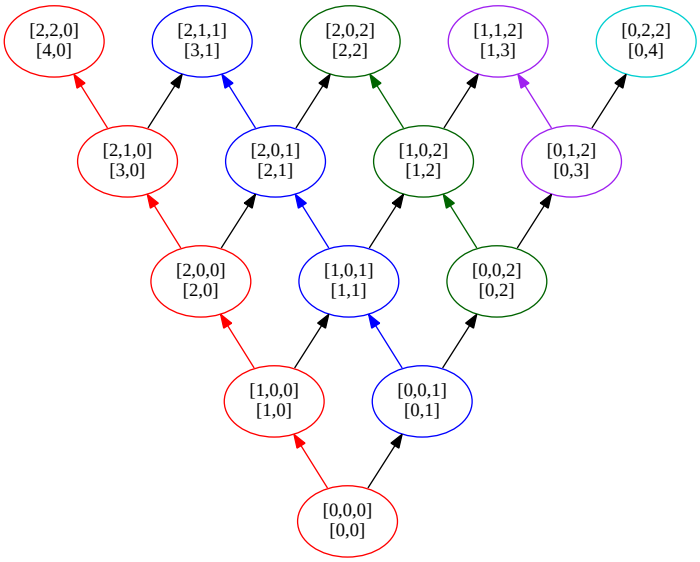
The construction is done using a recursive method where we build a (n, q, k) update code using $(n-1, q, k-1)$ update code. Thus, we only need to define the $(n, q, 1)$ update code and then rest of the floating codes can be derived from there.

To illustrate the construction of the code, we start with two examples, $(3, 3, 2)$ and $(4, 3, 3)$ update codes. We then present the recursive algorithm and then discuss the proof that the construction algorithm creates a valid update code. Since we are only investigating the t_1 optimality, we are only interested in the first $(n-k+1)(q-1)+1$ generations. We will only look that portion of the poset and ignore for now the remaining $(k-1)(q-1)$ generations which would require multiple cell increments.

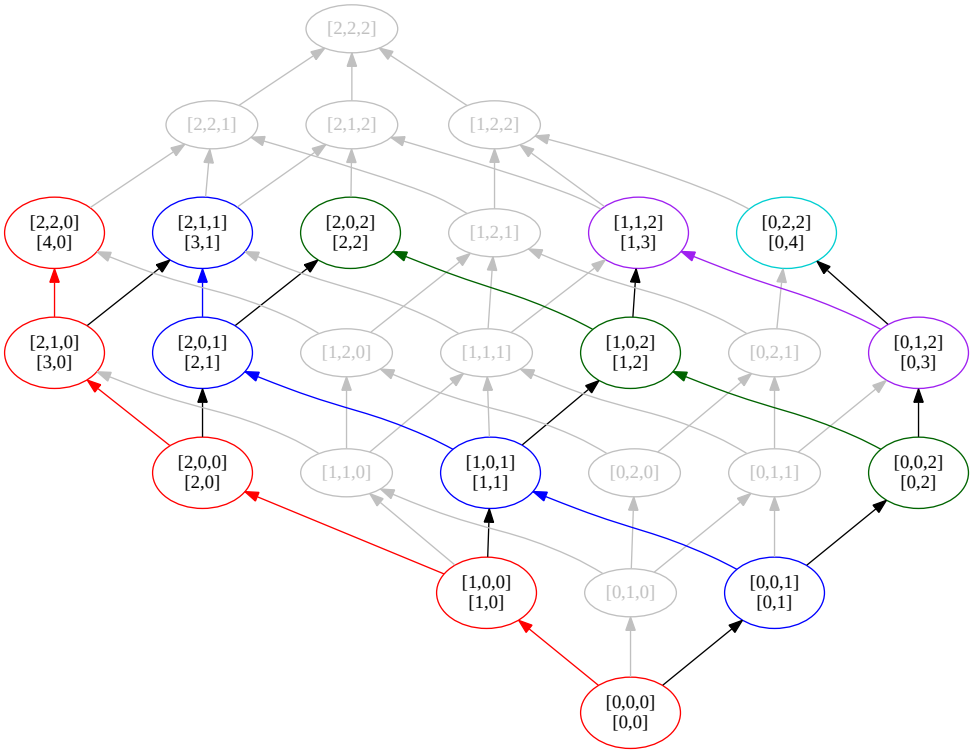
3.4.2 Example Constructions

3.4.2.0.1 $(3, 3, 2)$ **Code** Figure 3.5 shows the (3,2,2) update code construction where Figure 3.5a shows the update code poset only (without the cell state vectors that are not used). Figure 3.5b shows the update code in the full cell poset. The different colored vertices in the code represent sub-posets that are made from $(2, 3, 1)$ update code.

The red-colored sub-poset P_1 in Figure 3.5 is the $(2, 3, 1)$ update code but with a co-ordinate added to the end of both the cell and variable state vectors and set to 0. The blue sub-poset P_2 is P_1 but with $(0, 0, 1)$ added to the all the cell state vectors in P_1, $(0, 1)$ added to all the variable state vectors in P_1 and the top generation taken out. All the other sub-posets P_3, P_4, P_5 illustrated by the different colors are derived from P_1 by adding different roots to P_1.



(a) Update Code



(b) Code Embedded in Cell State Poset

Figure 3.5: Constructing update code for $k = 2$ from update codes for $k = 1$. $(n = 3, q = 3)$

3.4.2.0.2 $(4,3,3)$ **Code** We construct the $(4,3,3)$ update code using the $(3,3,2)$ update code F_1 we constructed above. The red sub-poset P_1 in Figure 3.7 is F_1 with a co-ordinate added to the end of the cell state vectors and variable state vectors of F_1 and set to 0. The blue sub-poset P_2 is P_1 with the root vector $(0,0,0,1)$ added to the cell state vectors and $(0,0,1)$ added to the variable state vectors in P_1.

The interesting thing to observe is how P_1 and P_2 are connected together. Since P_2 starts one generation above P_1, we draw the edge from the vertices in the ith generation in P_1 to the $i+1$st generation of P_2. The number of vertices are equal since they are both from the ith generation of P_1 and we draw the vertices in the same order as they appear so there is no ambiguity. We will prove why this is always possible in Section 3.4.4.

The other thing to look at is the generation of the posets $P_2, P_3, \ldots, P_5$. They are all constructed from P_1 by adding a root c_r to all the cell state vertices and c_u to all the vector state vertices. The c_u is the vector $(0,0,i)$ where i is the generation number. The method to choose the sequence of c_r is described in 3.4.3.

3.4.3 Construction Algorithm

Next we describe the construction algorithm for generating update code for arbitrary q, k and $n \in \{k, k+1\}$, or arbitrary n, q and $k = 2$. Here, the input is n, q, k and the output is an update code. The update code is represented as an array of arrays where each generation is an array of vector vertices.

The outline of the algorithm is given in Algorithm 1. The update code is generated recursively and to create a update code of (n,q,k) we use a update code for $(n-1,q,k-1)$. Eventually, when we reach $k = 1$ in the recursive algorithm, which means that there is only one variable, we can easily generate the update code for $(\hat{n},q,1)$ for $\hat{n} = n - k + 1$. We describe the procedure for generating $(\hat{n},q,1)$ update codes in the function `Create_1_Update_Code` in Section 3.4.3.1.

After creating the $(n-1,q,k-1)$ update code, we create the sub-poset P_1 by adding a

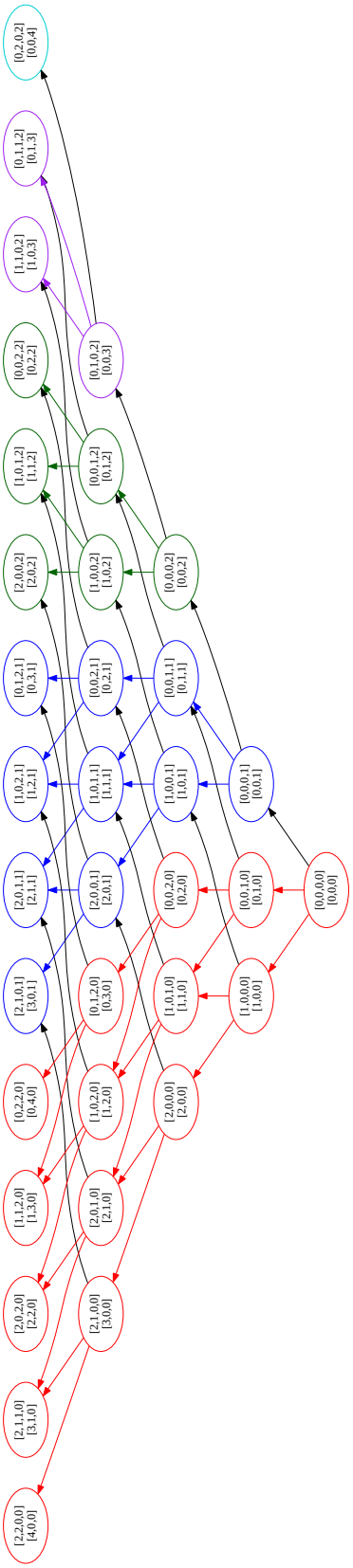


Figure 3.6: Update Code: Constructing an t_1-optimal update code for $k = 3$ from update codes for $k = 2$. $(n = 4, q = 3)$ with each color indicating a different sub-poset used

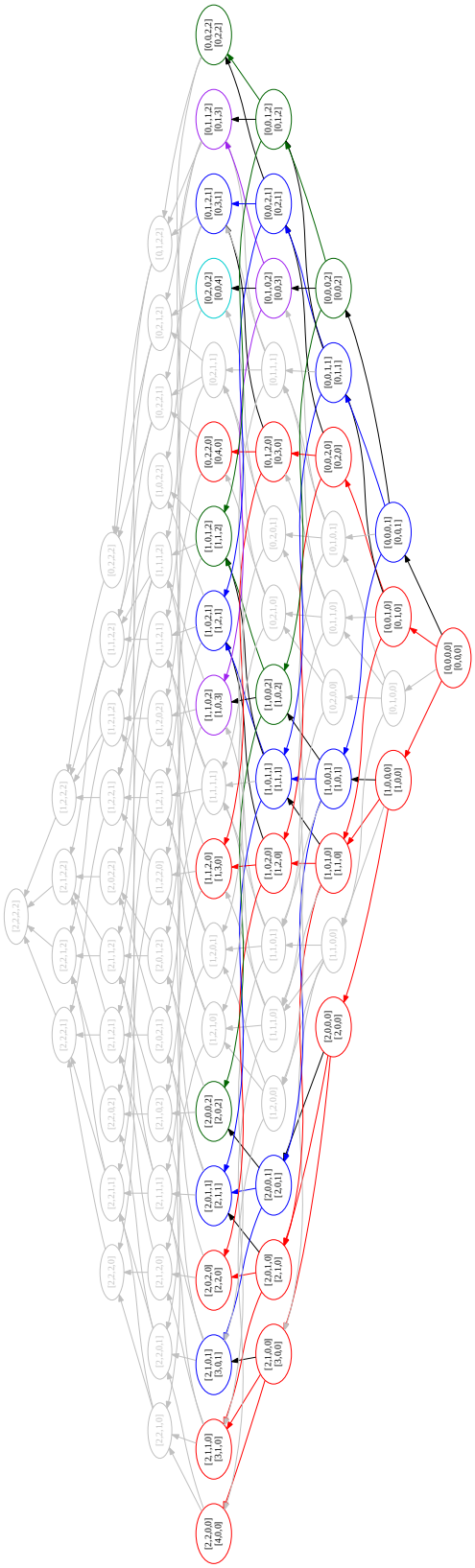


Figure 3.7: Update Code in Cell State Poset: Constructing an t_1-optimal update code for $k = 3$ from update codes for $k = 2$. $(n = 4, q = 3)$ with each color indicating a different sub-poset used

new co-ordinate at the end of the cell state vector and variable state vector and marking them all as zeros. After that, we generate the sequence of cell state vector roots that we are going to use to generate the sub-posets. We are going to generate $(n-k+1)(q-1)$ sub-posets and need as many roots. Note, the update state vector root sequence are zeros everywhere except the last co-ordinate which is the generation number.

Algorithm 1 Create (n, q, k) t_1-optimal update code

```
1: function F = CreateUpdateCode(n, q, k)
2:     if k = 1 then
3:         F = Create_1_Update_Code(n, q)
4:     else
5:         F_1 = CreateUpdateCode (n − 1, q, k − 1)
6:         P_1 = Add_Zero_Coordinates(F_1)
7:         F = P_1
8:         R = Generate_Roots (P_1)
9:         for i = 1 to (n − k + 1)(q − 1) do
10:            P_{i+1} = Create_Subposet (R[i], i)
11:            F = Join_Vertices (F, P_{i+1})
```

We build the rest of the update code F by iteratively generating the sub-posets P_i from the roots generated above and then and join the vertices of P_i to the update code. Once we have created $(n-k+1)(q-1)$ sub-posets and joined them all together, we have the complete update code.

Next, let's look at the functions in Algorithm 1 closely. In the entire update code construction algorithm, the only variable is how we construct the $(\hat{n}, q, k = 1)$ update code. This determines the roots and hence $P_1, P_2, \ldots$ and so, the final update code. Given (n, q) there are many ways to create a $(n, q, k = 1)$ update code but we will create the update code in the method given below. This allows the update code to be valid which we prove in Section 3.4.4.

3.4.3.1 `Create_1_Update_Code`

We describe the algorithm for creating an $(n, q, 1)$ update code. We only have to store one update cell so the vector state variable just the generation number. The total number

of generations is $n(q-1)+1$ and each generation has just one cell state vector. We start by letting all the cell state vectors be zero in the first generation and then continue by adding a single value to the left-most co-ordinate until it reaches $q-1$. Then, we move to the next left-most co-ordinate. So, the sequence of cell state vectors will be

$$\begin{gathered}(0,0,0,\ldots,0),(1,0,0,\ldots,0),\ldots,(q-1,0,0,\ldots,0),\\ (q-1,1,0,\ldots,0),\ldots,(q-1,q-1,0,\ldots,0),\\ \vdots\\ (q-1,q-1,\ldots,1)\ldots,(q-1,q-1,\ldots,q-1)\end{gathered}$$

The vector state variable is just the generation number.

3.4.3.2 `Add_Zero_Coordinates`

After we have generated a $(n-1,q,k-1)$ update code, we need to add one more co-ordinates to the end of the cell state and update state variables to make it into a (n,q,k) code. The value for the added co-ordinate is 0 and this gives us P_1, the first sub-poset of the update code. We will generate the rest of the sub-posets from P_1. We also generate the update code by attaching the sub-posets to P_1 and make a copy of P_1 to F for it.

3.4.3.3 `Generate_Roots`

For (n,q,k) update code, we have to generate $(n-k+1)(q-1)$ sub-posets and join them together. So, we have to generate $(n-k+1)(q-1)$ roots to generate each of the sub-posets. The roots should have weight from $1,2,\ldots,(n-k+1)(q-1)$ and each should be a cell increment of the previous one.

The root chosen to generate a sub-poset must be chosen so that the vertices in the generated sub-poset do not occur in any previous sub-posets. In other words, sub-posets do not share vertices. We will prove that the algorithm to generate the roots enforces this

property in Section 3.4.4.2. For example, in P_1 we have chosen the last co-ordinate of the cell state variables to be 0. If our root was 1 in the last co-ordinate, i.e., $(0, 0, \ldots, 0, 1)$, then the vertices generated from this root would not have occurred in the previous sub-posets. Similarly, $(0, 0, \ldots, 0, 2)$ is another root and we can go on to $(0, 0, \ldots, 0, q-1)$. However, we need $(n-k+1)(q-1)$ roots and the above is just $(q-1)$ roots. Thus, we need to find $(n-k+1)$ such co-ordinates.

The algorithm to generate the roots relies on the algorithm to find $(n-k+1)$ co-ordinates called root locations which we discuss below. For each of the co-ordinates, we generate $q-1$ roots giving the total of the required $(n-k-1)(q-1)$ roots. For each root location, we start with 1 and then generate successive roots until we reach $q-1$ at that location. The root locations used are largest co-ordinate first and then to lower co-ordinates.

Let $z_1, z_2, \ldots, z_{n-k+1}$ be the root locations in order of co-ordinate. By our construction, $z_{n-k+1} = n$ because we added a co-ordinate at the nth location and set it to 0. Thus, the sequence of generated roots will be

$$
\begin{gathered}
(0, \ldots, 0, 1), \ldots, (0, \ldots, 0, q-1), \\
(\ldots, 1_{z_{n-k-2}}, \ldots, q-1), \ldots (\ldots, q-1_{z_{n-k-2}}, \ldots, q-1), \\
\vdots \\
(\ldots, 1_{z_1}, \ldots, q-1_{z_2} \ldots, q-1), \ldots \\
(\ldots, q-1_{z_{n-k-2}}, \ldots, q-1)
\end{gathered}
$$

In other words, let $i = \alpha(q-1) + j$ for $0 \leq \alpha < n-k-1$ and $0 < j < q$. Then,

$$\boldsymbol{r}_{i,\beta} = \begin{cases} 0 & \beta < z_{n-k-\alpha} \\ j & \beta = z_{n-k-\alpha} \\ q-1 & \beta > z_{n-k-\alpha} \\ 0 & \beta \neq z_\eta \text{ for } \eta = 1, \ldots, n-k+1 \end{cases}$$

Algorithm 2 Find $n-k+1$ root locations from P_1

```
1: function L = FIND_ROOT_LOCATIONS(P_1)
2:     S = 0
3:     for i = 1 to q do
4:         for each cell state vector v in generation i do
5:             S = S + v
6:     L = find(S == 0)
```

To find the root locations, we take the sum of all the cell state vectors in the first q generations and the locations that are zero serve as the root locations as given in Algorithm 2. We give the details of why the algorithm works in the section 3.4.4.1

3.4.3.4 `Create_Subposet`

We take the poset P_1 and add the given root to each cell state variable in the sub-poset. With $i = 1, \ldots, (n-k-1)(q-1)$, we set the last co-ordinate of the update state vector to the generation number. Lastly, we take out the top i generations after the addition to get the sub-poset P_{i+1}.

3.4.3.5 `Join_Vertices`

Finally we join the vertices of the newly created poset P_i to the update code. The root of the vertex P_i is joined to the root of the vertex P_{i-1} by creating an edge between the root of P_{i-1} and P_i. Thus, all the vertices of P_i are one generation higher than the vertices of P_{i-1}.

We call two vertices $\boldsymbol{c}_{i-1} \in P_{i-1}$ and $\boldsymbol{c}_i \in P_i$ corresponding vertices if they are from the same vertex in P_1. In other words, $\boldsymbol{c}_{i-1} = \boldsymbol{v} + \boldsymbol{r}_{i-1}$ and $\boldsymbol{c}_i = \boldsymbol{v} + \boldsymbol{r}_i$ for some $\boldsymbol{v} \in P_1$. Thus, $\boldsymbol{c}_{i-1}$ and $\boldsymbol{c}_i$ have the same positions within the sub-posets P_{i-1} and P_i.

For connecting the rest of the vertices, just like the root, we join the corresponding vertices of P_{i-1} to P_i for each generation. This is best illustrated in the example constructions by black edges between different sub-posets and also Figure 3.8 which shows joining of the roots and corresponding vertices. The edges are connected in the order they appear in each generation from one sub-poset to the other. Each vertex in the sub-poset now connects to one other corresponding vertex and thus, the number of outgoing edges of the vertices have increased from $k-1$ to k for all the vertices. Note that each sub-poset P_i has i generations from $P_0 + \boldsymbol{r}_i$ removed from the top and so, the top generation of P_i does not have any outgoing edges.

The final sub-poset only has the root and thus no outgoing edges. After the final sub-poset is joined, we have the full update code with each vertex has k outgoing edges.

3.4.4 Proofs for the Algorithm

The algorithm given above assumes a certain structure in the poset exists but we have to prove that they do for arbitrary values of n, q, k. We also have to prove that after all the structures are constructed, we have a valid floating code by showing that each update cell increment can be satisfied with a single cell increment.

The structures that we have to verify are:

1. In the algorithm for finding the root locations, we need to prove that we will always find $(n - k + 1)$ zero locations if we sum up all the cell state vectors in the first q generations of P_1.

2. When we generate the sub-posets from the root vectors, we have to guarantee that each cell state vector in the sub-poset does not exist in any other sub-poset.

3. When joining the sub-posets, the edges exists between the two corresponding vertices in the cell poset.

When the above have been proved, we can verify that the sub-posets

$$P_1, P_2, \ldots, P_{(n-k+1)(q-1)+1}$$

are disjoint (in terms of cell state vectors). Each vertex has k outgoing edges that are in the cell poset. After that, we have to prove that the above poset (a sub-poset of the (n, q) cell poset) is an update code by satisfying the property that for each cell state vector that represents a variable state vector, its outgoing edges lead to other cell state vectors that represent all the possible updates state vectors. When the above is proved, we can guarantee a valid update code.

The reason that $n \in \{k, k+1\}$ when $k > 2$ is because it would otherwise violate condition 2 from above regarding the structure of the constructed update code. Some of the subposets generated would have vertices that have already been used before in a previous sub-poset.

3.4.4.1 Root Locations

The lemma below says that there are $n-k+1$ zeros when we sum the first q generations of the sub-poset P_1.

Lemma 3.4.1. *Let F be a $(n-1, q, k-1)$ update code as in the above algorithm and let P_1 be the poset created by adding one co-ordinate at the end of the cell state vector and assigning that co-ordinate 0 throughout the poset. The sum of all the cell state posets in the first q generations of P_1 has zeros at $n-k+1$ co-ordinates.*

Proof. We will prove this by induction on k, the number of variables. For the base case, let $k = 2$ and that P_1^2 be the sub-poset made from a $(n-1, q, k=1)$ update code F by adding a co-ordinate to the end and assigning it 0. Then, P_1^2 has n co-ordinates and

is a sub-poset of the update code $(n, q, 2)$. For the first q generations, we only update the first location and thus, the sum of the first q generations would have zeros in all the co-ordinates except the first. For F, that is $n-2$ and for P_1 that is $n-1 = n-2+1$ which satisfies the case for $k = 2$.

Now, let us assume that P_1^k, a sub-poset of (n, q, k) update code, generated by adding a co-ordinate to a $(n-1, q, k-1)$ update code has the property that summing the first q generations of P_1^k has zeros in $n-k+1$ locations.

Consider P_1^{k+1} generated from the (n, q, k) update code F_k which is a sub-poset of $(n+1, q, k+1)$ update code. The first q generations of the (n, q, k) code is made from P_1^k and $q-1$ other posets derived from by adding a root. We used the last co-ordinate for the first $q-1$ root values. Thus, if we added all the vertices in the first q generations, we would have one less zero location because we have added root vectors that are non-zero in the last co-ordinate. Thus, the number of zeros of the first q generations of F_k is $n-k$. When we create P_1^{k+1}, we add a new zero to co-ordinate and increase the number of zeros. Thus, we have $n-k+1$ zeros in P_1^k. Since, $n-k+1 = (n+1)-(k+1)+1$, this is also true for $k+1$. Thus, by induction, it is true for all values of (n, q, k). □

This gives a simple corollary linking the weight of the vector and zero co-ordinates.

Corollary 3.4.2. *Let $\boldsymbol{v} \in P_1$. If $w(\boldsymbol{v}) \leq q$, then $\boldsymbol{v}$ is zero in $n-k+1$ co-ordinates.*

Proof. From 3.4.1, the sum of all the vectors in the first q generations has zeros in $n-k+1$ co-ordinates. Since $\boldsymbol{v}$ is in the first q generation, it must have zero in $n-k+1$ co-ordinates. □

We can generalize the above lemma to the following to extend to any number of generations:

Lemma 3.4.3. *Let P_1 be the poset as constructed above. The sum of all the cell state posets in the first $d(q-1)+1$ generations of P_1 has zeros at $n-k+2-d$ co-ordinates.*

Note that when $d = n - k + 1$ the total number of zero co-ordinates is 1 and the zero co-ordinate is at the last co-ordinate. When $d = 1$, we get Lemma 3.4.1.

Lemma 3.4.4. *Let Z be the zero co-ordinates of the sum of the first q generations of P_1. If $\boldsymbol{v} \in P_1$ and is non-zero in h of these zero locations, then*

$$w(\boldsymbol{v}) \geq h(q-1) + 1$$

Note that when $h = 0$ and there are no non-zero locations from $\boldsymbol{z}$, then $w(\boldsymbol{v}) \geq 1$ and when $h = n - k + 1$, then $w(\boldsymbol{v}) \geq (n - k + 1)(q - 1) + 1$.

As a consequence, we have the following lemma:

Lemma 3.4.5. *Let $\boldsymbol{c} \in P_i$. Then, $\boldsymbol{c} = \boldsymbol{v} + \boldsymbol{r}_i$ for some $\boldsymbol{v} \in P_i$ and $\boldsymbol{r}_i$ the root that generates P_i. Additionally, $\boldsymbol{v}$ and $\boldsymbol{r}_i$ cannot have two common co-ordinates that are non-zero.*

3.4.4.2 Disjoint Sub-Posets

We next prove that the sub-posets generated do not have a vertex in common. This comes from the way we constructed the sub-posets. From Lemma 3.4.1, we have that there are $n - k + 1$ zero co-ordinates.

For $k = 2$, the roots are constructed so that these zeros are filled from right to left while these co-ordinates are filled up left to right in the update code as we go higher up in the generations. From the construction, there never is a case when these two meet and so, two sub-posets cannot have the same cell state vector.

For $k > 2$ and $n \in \{k, k + 1\}$, it has two zero co-ordinates from Lemma 3.4.1. For the first $q - 1$ updates, the nth co-ordinate is used as the root location and $q - 1$ root vectors created. For the second set of $q - 1$ updates, the height of the generated sub-posets will be of height at most q and thus, the second zero co-ordinate can be used for generating $q - 1$ more roots. The nth location will have been used by the first $q - 1$ updates and so then the second set of updates can use the second root location.

The reason we restrict $n \in \{k, k+1\}$ when $k > 2$ is that while there are $n-k+1$ zero co-ordinates in the first q generations, the number of zeros in the next $q-1$ generations reduces by $k-1$, each variable taking up one more cell. When $k = 2$, this is not a problem because the zero co-ordinates reduces by 1 and we can build t_1 optimal update codes. For $k > 2$, the zero locations are used up at $k-1 > 1$ locations on each $q-1$ updates but we run out of root locations before we can get $n-k+1$ generations.

Lemma 3.4.6. *Let $P_1, P_2, \ldots, P_{(n-k+1)(q-1)+1}$ be the sub-posets as generated above. They do not have a cell state vector (vertex) in common.*

Proof. Assume to the contrary that there are two cell state vectors $\boldsymbol{c}_i \in P_i$ and $\boldsymbol{c}_j \in P_j$ such that $\boldsymbol{c}_i = \boldsymbol{c}_j$ with $i < j$. Then, there must be two cell state vectors $\boldsymbol{v}_1, \boldsymbol{v}_2 \in P_1$ and taking $\boldsymbol{r}_i$ and $\boldsymbol{r}_j$ are the roots of P_i and P_j that

$$\boldsymbol{v}_1 + \boldsymbol{r}_i = \boldsymbol{v}_2 + \boldsymbol{r}_j$$

and equivalently,

$$\boldsymbol{v}_1 - \boldsymbol{v}_2 = \boldsymbol{r}_j - \boldsymbol{r}_i \tag{3.4}$$

We look at two cases.

Suppose $\boldsymbol{r}_i$ and $\boldsymbol{r}_j$ vary at one co-ordinate α only. Since $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ are zero in that co-ordinate, the left hand side of Equation 3.4 would be zero on the α co-ordinate and non-zero on the right. This is a contradiction and $\boldsymbol{r}_i$ and $\boldsymbol{r}_j$ cannot vary in just one location.

Suppose that $\boldsymbol{r}_i$ and $\boldsymbol{r}_j$ vary at more than one location. From Equation 3.4, we have that $\boldsymbol{v}_1 < \boldsymbol{v}_2$ and the increments that take $\boldsymbol{r}_i$ to $\boldsymbol{r}_j$ also takes $\boldsymbol{v}_2$ to $\boldsymbol{v}_1$.

Let us first consider the case of $k = 2$. Let $z_1, z_2, \ldots, z_{n-k+1}$ be the zero locations used to construct the roots. From the construction, we increment by 1 for each zero co-ordinate until it reaches $q-1$. We start from the rightmost z_{n-k+1} and then choose the next left co-ordinate when it is $q-1$. Then, $\boldsymbol{r}_j - \boldsymbol{r}_i$ will be non-zero in the co-ordinates

$z_{s-\lambda}, z_{s-\lambda+1}, \ldots, z_s$ for some $s < n-k+1$ and $\lambda > 1$.

Now, let us consider the sequence of updates from $\boldsymbol{v}_2$ to $\boldsymbol{v}_1$. The increments can happen in co-ordinates not in $z_1, z_2, \ldots, z_{n-k+1}$. But also, after each $q-1$ updates, updates can happen for each additional z co-ordinate going from the left to the right. For example, when we at the zero generation, updates only occur in co-ordinates not in $z_1, z_2, \ldots, z_{n-k+1}$. After the first q updates, updates can also happen in the z_1 co-ordinate. And, then from there, to z_2, z_3 and so on. From Equation 3.4 and that $\boldsymbol{r}_j - \boldsymbol{r}_i$ are non-zero in the co-ordinates $z_{s-l}, z_{s-l+1}, \ldots, z_s$, we have that $\boldsymbol{v}_1 - \boldsymbol{v}_2$ are also non-zero in the same co-ordinates.

From above, we have that

$$w(\boldsymbol{v}_1) > s(q-1)+1$$

and since $\boldsymbol{r}_i$ is non-zero also at the s co-ordinate,

$$w(\boldsymbol{r}_i) > (n-k+1-s)(q-1)$$

Now, looking at the weight of $\boldsymbol{c}_i$,

$$\begin{aligned} w(\boldsymbol{c}_i) = w(\boldsymbol{v}_1 + \boldsymbol{r}_i) &= w(\boldsymbol{v}_1) + w(\boldsymbol{r}_i) \\ &> (n-k+1)(q-1)+1 \end{aligned}$$

which is not possible because the update codes are only up to $(n-k+1)(q-1)+1$ generations. Thus, we have a contradiction.

Thus, $i = j$ and two sub-posets cannot have a cell state vector in common.

Next, when $k > 2$ and $n \in \{k, k+1\}$, there are two root locations and one of them is the nth co-ordinate and let the other one be the sth co-ordinate. For the first $q-1$ updates, we use the nth co-ordinate. For the next $q-1$ locations, note that all the sub-

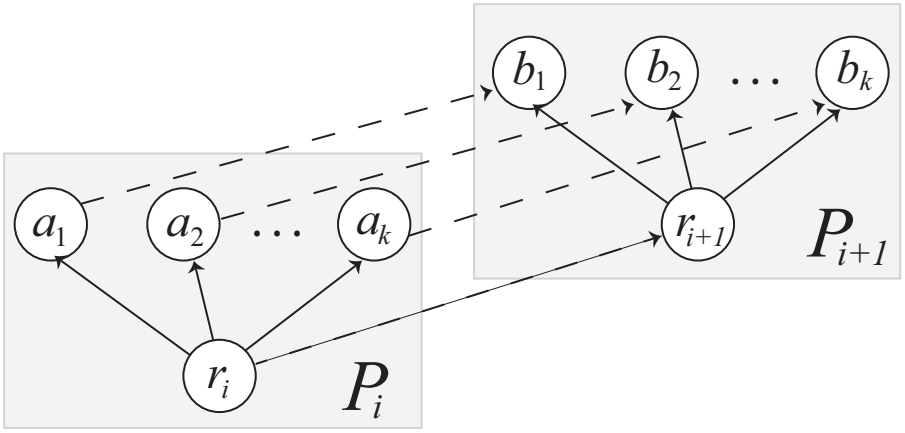


Figure 3.8: Connecting Two Sub-Posets

posets are of height less than or equal to q. Thus, all the sub-posets for the second $q-1$ update are zero in the sth co-ordinate. Thus, we can create $q-1$ roots at the sth location and none of the vectors are are repeated because the sub-posets are non-zero in the root locations. □

3.4.4.3 Existence of the Edges for Joining Posets

We have said in the construction algorithm that we can draw edges between the poset P_i and P_{i+1} but haven't proven that such edges exist the way we have described it. The next lemma states that these edges must exist.

Lemma 3.4.7. *Let $\boldsymbol{r}_i$ be a root and $\boldsymbol{r}_{i+1}$ be the next root by doing a cell increment. Let P_i be the sub-poset generated by $\boldsymbol{r}_i$ and P_{i+1} be the sub-poset generated by $\boldsymbol{r}_{i+1}$, done by adding the root to the sub-poset P_1. We connect an edge between $\boldsymbol{r}_i$ and $\boldsymbol{r}_{i+1}$. Then, there must be an edge between a vertex in P_i to a vertex at P_{i+1}. Furthermore, let $\boldsymbol{v} \in P_1$ and if $\boldsymbol{v} + \boldsymbol{r}_{i+1} \in P_{i+1}$, then there is an edge between $\boldsymbol{v} + \boldsymbol{r}_i \in P_i$ and $\boldsymbol{v} + \boldsymbol{r}_{i+1} \in P_{i+1}$ (i.e. there is an edge between two corresponding vertices in the sub-posets P_i and P_{i+1}).*

Proof. We connect the root of P_i to the root of P_{i+1} as shown in Figure 3.8 by connecting $\boldsymbol{r}_i$ and $\boldsymbol{r}_{i+1}$. Note that $\boldsymbol{r}_{i+1} = \boldsymbol{r}_i + \boldsymbol{\delta}_1$ where $\boldsymbol{\delta}_1$ is some unit increment.

By connecting the roots of P_i and P_{i+1}, $\boldsymbol{r}_i$ is now the common root of $\boldsymbol{a}_1$ and $\boldsymbol{r}_{i+1}$ and they must now have a cover in common from Theorem 3.3.7. Now, $\boldsymbol{b}_1$ covers $\boldsymbol{r}_{i+1}$ and also covers $\boldsymbol{a}_1$ since $\boldsymbol{b}_1 = \boldsymbol{a}_1 + \boldsymbol{\delta}_1$, there is an edge between $\boldsymbol{a}_1$ and $\boldsymbol{b}_2$.

Suppose we have $\boldsymbol{v} \in P_1$. Let $\boldsymbol{c}_i = \boldsymbol{v} + \boldsymbol{r}_i \in P_i$ and $\boldsymbol{c}_{i+1} = \boldsymbol{v} + \boldsymbol{r}_{i+1} \in P_{i+1}$. Note that $\boldsymbol{c}_{i+1} = \boldsymbol{c}_i + \boldsymbol{\delta}_1$ and thus, $\boldsymbol{c}_{i+1}$ covers $\boldsymbol{c}_i$ since $\boldsymbol{\delta}_1$ is a unit increment. Then, we draw an edge between them to denote the update of the kth variable. □

3.4.4.4 Validity of the Update Code

Having established the structural properties of the update code, we have to verify the final step that this is indeed a valid update code. For each cell state vector, we have to be able to increment each of the k update cells by going up one generation in the cell state poset.

Theorem 3.4.8. *The poset and the update map generated as above is a valid update code.*

Proof. We have shown that the update code generated when $k = 1$ is valid. Now, let us assume that the update code for (n, q, k) is valid. We will show that the update code for $(n + 1, q, k + 1)$ is also valid and prove by induction.

Since the $(n + 1, q, k + 1)$ update code is made up sub-posets made from the (n, q, k) code, and by assumption the (n, q, k) code is valid, the first k update cells can be incremented on the $(n + 1, q, k + 1)$ code. We just have to prove that the $k + 1$st update cell increment is possible.

For each cell state vector we construct, we join it to the next sub-poset where the update state vector is exactly the same except the last update cell, as described above. This allows for the increment of the $k + 1$st update cell as we go up one generation and thus, we are able to increment all $k + 1$ update cell at each cell state poset. Therefore, the poset generated by the above algorithm is a valid update code. □

3.4.5 Mappings without Constructing the Poset

The algorithm we have presented above creates an update code as a poset. For large values of (n, q, k), constructing and using an entire poset will not be practical. We want encode, decode and update mappings that can work with cell state vectors and update state vectors only without having to construct the entire poset first.

3.4.5.1 Notes on the Code Construction

The update code we have constructed has a stricter structure than that is required for update codes. Each update state vector is mapped to a single cell state vector. The general update code can have an update state vector mapped to multiple cell state vectors depending on the path it took to get to there. Consequently, our construction algorithm simply boils down to finding a subposet of the cell poset that is isomorphic to the update poset where the all zero update state vector is mapped to the all zero cell state vector.

3.4.5.2 Decode Mapping

We want a decode function that takes a cell state vector and outputs an update state variable or that tells us that the given cell state vector is not assigned to any update state variable. We will first look at $k = 1$ and then give a recursive function for $k > 1$.

Suppose we have the parameters (n, q, k) for the update code and a cell state vector $(c_1, c_2, \ldots, c_n)$ and we want to find the update state vector $(u_1, v_2, \ldots, u_k)$ that corresponds to this cell state vector for the given code parameters.

3.4.5.2.1 $k = 1$ When $k = 1$, the value is simply the number of the generation which can be obtained by the sum of the cell state vector. Thus,

$$v_1 = \sum_{i=1}^{n} c_i$$

Note this is only valid when the cell state vector is being used by the code. We can check if the cell state vector is being used or not by making sure it confirms to how the $k = 1$ cell state vectors are assigned. The first co-ordinate is incremented up to $q - 1$ and then the next until $q - 1$. So, we can just check that the following conditions hold

$$c_i > 0 \quad \Longrightarrow \quad c_{i-1} = q - 1$$

for $i = 2 \ldots n$. If the above condition is violated, then the cell state vector is not assigned to an update state vector.

3.4.5.2.2 $k > 1$ When we have more than one variable, we decode the kth variable and then create another cell state vector for code parameters $(n - 1, q, k - 1)$. We can then recursively decode.

There are $(n - (k - 1) + 1)(q - 1)$ copies of the poset P_1 and as many roots. There are $(n - (k - 1) + 1)$ root locations. We have calculated the roots by summing up the first q levels of the poset but since we don't want to construct a poset, we use a different algorithm to find the root locations.

The root locations are given by the following where we denote the root locations by $z_1, z_2, \ldots, z_{n-k}$,

$$\begin{aligned} z_1 &= n \\ z_2 &= n - (k - 1) \\ z_3 &= n - (k - 2) \\ &\vdots \\ z_{n-k} &= 2 \end{aligned}$$

We then take the vector at the root locations and that will be our root. The sum of

the root will tell us how many times the root has been updated.

$$v_k = \sum_{i=1}^{n-k} c_{z_i}$$

To check if the cell state vector is being used for an update state vector, we recursively go down to the cell state vector for $k = 1$ and if that cell state vector is valid, then the original cell state vector is also valid.

3.4.5.3 Encode Mapping

Given an update state vector, we want to find the cell state vector that encodes the update state vector. From our discussion above, each update state vector in our construction decodes to only one cell state vector.

3.4.5.3.1 $k = 1$ This is the case when we just have one update cell and the update state vector is (u_1). We start from the leftmost cell, increment it until it reaches $q - 1$ and then move to the next cell. Let

$$\eta = \left\lfloor \frac{u_1}{q-1} \right\rfloor$$

Then, we have for the cell state vector $(c_1, c_2, \ldots, c_n)$

$$\begin{aligned} c_1 &= q - 1 \\ &\vdots \\ c_\eta &= q - 1 \\ c_{\eta+1} &= u_1 \mod q - 1 \\ c_{\eta+2} &= 0 \\ &\vdots \end{aligned}$$

3.4.5.3.2 $k > 1$ We have the update state vector $(u_1, u_2, \ldots, u_k)$. Each increment of the update cell is encoded by an increment in the flash cells. For the first update cell, the pattern is increments is the same as for $k = 1$. For the rest of the update cells u_i for $i = 2, \ldots, k$, we increment similarly but from right to left and in the root locations for $(n-i, q, k-i)$ update code. The root locations can be directly calculated as we described in Section 3.4.5.2.2. Thus, we calculate $\eta_i = \left\lfloor \frac{u_i}{q-1} \right\rfloor$ and set the last η_i root locations to $q-1$. Then, the next root location cell is incremented by $u_i \mod q-1$.

3.4.5.4 Update Mapping

Since each update state vector is mapped to a single cell state vector, the update mapping can be inferred from the encode and decode mappings. To update a cell state vector, we decode it to the update state vector and then increment the desired update cell to get the new update state vector. After that, we decode it to a cell state vector. Since each update state vector only maps to a single cell state vector in our construction, there is no ambiguity here. Thus, the update mapping to the new cell state vector can be inferred from just the encode and decode mappings.

3.5 Generalization for $l > 2$ Floating Codes

The construction of the floating code from update codes we have given above is only for binary variables ($l = 2$). To generalize the construction for arbitrary l, we will use the property we prove in Theorem 3.5.1 that a non-binary variable floating code is isomorphic to a binary-variable floating code generated from an update code with adjusted parameters. Then, all we need for constructing floating codes for $l > 2$ is to construct the isomorphic binary $l = 2$ floating code from update codes and then re-label the vector state variables. They then go from an array of binary variables to a different array of l-ary variables.

3.5.1 Isomorphism Between Codes

The following theorem is the basis for constructing floating codes for $l > 2$. It turns out that an t_1-optimal floating code for arbitrary parameters can be made from a t_1-optimal floating code for binary-variable parameters by simply relabeling its vertices for l-ary variables.

Theorem 3.5.1. *Let $n, q, k, l \in \mathbb{N}$ with $n \geq k(l-1)$. Then, there exists a floating code $\mathcal{F}(n, q, k, l)$ with parameters n, q, k, l such that*

$$\mathcal{F}(n, q, k, l) \cong \mathcal{F}(n, q, k(l-1), 2)$$

where $\cong$ denotes an isomorphism and $\mathcal{F}(n, q, k(l-1), 2)$ is a binary variable floating code with parameters $(n, q, k(l-1), 2)$ generated from an update code that we can construct using the above algorithm.

Proof. We will show that an isomorphic floating code with parameters n, q, k and l exists by taking the binary-variable floating code $\mathcal{F}(n, q, k(l-1), 2)$ generated from an update code, relabeling the variable state vectors to l-ary k variables and then, showing that the end-result is a floating code.

Given the parameters n, q, k, l, let $\boldsymbol{\nu}$ be a vertex in the floating code and let $V_2(\boldsymbol{\nu})$ denote the binary variable vector for the vertex $\boldsymbol{\nu} = (u_1, u_2, \ldots, u_{k(l-1)})$. Each vertex in the floating code has $k(l-1)$ covers and let the covers be $\boldsymbol{\nu}_1, \boldsymbol{\nu}_2, \ldots, \boldsymbol{\nu}_{k(l-1)}$.

By our construction, $\boldsymbol{\nu}_1$ represents the update to the first variable u_1, $\boldsymbol{\nu}_2$ update to the second variable u_2 and so on. Thus, the order of the covers denotes which variable is updated and that $\boldsymbol{\nu}_i$ is an update to the ith variable u_i. In notation,

$$[V_2(\boldsymbol{\nu}_i)]_i = u_i + 1 \mod 2$$

Now we relabel each binary variable vector in $\boldsymbol{\nu}$ to l-ary k-variable vectors and denote

the vector by $V_k(\boldsymbol{\nu}) = (v_1, v_2, \ldots, v_k)$. The first $l-1$ covers

$$\boldsymbol{\nu}_1, \boldsymbol{\nu}_2, \ldots, \boldsymbol{\nu}_{l-1}$$

will represent updates to the variable v_1, the next $l-1$ covers

$$\boldsymbol{\nu}_{(l-1)+1}, \boldsymbol{\nu}_{(l-1)+2}, \ldots, \boldsymbol{\nu}_{2(l-1)}$$

updates to v_2 and so on until the last $l-1$ covers

$$\boldsymbol{\nu}_{(k-1)(l-1)+1}, \boldsymbol{\nu}_{(k-1)(l-1)+2}, \ldots, \boldsymbol{\nu}_{k(l-1)}$$

updates to v_k.

For variable v_i, let $\boldsymbol{\nu}_{(i-1)(l-1)+j}$ be the jth cover. The jth cover for variable v_i will represent the update to $v_i + j \mod l$. In notation,

$$\left[V_k(\boldsymbol{\nu}_{(i-1)(l-1)+j})\right]_i = v_i + j \mod l$$

We now need to show that the relabeled code is a valid floating code for parameters n, q, k and l.

The (n, q, k, l) floating code should provide for $k(l-1)$ updates, i.e., $(l-1)$ updates for each of the k variables. Since we used a binary-variable $(n, q, k(l-1), 2)$ floating code for the relabeling, there are enough covers to provide $k(l-1)$ variable updates.

To prove that it is a valid floating code, we have to show that when a vertex is a cover of multiple vertices, all the updates should result in the same variable state vector. In other words, if $\boldsymbol{\psi}$ is a cover of vertices $\boldsymbol{\nu}$ and $\boldsymbol{\omega}$, and $\boldsymbol{\psi}$ is the ith variable update of size δ_i of the variable state vector $\boldsymbol{\nu}$ and jth variable update of value δ_j of the variable state vector $\boldsymbol{\omega}$, the result of both of these updates should result in the same variable state

vector.

For the first generation $g = 1$, there is only the all zero variable state vector which we map to the k-variable zero vector. For $g = 2$, all the vertices cover the zero vector and thus is vacuously true.

Let $\boldsymbol{\psi}$ be a vertex in the $h+1$ generation of the floating code and covers vertices $\boldsymbol{\nu}$ and $\boldsymbol{\omega}$ in generation h. By Theorem 3.3.7, the vertices $\boldsymbol{\nu}$ and $\boldsymbol{\omega}$ also have a common root $\boldsymbol{\sigma}$ that lies in generation $h-1$. The relationship between the vertices are shown in Figure 3.10.

Let the binary variable state vector stored in $\boldsymbol{\sigma}$ be $(u_1, u_2, \ldots, u_{k(l-1)})$. Let $\boldsymbol{\nu}$ and $\boldsymbol{\omega}$ be the update to the ith and jth binary variable of $\boldsymbol{\sigma}$ and that $i < j$. Then,

$$V_2(\boldsymbol{\nu}) = (u_1, u_2, \ldots, \overline{u_i}, \ldots, u_j, \ldots, u_{k(l-1)})$$
$$V_2(\boldsymbol{\omega}) = (u_1, u_2, \ldots, u_i, \ldots, \overline{u_j}, \ldots, u_{k(l-1)})$$

where $\overline{u_i} = u_i + 1 \mod 2$. Now, $\boldsymbol{\psi}$ is a cover of both $\boldsymbol{\nu}$ and $\boldsymbol{\omega}$ and since we constructed a valid binary-variable floating code, we know that updates to both $\boldsymbol{\nu}$ and $\boldsymbol{\omega}$ result in the same variable state vector in $\boldsymbol{\psi}$.

This leaves two choices possible on how the variable vectors of $\boldsymbol{\nu}$ and $\boldsymbol{\omega}$ are updated. Either the ith variable of $\boldsymbol{\nu}$ and jth variable of $\boldsymbol{\omega}$ is updated to result in

$$V_2(\boldsymbol{\psi}) = (u_1, u_2, \ldots, u_i, \ldots, u_j, \ldots, u_{k(l-1)})$$

or, the jth variable of $\boldsymbol{\nu}$ and ith variable of $\boldsymbol{\omega}$ is updated to result in

$$V_2(\boldsymbol{\psi}) = (u_1, u_2, \ldots, \overline{u_i}, \ldots, \overline{u_j}, \ldots, u_{k(l-1)})$$

We will next show that by our construction, only the second type of update is possible.

By our construction, since the update from $\boldsymbol{\sigma}$ to $\boldsymbol{\nu}$ is in the ith variable, the update

from $\boldsymbol{\omega}$ to $\boldsymbol{\psi}$ must also be in the ith variable. The portion of the sub-poset from $\boldsymbol{\sigma}$ to $\boldsymbol{\nu}$ and $\boldsymbol{\omega}$ to $\boldsymbol{\psi}$ are copies of each other by our construction. This is shown in Figure 3.10 as shaded boxes which are built from the same portion of a sub-poset. Thus, it must be that

$$\boldsymbol{\psi} = (u_1, u_2, \ldots, \overline{u_i}, \ldots, \overline{u_j}, \ldots, u_{k(l-1)})$$

Thus, updates from $\boldsymbol{\sigma}$ to $\boldsymbol{\nu}$ to $\boldsymbol{\psi}$ is updates to the ith variable then the jth variable. And, the update from $\boldsymbol{\sigma}$ to $\boldsymbol{\omega}$ to $\boldsymbol{\psi}$ is updates to the jth variable then ith variable.

Now, for our k-ary variables, let

$$i = (l-1)\alpha + \delta_\alpha$$
$$j = (l-1)\beta + \delta_\beta$$

Let $\boldsymbol{\sigma}$ be the l-ary variable vector $(v_1, v_2, \ldots, v_k)$. Then, by our update pattern we have when $\alpha \neq \beta$

$$V_k(\boldsymbol{\nu}) = (v_1, v_2, \ldots, \overline{v_\alpha + \delta_\alpha}, \ldots, v_\beta, \ldots, v_k)$$
$$V_k(\boldsymbol{\omega}) = (v_1, v_2, \ldots, v_\alpha, \ldots, \overline{v_\beta + \delta_\beta}, \ldots, v_k)$$

where $\overline{v_\alpha + \delta_\alpha} = v_\alpha + \delta_\alpha \mod l$. When $\alpha = \beta$, we have

$$V_k(\boldsymbol{\nu}) = (v_1, v_2, \ldots, \overline{v_\alpha + \delta_\alpha}, \ldots, v_k)$$
$$V_k(\boldsymbol{\omega}) = (v_1, v_2, \ldots, \overline{v_\alpha + \delta_\beta}, \ldots, v_k)$$

Next, updating $\boldsymbol{\nu}$ in the jth location and $\boldsymbol{\sigma}$ in the ith location, we have

$$V_k(\boldsymbol{\nu}_j) = (v_1, v_2, \ldots, \overline{v_\alpha + \delta_\alpha}, \ldots, \overline{v_\beta + \delta_\beta}, \ldots, v_k)$$
$$V_k(\boldsymbol{\omega}_i) = (v_1, v_2, \ldots, \overline{v_\alpha + \delta_\alpha}, \ldots, \overline{v_\beta + \delta_\beta}, \ldots, v_k)$$

when $\alpha \neq \beta$ and when $\alpha = \beta$, we have

$$V_k(\boldsymbol{\nu}_j) = (v_1, v_2, \ldots, \overline{v_\alpha + \delta_\alpha + \delta_\beta}, \ldots, v_k)$$
$$V_k(\boldsymbol{\omega}_i) = (v_1, v_2, \ldots, \overline{v_\alpha + \delta_\beta + \delta_\alpha}, \ldots, v_k)$$

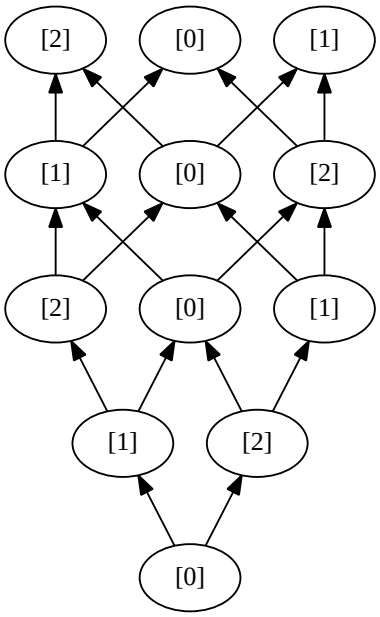
Thus, in both cases, $V_k(\boldsymbol{\nu}_j) = V_k(\boldsymbol{\omega}_i)$ and thus, $V_k(\boldsymbol{\psi})$ represents the same vector state variable. Thus, the relabeled floating code is a valid (n, q, k, l) floating code. Because we constructed our (n, q, k, l) code by simply relabeling the binary-variable floating code, it is isomorphic to the binary-variable floating code.

□

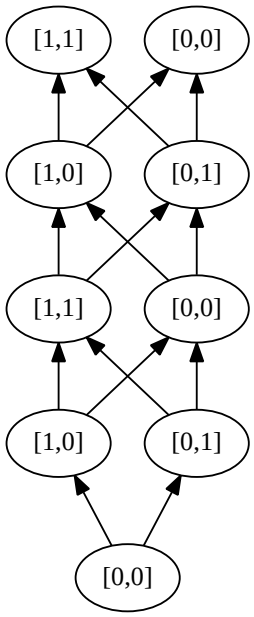
Note that while the floating codes are isomorphic, the variable posets are not isomorphic. For example, when $k = 1, l = 3$, after the second generation, each generation has 3 variable state vectors (see Figure 3.9a) whereas when $k = 2, l = 2$ has only has 2 variable state vectors (see Figure 3.9b). From above, $\mathcal{F}(n, q, 1, 3) \cong \mathcal{F}(n, q, 2, 2)$ and so, the floating codes are isomorphic. This is illustrated in Figure 3.9 which shows the two different variable posets which has the same isomorphic floating codes in Figure 3.9c and from our previous example Figure 3.5a.

3.5.1.1 t_1-Optimality

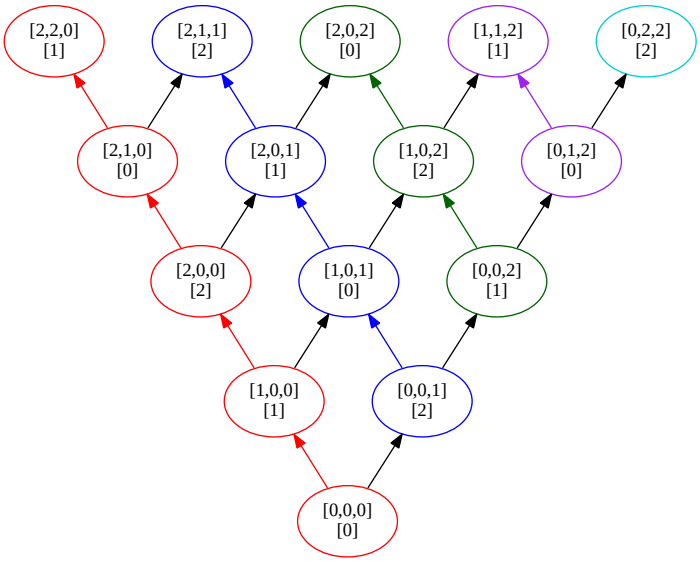
Note that the floating code for $l > 2$ from Theorem 3.5.1 is also t_1-optimal and has deficiency of $O(qkl)$. To see this, note that by the t_1-optimality definition from Section 3.2.7.1, the floating code needs to provide $[n - k(l-1) + 1](q-1)$ updates via single cell increments. The floating code $\mathcal{F}(n, q, k(l-1), 2)$ provides $[n - k(l-1) + 1](q-1)$ updates via single cell increments which is exactly the number of single cell increment updates that is required for optimality for the general $l > 2$ case. Thus, all the codes from Theorem 3.5.1 for $l > 2$ are also t_1-optimal.



(a) Vector Poset For $k = 1, l = 3$



(b) Vector Poset For $k = 2, l = 2$



(c) Floating Code For $k = 1, l = 3$

Figure 3.9: The vector posets for $k = 1, l = 3$ and $k = 2, l = 2$ are not isomorphic but their floating codes are isomorphic

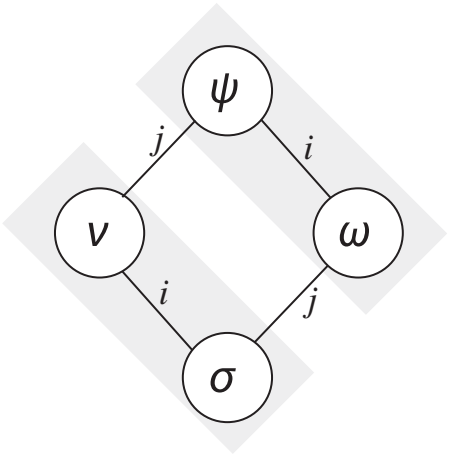


Figure 3.10: Relationship of vertices used in proof of Theorem 3.5.1

3.5.1.2 Floating Codes from Update Codes

For the relabeling algorithm to work, the floating code must be constructed from an update code. The reason is that, in floating codes using an example, the update state vectors $(1, 3)$ and $(3, 1)$ would represent the same variable state vector $(1, 1)$. Then, we could have an edge to both $(1, 3)$ and $(3, 1)$ from $(1, 2)$ in floating codes which is $(1, 0)$ in the floating code. However, such an edge would not be permissible in an update code.

3.5.2 Relabeling Algorithm

The relabeling algorithm works by utilizing the structure of the isomorphic binary-variable floating codes generated from update codes. In the first step, for the first generation which only has a single vertex, this is set to the zero variable state vector with k variables. Next, the algorithm labels each generation after the first from the generation below it. The floating code $\mathcal{F}(n, q, k(l-1), 2)$ has vertices with $k(l-1)$ outgoing edges and hence, $k(l-1)$ covers. The order in which the covers are connected to the vertices will determine the labeling.

Suppose we have a l-ary variable state vector with k variables $(v_1, v_2, \ldots, v_l)$. For each covering vertex, we will label them in the following way in the order they are in the

floating code :

$$
\begin{gathered}
(\overline{v_1+1}, v_2, \ldots, v_k) \\
(\overline{v_1+2}, v_2, \ldots, v_k) \\
\vdots \\
(\overline{v_1+(l-1)}, v_2, \ldots, v_k) \\
\vdots \\
(v_1, \overline{v_2+1}, \ \ldots, v_k) \\
\vdots \\
(v_1, v_2, \ldots, \overline{v_k+(l-1)})
\end{gathered}
$$

where $\overline{v_i+j} = v_i + j \mod l$.

The relabeling algorithm is given in Algorithm 3. The floating code is given by F whose first index indicates the generation and the second index the vertex in the generation. The function `Get_Covers` gets the covers of a vertex which are used in the floating code and vertex indices. The variable λ decides which cover to update and i_2 and i_3 determine which variable to update and by how much.

Algorithm 3 Relabel a $(n, q, k(l-1), 2)$ to (n, q, k, l) code

```
1: function RELABEL_CODE(F)
2:     for i = 1 to t do (each generation in F)
3:         for each vertex ν in generation i do
4:             c = Get_Covers(ν)
5:             for i_2 = 1 to k do (each variable)
6:                 for i_3 = 1 to l − 1 do
7:                     λ = c[i_2(l − 1) + i_3]
8:                     F(i + 1, λ) = ν
9:                     F(i + 1, λ)_{i_2} += i_3 mod l
```

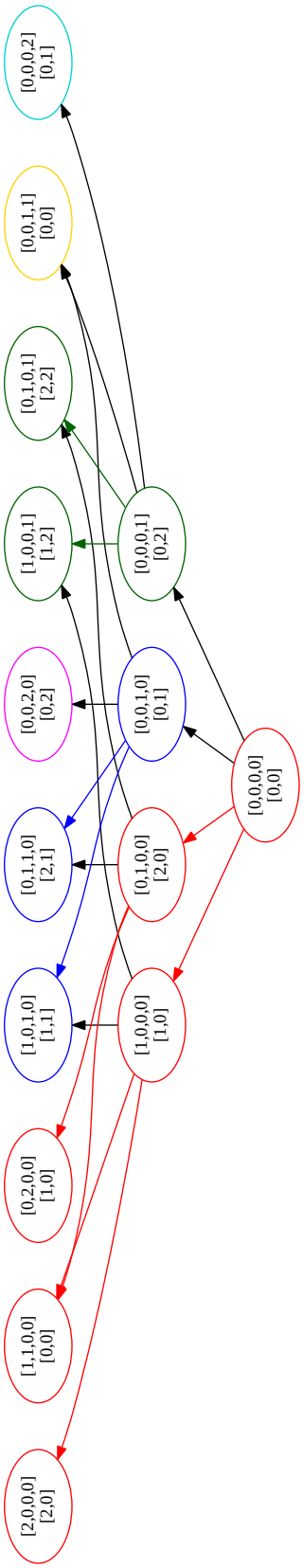


Figure 3.11: $(4, 3, 2, 3)$ floating code with two ternary-valued variables

3.5.3 Examples

The $(3, 3, 1, 3)$ floating code for single variable with $l = 3$ is given in Figure 3.9c. The floating code for two variables with $l = 3$ ternary-variables is given in Figure 3.11 with parameters $(4, 3, 2, 3)$. This floating code is a relabeling of the binary-variable floating code with 4 variables that has parameters $(3, 3, 4, 2)$.

While it is possible to create the $l > 2$ floating code by generalizing the construction algorithm for $l = 2$ and constructing the floating code for (n, q, k, l) by using the $(n - (l - 1), q, k - 1, l)$ floating code, it becomes much more complicated. On each generation, we would have to add multiple copies of the sub-posets and the added sub-posets would have to be joined to previous sub-posets in differing ways. Figure 3.9c shows the $(4, 3, 2, 3)$ code constructed from $(2, 3, 1, 3)$ code with different colors denoting the different sub-posets. We add two copies of sub-posets in the second generation and three copies in the third. Note in the third generation, the new sub-posets are joined twice in some cases and only once in others.

Here, we have presented update codes, a class of floating codes. We gave an algorithm for the construction of t_1-optimal update codes for certain parameters which we derived from the poset structure of the code. We then examined the isomorphism properties of update codes to show that binary floating codes from update codes are isomorphic to l-ary floating code.

CHAPTER 4

Minimizing Write Amplification on Offline Workloads

4.1 Introduction

Using the disk interface for flash memory produces *write amplification* [53], an undesirable by-product of translating write, update and delete operations to native flash memory operations. Write amplification occurs due to the block erase before program architecture and asymmetry of flash memory's program and erase operation sizes. Data is programmed in chunks called pages but this is only possible after it has been erased as a block of 16-128 pages. Most blocks contain a mix of valid and obsolete pages. When a block is going to be erased, it must have its valid pages copied somewhere. If this internal copying is not done before the erase, the valid data is lost after the erase. The result is that the total number of system writes is now greater than the number of user writes because of the internal copying. Therefore, the number of user writes has now been amplified and this amplification factor is termed write amplification.

Write amplification is a parasitic drain on SSD resources and the extraneous writes waste precious time and cuts down the lifetime of the SSD. Thus, reducing write amplification will increase system endurance and performance. The performance increase comes from not having to spend the extra time programming the extra writes as well as saving the time from not having to do the extra block erases caused by the extra writes. Endurance would also increase by not having to perform extra block erases. Each erase operation causes stress to the flash memory storage structure which then wears out after

about 10,000-100,000 erases. When it is worn out, it cannot reliably hold data and so, reducing the extra erases from write amplification increases the lifetime and the endurance of the SSD. As such, reducing write amplification is a major SSD design goal.

4.1.1 Data Placement and Layout Management

One technique to reduce write amplification is to manage flash memory data placement. On first impression however, data placement should not matter in flash memory. Data can be placed anywhere on the device and later retrieved at a fixed constant response time, i.e., no location based latency. However, the technique to reduce write amplification through placement is to group data by when it is expected become invalid. If the data pages in a block are invalidated as close in time as possible, minimal internal copying would be required before the block is erased. Minimal internal copying translates to minimal write amplification.

All the data placement decisions are made by the SSD and to see this, we have to look at the architecture of the SSD. To provide the update operation, SSDs must use an out-of-place update system. In this system, all data addresses are logical and the connection to the real physical location of the data is through an internal map in the SSD. Hence, a data update is translated to a clean page write and then an update of the internal map. Thus, it is up to the SSD to choose the physical location of the clean page to write to and therefore, all data placement decisions are fully determined by the SSD.

Here, we study minimization write amplification through data placement. We define *layout management* as a logical division of the functions of the SSD that deals with data placement, i.e., it makes the decisions on where to place a particular piece of data in the vast gigabytes of storage that is available. We investigate the theoretical limits of write amplification minimization by using offline workloads and algorithms. The purpose of the study is to find out how much write amplification reduction can be done using layout management.

Before designing an algorithm or implementing a method to reduce write amplification, it would be ideal to know how much reduction is actually possible by knowing the limits of write amplification minimizaton. Write amplification is inherently caused by two interacting factors, first flash memory's block erase before program architecture and second, the lack of information regarding the nature of the data that is stored together in a block. By using offline workload data, we can remove the second factor as we have perfect information and can calculate the write amplification caused only by the inherent flash memory architecture. We do this by using layout management algorithms that decide which blocks to store which data.

Ideally, we would like to get zero write amplification for any workload given particular flash memory parameters (size, over-provisioning) using some layout management algorithm but we prove in Section 4.3 that this is not possible. Furthermore, we show that the minimum over-provisioning amount that guarantees zero write amplification for any workload is the trivial over-provisioning amount, i.e., the number of blocks equaling the number of sectors to be stored. Finally, we give an estimation algorithm using decomposition of workloads into sub-workloads that gives an estimate of the minimum write amplification for an offline workload and flash memory.

In Section 4.4, we discuss the general layout management problem: given a flash memory configuration and a workload, what is the least write amplification possible using some layout management algorithm? The question of if the general layout management problem is NP-hard is an open problem but regardless if it is NP-hard or not, we still efficient algorithms to estimate the minimum write amplification because workloads that we are interested are hundreds of millions of operations in length and even polynomial time algorithms would be too slow. In Section 4.6, we describe a technique for workload decomposition that we use to give an algorithm for estimating the general layout management and write amplification minimization problem. In Section 4.7, we describe the estimation algorithm and in Section 4.8 give the results from experiments on the

estimation algorithm using synthetic and trace workloads.

This estimate of the limits of write amplification minimization would give an algorithm developer looking to reduce write amplification a general idea of what is inherently possible using data placement algorithms. This also gives an idea of the effectiveness of the developer's algorithm since just looking at the percentage decrease in write amplification or a similar metric does not really give a good idea of it because it can vary a lot between workloads and flash memory configurations. While the algorithm that we present here is just an estimate and only gives a rough idea of the absolute minimum write amplification that is possible, we show that large reduction in copybacks is possible especially for higher over-provisioning systems and write amplification can be pushed down to zero well below the trivial over-provisioning amount.

4.1.2 Related Work

Write amplification was introduced and its system properties explored in [53] and the authors have followed it up with data placement algorithm (conceptually similar to layout management that we describe here) called container marking that reduces write amplification as well as performs wear leveling [54]. The fundamental write performance have also been studied in [119].

The idea of reducing copybacks have been studied much earlier and were called by various names like cleaning efficiency [1] and cleaning costs [50, 43].

Mathematical models of write amplification have been developed in [107] from experimental outcomes and [14, 90, 30] from basic principles. However, the model from basic principles suffer from state explosion which is not computationally feasible to be calculated for large number of blocks.

Managing data of different invalidation times have been explored as static and dynamic data in [1] but not as offline data that we view here. Using data characteristics to perform wear leveling in flash memory have been studied in [1, 43, 21]. Other methods of reducing

write amplification like using coding schemes has been given in [60]. Workload analysis have been done to find patterns [104] and good IO parameters [81].

The general flash memory and SSD architecture has been analyzed in [78], the multi-level parallelism of SSDs analyzed in [55, 102, 88] and other issues in flash memory SSDs like reliability [59], scheduling [11] and object based file systems [124].

4.2 Write Amplification (WA)

Storage systems and magnetic hard disks provide an interface where it is presented as a very large, linear array of sectors. When performance is required, an unwritten handshake between the storage system and operating system is followed where it is assumed that the sequential sectors are also physically sequential and that there is no mechanical latency when accessing sequential sectors. Disk interfaces like SATA are been built around this linear array assumption. File system and database system IO routines written with the hidden handshake in mind for high performance by sequentializing data accesses and writes. Since flash memory does not have location-based latency, this unwritten handshake does not play a part. However, in order to support the layers of existing software and hardware above it, it must support the linear array interface.

When SSDs present flash memory as a linear array of updateable sectors, the underlying architecture results in creating write amplification in the system. Here, flash memory's characteristics of erasing before writes and the asymmetry of erases and writes are completely hidden from the user. The user sees flash memory as a linear array of updateable sectors and gets no knowledge of the underlying structure of flash memory. Write amplification, in some ways, is the cost of maintaining the illusion that flash memory is a linear array of updatable sectors.

Rewriting every piece of higher level IO routines such that it reflects flash memory architecture would in effect minimize write amplification because the user is aware of

their activities that cause copybacks, which leads to write amplification, and would design to reduce it. However the current view of storage as an array of updateable sectors is ingrained and is also a very simple and convenient abstraction, and so, a complete redesign of IO routines would be too difficult to achieve in reasonable time. Additionally, before even contemplating rewriting all the IO routines to use flash memory conventions, we have to investigate the limits of the current array interface design. Without knowing its limits, we cannot conclusively say whether there is a better design or algorithm that could possibly achieve as low a write amplification as flash memory exporting the linear array of updatable sectors interface.

The unwritten handshake for magnetic hard disks is not a performance trick that applies to flash memory and no similar rules specifically designed for flash memory exists that we know of. Sequential sectors do not necessarily correspond to sequential pages in the block and there is no mechanism in the interface to enforce this. Additionally, the unwritten handshake is not emulated in SSDs because it does not affect read speeds or throughput because there is no mechanical latency delay in flash memory. In this regard, flash memory SSDs are even more opaque about the internal structure than magnetic hard disks. All that the user sees is a sector array without any inter-sector structure while all the SSDs sees is a stream of operations from the user.

Put in another way, the information that the file system or other higher level IO application has on the data is not sent through the SSD interface. The requests are broken down into a sequence of sector operations and no information regarding the sectors is available for use by the SSD. For example, the information regarding which sectors belong together into a single file would be useful because these sectors are likely to be deleted or updated together. If the SSD has this information, it can group these sectors into pages of the same block. This would end up reducing write amplification because different kinds of data will not be mixed up in the same block. By using methods like these, the linear array interface can be made more efficient without requiring the user to

know flash memory architecture and IO conventions.

Here we use offline workloads where we know the entire workload beforehand and this is to emulate sending all the information of the workload through the interface. Then, if perfect information were available to the SSD, then by using optimal algorithms for layout management, we can estimate the limit of how much we can reduce write amplification. For offline optimal algorithms, the SSD knows when the next write to update a sector is going to invalidate a page. With this knowledge, the SSD can put sectors that will be invalidated around the same time into pages of the same block and when the block is to be erased, there will be minimal need for copybacks as all the pages in the block would be have been invalidated together. With this perfect information, we can estimate the lower bound of write amplification for the given workload and SSD settings. Thus, offline workloads will give us an estimate of the increase in write amplification from the loss of information in the user-SSD interface and hence, the limits of reducing write amplification through layout management.

Finding the theoretical limits of layout management has a multitude of practical uses. It gives us a clear idea of the cost of the linear updateable array architecture for SSDs without the cost from the inefficiencies of the current designs and algorithms. In other words, it gives us by how much we can improve current algorithms for reducing write amplification by possibly finding smarter designs. Since SSDs are equipped with a powerful controller, it can infer information from the sequence of operations by analyzing the workload. Alternatively, SSD designers can also implement methods of transferring the information through some sort of API on top of the array interface. If the limits of layout management are not good enough, then we can propose a different interface for SSDs. Thus, by finding the limits of reduction of write amplification by layout management, we have a method to analyze the effectiveness of new designs or algorithms aiming to reduce write amplification.

4.2.1 System Write Amplification

For magnetic hard disk based storage systems, the total number of system writes is less or equal to the number of user issued writes (the less than part owing to caching). As we have discussed, *copybacks* increase the number of system writes and so, the number of system writes becomes greater than the number of user issued writes. Hu et al. [53] defined this factor of amplification in user writes by *write amplification*.

Putting it in an equation, let u be the total number of user issued writes, s be the total number of system writes and b the total number of internal copybacks. The write amplification a can then be expressed as

$$a = \frac{s+b}{u}$$

For simplification, let us assume that $u = s$, i.e., each user issued write turns into a system write. Then,

$$a = \frac{u+b}{u} = 1 + \frac{b}{u} \tag{4.1}$$

Thus, write amplification is directly related to the number of internal copybacks.

The average number of internal copybacks per block is given by the measure VPE (valid pages on erase).

$$v = \frac{b}{b+u}$$

4.2.2 Over-provisioning

An important factor that determines the number copybacks is over-provisioning. Over-provisioning is a technique where the capacity of the SSD is displayed as being much lower than the actual capacity. The reasoning is that as more space is left on the flash memory, the blocks get more time between block erasures, and so, get more time for pages in the block to become invalid and so have less valid pages to be copied back when the block is

garbage collected.

There was two ways to denote the over-provisioning of an SSD, the over-provisioning proportion and over-provisioning factor. Over-provisioning proportion p is defined as the proportion of the flash memory that is occupied. For example, when $p = 0.95$, 5% of the flash memory is unused or has invalid data on it and when $p = 0.5$, half the flash memory is either unused or has invalid data. Over-provisioning factor o is the ratio of the actual capacity and the advertised capacity. For example, when $o = 2$, the actual capacity is twice that of the advertised capacity.

4.2.3 Garbage Collection

Garbage collection involves finding a block based on some criteria, copying the valid pages in the block elsewhere and then erasing the block. One such criterion that works to minimize write amplification is to choose the block with the lowest number of valid pages. In this way, we minimize the number of copybacks for that garbage collection step.

However, for practical implementations, garbage collection can also factor in wear leveling by considering the number of erases of the block or the time since last erasure on the block. For our purposes, we only consider the garbage collection method of choosing the block with the lowest number of valid pages.

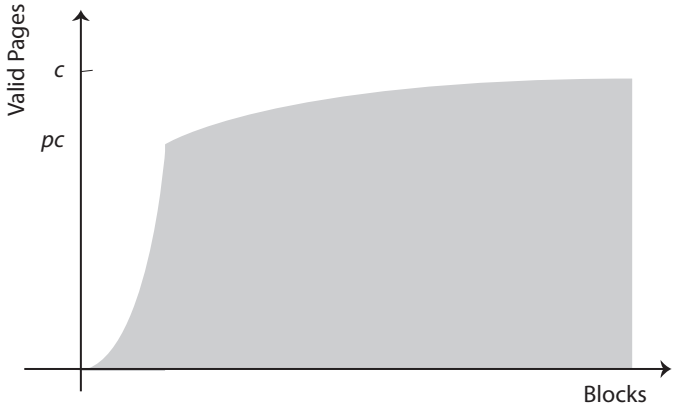
4.2.4 Worst Case Scenario

The worst case for write amplification is when the valid pages are distributed evenly across all the blocks for each garbage collection step. Let c be the number of pages per block. So, if flash memory has p proportion of flash memory occupied, then the worst case is if each block has at least $\lfloor pc \rfloor$ pages occupied. The garbage collector then has to choose a block with $\lfloor pc \rfloor$ valid pages and create a copyback of $\lfloor pc \rfloor$ pages.

Figure 4.1a shows the worst case graphically as a valid pages distribution. Each block has exactly pc valid pages for each garbage collection cycle.

(a) Worst case distribution of valid pages for write amplification

(b) Best case distribution of valid pages for write amplification



(c) Alternate best case distribution of valid pages for write amplification

Figure 4.1: Valid Page Distributions and Write Amplification

To use it in equation 4.1, for every c writes, we have pc copybacks and so only $c - pc$ user writes. Then, the write amplification is given by

$$a = 1 + \frac{pc}{c - pc} = \frac{1}{1 - p} \tag{4.2}$$

and

$$v = \frac{cp}{c} = p$$

From equation 4.2, we see that for flash memory half full, the worst case is copybacks being as much as writes. For $p = 0.9$, the copybacks are about 10 times the number of user writes and for $p = 0.95$, write amplification is a massive 20.

4.2.5 Best Case Scenario

The best-case scenario is when during every garbage collection step there is a block with no valid pages in it and all the pages contained in the block are invalid. This can be illustrated by figure 4.1b. To have such blocks with no valid pages, some blocks have to have all valid pages. In such a situation, there is no write amplification and $a = 0$.

Note that in both graphs in figure 4.1a and 4.1b, the area under the curve (denoted by the grey region), which gives the number of valid pages, is the same in both graphs. Additionally, only a single block has to have zero valid pages on each and every garbage collection cycle for write amplification to be zero and so the valid page distribution graph can look different and still be optimal as in Figure 4.1c.

4.2.6 Flash Memory Terminology and Notation

Flash memory is made up of n *blocks* and each block has c *pages*. The page is the smallest read and program unit while the block is the smallest erase unit. The SSD interface shows the storage memory as a linear array of m *sectors*. The size of a sector is the same as the size of a page. However, the capacity of flash memory is under-advertised to provide

over-provisioning and $m < nc$.

4.3 Offline Workloads

In flash memory SSDs, all that the user sees is a sector array without any inter-sector structure. At the same time, all the SSDs sees is a stream of operations from the user. Thus, the information that the file system or other higher level IO application has on the data is not sent through the SSD interface. The requests are broken down into a sequence of sector operations and no information regarding the sectors is available for use by the SSD. For example, the information regarding which sectors belong together into a single file would be useful because these sectors are likely to be deleted or updated together. If the SSD has this information, it can group these sectors into pages of the same block. This would end up reducing write amplification because different kinds of data will not get mixed up in the same block.

We use offline workloads where we know the entire workload beforehand and this is to emulate sending all the information of the workload through the interface. If perfect information on the workload were available to the SSD, it can put sectors that will be invalidated around the same time into pages of the same block. When the block is to be erased, there will be minimal need for copybacks as all the pages in the block would be have been invalidated close together. With this perfect information, we can estimate the minimum amplification for the given workload and SSD settings.

When the workload is offline, the layout management algorithm will be able to look at any operation in the future while processing the current operation. If an algorithm is able to achieve zero write amplification for a given workload and flash memory configuration, it will be an offline optimal algorithm as zero write amplification is the best we can get. If not, it will at least give an upper bound on the least possible write amplification for the workload.

For simplification, we assume that for each time index t, there is only one write operation. We can make this simplification because we are only concerned about write amplification and so, can ignore the time between the operations. We ignore read operations as well because they do not play any part in write amplification. We ignore delete operations because they change the amount of free space in the flash memory by invalidating pages which we model by over-provisioning (though delete operations will be used later in the estimation algorithm).

4.3.1 A Zero Write Amplification Layout Management Algorithm Does Not Exist for Any Offline Workload

The first question to examine is whether given any offline workload, if there exists a layout management algorithm that will give us zero write amplification. We will give a counter-example to show that this is not the case.

Theorem 4.3.1. *Given an offline workload and a flash memory configuration, a layout management algorithm that gives zero write amplification cannot be guaranteed.*

Proof. Let us consider a flash memory with n blocks with c pages per block that stores $(n-1)c$ sectors. Next, consider the workload where one of the sectors, say sector s, is static and is never updated after it is written. Next, we have to show that for any layout management algorithm, the write amplification cannot be zero.

Let us assume that there is a layout management algorithm that gives zero write amplification. When sector s is written, it must be written to a page in some block, say block b. Now, from our assumption, that there is zero write amplification and that sector s is static, block b cannot be erased after sector s is written to it. Otherwise, it would force the copyback of sector s and create a positive write amplification.

If block b is never erased, then this leads to a contradiction. When block b is never erased, it cannot be used. This leaves us effectively $n-1$ blocks to store $(n-1)c$ sectors.

However, this is not possible because each write would require erasing a block and copying back all the sectors in the block except the one being updated. The sector copyback would result in positive write amplification and thus, a contradiction. □

Thus, even if we have an offline workload and we can choose any layout management algorithm we want, achieving zero write amplification is not possible.

In broader terms, write amplification is an inherent property of flash memory and not of any architectural designs for the disk system.

4.3.2 The Minimal Over-Provisioning for a Zero Write Amplification Layout Management Algorithm to Exist is the Trivial Over-Provisioning

While the example above showed that there exists an offline workload that can have non-zero write amplification for any layout management algorithm, we can turn it into a zero write amplification workload by simply adding a block to the flash memory. Therefore, the next question is, is there a minimum over-provisioning amount that will give us zero write amplification for any offline workload for some layout management algorithm? In other words, is there a minimum over-provisioning that if provided can we guarantee zero write amplification using an any offline workload?

4.3.2.1 Trivial Over-Provisioning

Given m sectors, if we have m blocks to store the sectors, we can zero write amplification. We would do this by using each block to store one sector only.

This is clearly not a practical method since we have an over-provisioning factor of c. A flash memory with 128 pages per block would only advertise $\frac{1}{128}$th of the flash memory as its capacity which is less than 1% of the available memory.

We want to find the smallest over-provisioning amount such that there exists a layout

management algorithm that will give us zero write amplification for any offline workload. However, we will show that we cannot improve upon the trivial over-provisioning factor of c, i.e., using m blocks to store m sectors. We will do so by constructing a workload that requires m blocks for m sectors.

Theorem 4.3.2. *Given an offline workload, the minimal over-provisioning that guarantees the existence of a layout management algorithm that produces zero write amplification is the trivial over-provisioning amount, i.e. m blocks for m sectors for an over-provisioning factor of c.*

Proof. We prove by constructing a workload with m sectors that requires m blocks in the flash memory to achieve zero write amplification. We will define the workload recursively and prove that it requires the m blocks by induction. First, we give the notation, then the recursive definition of the workload and then the proof.

4.3.2.1.1 Notation Let w_i denote a write operation to sector i. Then, w_1 would denote a write to sector 1. Let w_i^j denote j w_i operations in a row.

We denote a workload W as a sequence of write operations. Though for each workload, we give a sequence of writes, we will assume that the sequence is repeated a few number of times.

We assume a flash memory with blocks $B_1, B_2, \ldots$ with c pages per block. The sectors are denoted as $s_1, s_2, \ldots, s_m$ where m is the number of sectors we want to store in the flash memory.

4.3.2.1.2 Workload Definition Each workload W_i involves writes to the first i sectors, $s_1, \ldots, s_i$. Let the workload W_1 be the workload w_1 where writes are only to one sector s_1. Let the workload W_2 be the workload $w_2 W_1^c$ or $w_2 w_1^c$ and in W_2, the writes are for s_1 and s_2. Similarly, we can define W_3 which is a workload consisting of 3 sectors. We take $W_3 = w_3 W_2^c$ which when expanded comes to $w_3 (w_2 w_1^c)^c$.

Thus, we can recursively define out workload as follows

$$W_i = \begin{cases} w_i W_{i-1}^c & i > 1 \\ w_1 & i = 1 \end{cases}$$

4.3.2.1.3 Induction Proof: We need to show that for any workload W_n, we need n blocks to achieve zero write amplification (regardless of the layout management algorithm used).

We have zero write amplification if during at any time we run out pages to write to, there is a block that has no valid pages on it.

We present the proof as induction with first the base case and then the inductive step.

4.3.2.1.4 Base Case We require at least 2 blocks for W_2 because when we have two sectors s_1 and s_2, it is impossible for them to be both invalid at the same time. So, from the above condition, we must have two blocks to achieve zero write amplification.

Next, let us consider the workload W_3. Suppose we just use 2 blocks for W_3. To have zero write amplification, whenever a new block is required, both the valid sectors must reside on the same block.

First, we put sector s_3 in block B_1 (without loss of generality - s_3 can be put in B_2 and the same arguments hold). For us to have zero write amplification, this block cannot be erased until w_3 is encountered again because B_1 has the valid sector s_3 during all that time. But, during that time we must be able to write $(w_2 w_1^c)^c$.

During that time, we have to make c writes of sector s_2 and c^2 writes of sector s_1 totalling $c(c+1)$ writes. The number of writes between the first w_3 and the second w_3 is $c(c+1)+1$. Assuming that the blocks are clean when we start, we need to perform at least c block erasures to make the $c(c+1)+1$ writes. When we erase a block, both the valid sectors must be on the same block to give zero write amplification. However, this is not possible because block B_1 only has $c-1$ slots left while we need to store a valid

sector in block B_1 c times for the c block erasures.

Thus, we have a contradiction and so we cannot have zero write amplification using only two blocks. Thus, three blocks are required for W_3.

4.3.2.1.5 Inductive Step Let the workload W_k be a sequence of write operations for k sectors $(s_1, s_2, \ldots, s_k)$ that requires k blocks $(B_1, \ldots, B_k)$ to achieve zero write amplification and is constructed recursively

$$W_k = w_k W_{k-1}^c$$

So, let $W_{k+1} = w_{k+1} W_k^c$. We need to show that $k+1$ blocks are required for W_{k+1}.

Let us assume that zero write amplification can be achieved for W_{k+1} using k blocks. Suppose without loss of generality, we put sector s_{k+1} in block B_1. Then, the block cannot be erased until the next w_{k+1} operation without resulting in a positive write amplification. However, before the next w_{k+1}, we must process the workload W_k^c. Now, if W_{k+1} can also be processed using only k blocks, this means that W_k^c is being processed with $k-1$ blocks that can be erased and an additional $c-1$ writes to the kth block.

We next examine what this implies for W_k. By our assumption, W_k can be processed by k blocks with zero write amplification. From above, we see that W_k can be processed using $k-1$ blocks and some writes to the kth block. However, for processing W_k c-times, we only wrote to the kth block $c-1$ times. This means that when we processed W_k one of the times, it only required $k-1$ blocks. This then implies that we can process W_k using $k-1$ blocks contradicting out previous assumption.

Thus, we have a contradiction and our assumption that we can process W_{k+1} using k blocks is false and we need $k+1$ blocks for W_{k+1}. Thus, W_n requires n blocks to store with zero write amplification and the trivial bound of using m blocks for m sectors cannot be improved upon.

□

4.3.3 Time to Invalidation (TI)

Another way to view offline workloads is not as the full sequence of writes but each write operation coming in with the exact time in the future it will be invalidated. Thus, when a write comes in, we are also given a TI (time to invalidation) that tells us when the sector for the write will be invalidated in the future. The sequence of operations can be calculated from the TIs and vice versa. We denote each write operation as (s_t, i_t) where s_t gives the sector and i_t gives the TI.

When the sector data is written to disk, we calculate the *invalidation time (IT)* which is the exact time the sector will be updated. The units of both IT and TI are the number of operations into the future.

4.4 The General Offline Layout Management Problem

We have shown that there is no layout management algorithm that can achieve zero write amplification for any workload and any over-provisioning in Theorem 4.3.1, and that the smallest over-provisioning that guarantees no write amplification is the trivial over-provisioning in Theorem 4.3.2. The above problems tried to find a layout management algorithm that works for all possible workloads and perhaps that was the reason for the negative results. Thus, the next step is to look at the problems that relax this and study the problems with fixed workloads. Hence, we will have a fixed workload and some flash memory configuration (number of blocks, pages per block and sectors to be stored with over-provisioning coming from the number of blocks) and we will look for zero or minimal write amplification therein.

This leads us to the general layout management problems where we fix the workload to a specific workload W. We present two variations, the first looking for the minimal

over-provisioning that gives zero write amplification and the second looking for minimal write amplification when we are given the over-provisioning amount.

1. Given a workload W and flash memory configuration of c pages per block and m number of sectors, what is the least number of blocks needed so that there is a layout management algorithm that gives us zero write amplification? Note that, the number of blocks gives the over-provisioning amount as the number of sectors m is fixed.

2. Given a workload W and flash memory configuration (including fixed c, n and m or a fixed over-provisioning), what is the layout management that will give us the minimal possible write amplification?

The next question is if the above problems are NP-complete problems or are solvable by a polynomial algorithm. This is an open problem but it simple to see that they are NP. If we define layout management as a sequence of program/erase/copyback operations, we can see that it is polynomial time verifiable. We can run the sequence of programs, erases and copybacks operations in a simulator and verify that a given number of copybacks occurred, and hence, know that certain minimal write amplification is possible.

At each write operation, the layout manager has to make a decision on which block to store the sector to be written. The decision on which block to erase is left to the garbage collector and if there are multiple blocks with the same number of valid pages, the layout manager will have to help the garbage collector choose. If the write amplification is zero, there will always be a block where all the pages are invalid pages and can be erased without any copybacks. The "hard" decisions for the layout manager are when a sector is going to be updated far into the future. Due to the limited number of blocks available, these far-future sectors have to be put together even though there will be many writes between them, it has to be managed so that there is minimal write amplification.

4.4.0.1 Estimation Algorithm

Next, we present an estimation algorithm. The algorithm is based on decomposing working into sub-workloads where all the copybacks occur only from one sub-workload and hence, an estimate of the write amplification can be achieved from the number of copybacks in the one sub-workload. The general idea of the estimation algorithm is that most copybacks are caused by data that remain valid for a long period. Hence, we can use the technique of decomposing the workload into short-lived and long-lived data to find an estimate of the minimal write amplification.

4.5 Workloads With Zero Write Amplification

Since layout management algorithm and over-provisioning itself cannot itself guarantee zero write amplification, we next look at classes of workloads that can have zero write amplification through some layout management algorithm for some over provisioning. Our estimation algorithm will be based on decomposing general workloads into sub-workloads belong to a class that produces zero write amplification.

4.5.1 Invalidation Sequence

If we know the sequence in which sectors are going to be invalidated, then we can write to a block in flash memory such that the sequence of sectors that will be invalidated all lie on the same block. In other words, a block contains sectors that are a part of the invalidation sequence. In this way, all the pages in the block become invalid one after the other. For example if we know the sequence $\{s_t, s_{t+1}, \ldots, s_{t+c-1}\}$ of sectors that will be erased from time t onwards, if we place the above sectors in a single block, they will all become invalid one after the other.

To find the invalidation sequence of a workload, we note that the invalidation sequence is essentially the workload with a few adjustments. When a sector is written, it must be

invalidated first if the sector exists in the flash memory and thus, the sequence of writes is the sequence of invalidations. Therefore, we can easily derive the invalidation sequence from the workload.

To create the invalidation sequence, we note that each write operation creates a page invalidation unless the sector is not stored in flash memory and is being written to for the first time. We take the sequence of sectors for the workload, remove all the writes that are written to sectors for the first time that do not result in a page invalidation and the remaining sequence of sectors is the invalidation sequence.

Since the workload is offline, we know the entire workload and invalidation sequence beforehand. Therefore, when the layout manager has to decide on where to place a sector in flash memory, it can look ahead in the invalidation sequence and place it accordingly. Thus, at the end of every c write operations, we would need to have a block with no valid pages for zero write amplification.

4.5.2 Invalidation Range Restricted (IRR) Workloads

We next describe a class of workloads called *invalidation range restricted (IRR)* workloads that gives zero write amplification. Let the flash memory have n blocks and so, it has nc pages. For the workload to be an IRR workload, when a sector is written it must be updated in less than $(n-1)c$ time units in the future.

Formally, let $r \in \mathbb{N}$ be a non-zero number and a workload is an IRR workload with *range* r if for each sector s, when there is a write to sector s, it must be updated within r future writes. If the IRR workload is to be stored in a flash memory with at least r pages plus an additional block, there is an optimal layout management for the algorithm that produces zero write amplification.

4.5.3 Optimal LM algorithms for IRR workloads

To find the right place to store a sector for an IRR workload, we first take the $(n-1)c$ sectors in the invalidation sequence and divide into groups of c. The flash memory has n blocks and so each group in the invalidation sequence can be mapped to a block. For each write, we find which group in the invalidation sequence the write belongs to and perform the write to that block. Because the workload is IRR, we know that the sector will be invalidated within $(n-1)c$ future invalidations and thus, we will be able to perform a block erase for garbage collection without incurring any copybacks.

4.5.3.1 Example

We illustrate the above offline optimal algorithm using a small illustrative example. Consider the following workload with $n = 4$, $m = 8$ and $c = 4$ (we have a flash memory with 4 blocks each with 4 pages where we look to store 8 sectors):

1 2 3 4	5 1 6 2	7 8 4 3	6 1 3 7
2 5 8 7	1 4 8 7	3 5 6 4	1 5 2 6

This produces the following invalidation sequence:

1 2 4 3	6 1 3 7	2 5 8 7	1 4 8 7
3 5 6 4	1 5 2 6		

Note that this is an IRR workload. Note that each sector is updated within $(n-1)c = 12$ writes in the future and we should get zero write amplification.

The first step is assigning the first 12 sectors of the invalidation sequence into blocks. Thus, block 1 should contain sectors 1,2,3,4, block 2 sectors 6,1,3,7 and block 3 sectors 2,5,8,7. When the first 8 sectors are written we have the following state of flash memory,

~~1~~	1	5	
~~2~~	6	2	
3			
4			

We next add a block assignment from the invalidation sequence. Block 4 will have sectors 1,4,8,7. Now, processing the workload of sequence 7,8,4,3 (9th-12th operations in the workload), we have the following state of flash memory,

~~1~~	1	5	4
~~2~~	6	2	
~~3~~	7	8	
4	3		

Note that block 1 has all invalid pages and thus can be garbage collected. At the same time, note that block 2 contains the sectors 1,6,7,3 which are permutation of the invalidation sequence 6,1,3,7. Thus, the next 4 operations will invalidate all the pages in block 2.

To deal with the next sequence of writes, block 1 is erased and now clean. Since all the pages in the block were invalid, no copybacks were needed and thus, no write amplification occurred. After erasing block 1, we assign the sectors 3,5,6,4 from the invalidation sequence. We get the following state of flash memory after writing the next sequence of writes 6,1,3,7

6	~~1~~	5	4
3	~~6~~	2	1
	~~7~~	8	
	~~3~~	7	

6	2	~~5~~	4
3		~~2~~	1
5		~~8~~	8
		~~7~~	7

and the state of flash memory after processing the sequence of writes 2,5,8,7.

In this simple example, we can show an optimal layout management algorithm for offline workloads that have IRR property.

4.5.3.2 Online Optimal LM Algorithm

While the above layout management algorithm does give zero write amplification, we don't need an offline algorithm to produce zero write amplification when the workload is IRR. Let us consider the simple LM algorithm where a single writeblock is used and garbage collection is done by finding the block with the least number of valid pages. After a block is written, after a certain time range, all the pages in the block become invalid. Thus, after n writes, all the pages in the first block is invalid and garbage collection can take place without any copybacks. So, this simple online layout management algorithm also is optimal for IRR workloads.

We can illustrate the above algorithm after nc writes. We have the following state of flash memory

~~1~~	5	~~7~~	6
~~2~~	~~1~~	8	1
~~3~~	~~6~~	4	3
~~4~~	2	~~3~~	7

and block 1 can be erased without any copybacks. After another block write, we have the following state

2	~~5~~	~~7~~	6
5	~~1~~	~~8~~	1
8	~~6~~	4	3
7	~~2~~	~~3~~	~~7~~

and so, block 2 can be erased without any copybacks.

4.5.3.3 Limitations of IRR Workloads

While IRR workload have a very nice property of no write amplification, it does not have sectors that are static or near static, i.e., sectors that will be invalidated very far in the

future. Suppose we have n blocks in the flash memory, then it can hold nc pages. Thus, we can only keep the space for nc future invalidation sequences. When a sector comes in that will be invalidated really far into the future beyond nc time units in the future, we have to find an alternate strategy for placing the sector in the flash memory.

4.5.3.4 Effect of Caches

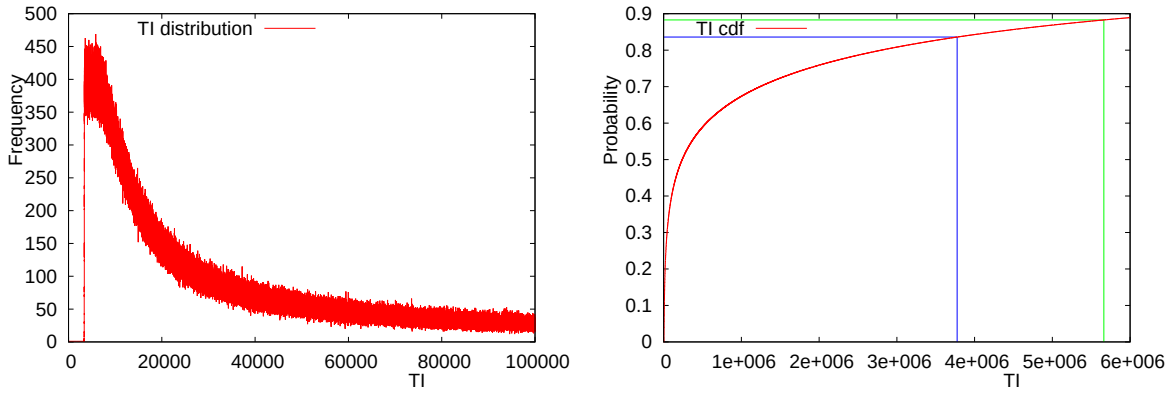
Note that when we have a cache between the actual workload and the flash memory and assuming the flash memory uses a LRU eviction algorithm, we will see that the workload will not update the same sector in the size of the window of the cache. In the example above, no sector is updated with less than 4 writes of the previous update.

The effect is that a block is filled before it comes up in the invalidation sequence. For example, before writing the 2nd group of invalidation sequence 6,1,3,7, the 2nd block already has the permutation of the sectors in the block. If the cache were larger, the number of blocks similarly filled up would match up the size of the cache. However, they cannot be erased prematurely because we are ignoring all the read operations and so, have to keep the sector data in the flash memory until the sector is invalidated.

4.5.4 Time to Invalidation (TI)

Another way to view offline workloads is not as the full sequence of writes but each write operation coming in with the exact time in the future it will be invalidated. Thus, when a write comes in, we are also given a TI (time to invalidation) that tells us when the sector for the write will be invalidated in the future. The sequence of operations can be calculated from the TIs and vice versa. We denote each write operation as (s_t, i_t) where s_t gives the sector and i_t gives the TI.

When the sector data is written to disk, we calculate the *invalidation time (IT)* which is the exact time the sector will be updated. The units of both IT and TI are the number of operations into the future.



(a) TI frequency distribution

(b) TI cumulative distribution

Figure 4.2: TI frequency and cumulative distributions for Zipf workload with $m = 3,774,873$

4.5.5 General Workloads

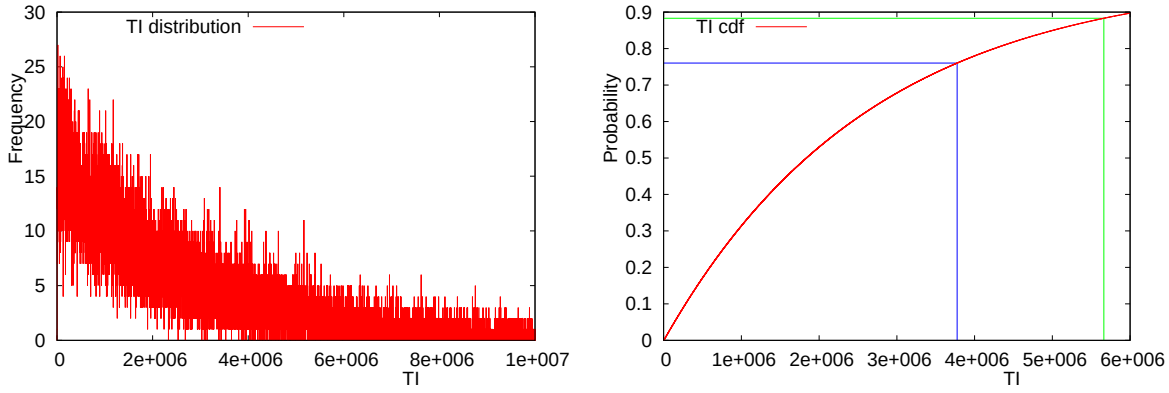
The general problem we are attempting to solve is that given a workload and flash memory parameters, we want to find the lowest write amplification possible. While we have shown an optimal algorithm that gives zero write amplification for an IRR workload, most workloads do not have this property.

While general workloads are not IRR workloads, most workloads do have a significant portion of the workload that are invalidated within a limited window of time. In essence, general workloads can be thought of as having an IRR workload inside them and we explore the process of decomposing a workload into IRR sub-workloads in Section 4.6. However, workloads differ and we first look at TI distributions of the workloads to get an idea of the properties of TIs.

4.5.5.1 Analyzing Workloads for IRR property

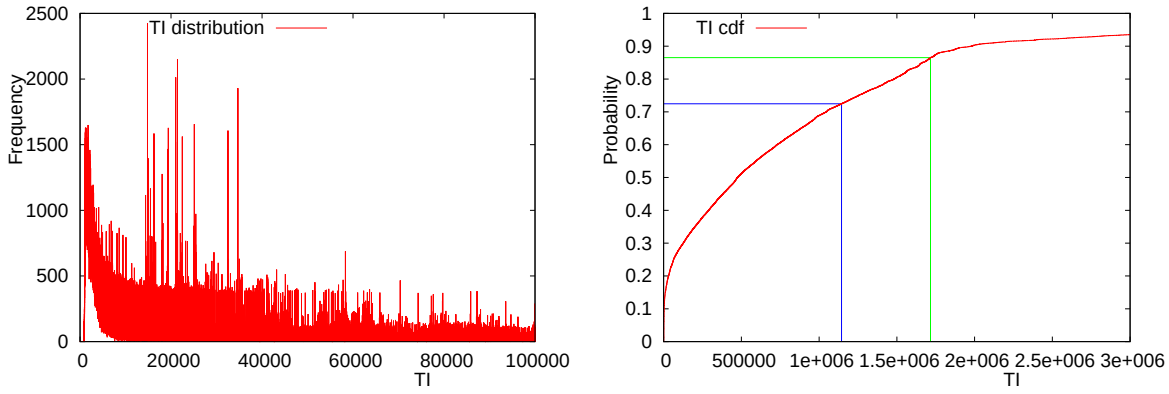
We next analyze some synthetic and trace workloads regarding the IRR property and find what portion of the write operations have TIs less than the size of the flash memory. To do this, we look at the TI distributions.

Figure 4.2a shows the TI distribution of the Zipf workload and figure 4.2b the corre-



(a) TI frequency distribution (b) TI cumulative distribution

Figure 4.3: TI frequency and cumulative distributions for uniformly random workload with $m = 3,774,873$



(a) TI frequency distribution (b) TI cumulative distribution

Figure 4.4: TI frequency and cumulative distributions for trace workload with $m = 1,144,230$

sponding cdf for $3,774,873$ sectors (using a cache that is 1% of the size required to store all the sectors). From the cdf, we can see that 83.6% of the writes in the workload have TI less than 3,774,873 and 88.3% less than 5,662,311 (corresponding to factors of 1 and 0.5 of the number of sectors). In essence, if we take the range to be the number of sectors m, over 83% of the workload is IRR. In fact, as we can see from the TI cdf, a large portion of the writes have short TIs as the cdf curve is very steep at first. This property of course comes from the very nature of the way Zipf workload is generated.

Now, figure 4.3 shows the distribution curves for uniformly random synthetic workload. Note that the graph of the frequency distribution is 10 times wider than that of the Zipf graph. On the same scale, the uniformly random workload TI distribution would look like a straight line as sectors have equal probability to be updated and so, TI frequency would look uniform. However, on the larger scale, the smaller TIs occur more frequently than the larger TIs. For the cdf of the uniformly random workload, the distribution is almost linear with a slightly steeper graph at lower TIs than at higher TIs.

Figure 4.4 shows the distribution curves for the trace workload and unlike the synthetic workloads, it is messy with large spikes in the distribution and bumps in the cdf. It has a very steep low TI cdf but then flattens out and increases linearly afterwards. Therefore, it is Zipf like first and then uniformly random later and so, a mix of both of the synthetic workloads.

In all of the workloads, we can see that a large portion of the workload has a small TI and in a sense, we can say that the large portion of the workload has the IRR property. In section 4.6, we explore decomposing the workload into sub-workloads where all the sub-workloads are IRR workloads except for one sub-workload. We can separate out the portion of the workload that has the IRR property and decompose the whole workload into separate sub-workloads. In this way, we can filter out the writes that causes the copybacks from the IRR workloads. The IRR workloads do not cause copybacks and hence, we can use this to make an estimate of the minimal number of copybacks that is

possible with a layout management algorithm.

4.6 Decomposition of Workloads

Since layout management algorithm or over-provisioning cannot guarantee zero write amplification, we next look at classes of workloads that can have zero write amplification through some layout management algorithm for some over provisioning. Our estimation algorithm will be based on decomposing general workloads into sub-workloads that belong to a class that produces zero write amplification.

4.6.1 Invalidation Range Restricted (IRR) Workloads

We next describe a class of workloads called *invalidation range restricted (IRR)* workloads that gives zero write amplification.

Definition 4.6.1. *Let $r > 0 \in \mathbb{Z}$. A workload is IRR with* range *r if for each write operation, the sector s that is written in the operation is updated within r future writes.*

If the IRR workload is to be stored in a flash memory with at least r pages plus an additional block, there is a layout management algorithm that produces zero write amplification.

For the workload to be IRR, when a sector is written it must be updated in less than $(n-1)c$ time units in the future. For example, if a flash memory has 10,000 pages in $n-1$ blocks, the workload must update each sector it writes in less than 10,000 writes from the current time. In other words, IRR workloads must have the sector written to them updated within a designated range.

Next, we give the layout management algorithm that gives zero write amplification for IRR workloads. Let us consider the simple layout management algorithm where a single writeblock is used and we do the garbage collection by finding the block with the least number of valid pages. After a block is filled and r additional writes performed, all the

pages in the block become invalid as each page must be updated in less than r time units. Thus, all the pages in the block are now invalid and garbage collection can reclaim the block without any copybacks. Hence, this layout management algorithm produces zero write amplification for IRR workloads.

4.6.2 Workload Decomposition

Since IRR workloads do not cause write amplification as we discussed in Section 4.6.1, we would like to decompose a workload into sub-workloads where all but one of the sub-workloads are IRR workloads. Suppose our workload is W and so, in notation, we would like to decompose W into sub-workloads $W_1, \ldots, W_k$ written as

$$W = W_1 \oplus W_2 \oplus \ldots \oplus W_k \tag{4.3}$$

where W_1 to W_{k-1} are IRR workloads.

Then, if γ denotes the estimate for the minimal write amplification caused by a workload, $\gamma(W_k)$ can be used as an estimation for the minimal write amplification for the workload W as $\gamma(W_i) = 0$ for $i = 1, \ldots, k-1$ since they are IRR workloads. In essence,

$$\lambda(W_k) = \lambda(W)$$

Instead of looking at write amplification, we will look at the number of copybacks which is equivalent. Then, the minimal number of copybacks caused by the workload W can be estimated by the number of copybacks in the non-IRR workload W_k.

For us to even attempt to perform a decomposition, we must first have a well-defined meaning of decomposition and the symbol $\oplus$.

First, we define workload decomposition and show that workload decompositions make sense. Suppose we want to decompose the workload as in equation 4.3 into k sub-workloads. We divide the flash memory into k separate regions $F_1, \ldots, F_k$ such that

the sub-workload W_i is written only to the region F_i in flash memory.

Since we want the sub-workload $W_1, \ldots, W_{k-1}$ to be IRR workloads, the range of these IRR workloads must be less than the size of the corresponding region $F_1, \ldots, F_{k-1}$.

The workload and the decomposition to its sub-workloads are consistent since the valid data stored as the entire workload on the flash memory as a single region or as sub-workloads in its own regions are the same. Thus, looking at the entire workload or its decomposition as sub-workloads always gives us the same set of valid data stored on disk at any point in time. What we are essentially doing by performing a decomposition is to mark where or which region in the flash memory the data will be written to as determined by which sub-workload it belongs to.

The main goal for the decomposition is that the all except one sub-workload are IRR workloads and thus, do not produce any copybacks in the IRR sub-workloads regions. We can just focus on one region F_k that produces copybacks for workload W_k to use as an estimate for the minimal number of copybacks W can produce.

4.6.2.1 Trivial Decompositions

There are trivial decompositions possible. If there are m sectors in the workload W, then it can be trivially decomposed into m sub-workloads where each sub-workload only has writes to a single sector. In addition, we can decompose W into an IRR workload by making the range of the workload the length of the workload. Finally, we can make W_k to be the entire workload with no IRR sub-workload and this would still technically be a decomposition.

However, the trivial decompositions are not very useful and we want to create a useful decomposition that we define next.

4.6.3 Useful Decompositions

A useful decomposition would be when the sum of the ranges of the IRR sub-workloads of the decomposed workload is less the size of the flash memory available and the left-over blocks is enough to store W_k.

Let $\iota(W)$ denote the range of an IRR workload (defined only if the workload is an IRR workload). Suppose we decompose W as above into k sub-workloads. Assume we have n blocks of flash memory available to us with c pages per block.

Suppose all k sub-workloads are IRR workloads after the decomposition, then

$$\sum_{i=1}^{k} \iota(W_i) \leq cn - k$$

and so, this workload results in zero write amplification. As a side note, this leads to an open problem of whether the converse of the above statement is true? If zero write amplification is possible for some workload, does it have to have a decomposition as above?

Otherwise, we would want all but the last sub-workload to be an IRR workload and

$$\sum_{i=1}^{k-1} \iota(W_i) < cn - k$$

and the last region R_k to have the remaining blocks ($cn - k - \sum_{i=1}^{k-1} \iota(W_i)$ blocks). In this way, write amplification for W can now be estimated by looking at only W_k.

4.6.4 Existence of Decompositions

The first question is why should such a decomposition even exist? While it would be convenient for such a decomposition to exist, we have not explored the fact that such a decomposition might not be possible or might only be possible under very limited circumstances.

To explore this, let us first consider a much simpler decomposition problem of decom-

posing W into just two sub-workloads, W_1 and W_2, where W_1 is an IRR workload. We take all the operations whose TI is less than some range r and put it in W_1 and the rest of the operations in W_2. After the decomposition, we should have an IRR workload W_1 with range r.

The main problem is that dividing the flash memory space might not make all the sub-workloads fit. We start with nc pages of flash memory with $nc > m$ so that all the sectors can be stored in the flash memory. If we divide it into regions R_1 and R_2, with the first of size $r + c$ pages and the second $nc - r - c$ pages, then the maximum number of sectors that region R_2 can hold at a time is $nc - r - c$. It might well be that W_2 stores more sectors at some point in time than the available pages in R_2 and so, W_2 would not fit in R_2 and a decomposition is not possible in this case. While all the sectors would fit in the entire flash memory, the decomposition divides up the flash memory into regions such that each region must have enough space to store the maximum number of sectors stored at one time by the sub-workload. The region sizes do not change during runtime and thus, the maximum sector stored at one given time of the sub-workload can occur in different times in the workload and their sum exceed the number of pages in the flash memory.

For the above problem to occur, sector writes must move from one workload to another. This is quite feasible because a sector can be updated regularly for a short while and then becomes static as it's not updated after that. In this case, the sector would first appear in W_1 and then move to W_2 afterwards.

Thus, decompositions do not always have to exist. One criteria we can use for the decomposition to exist is that the maximum number of sectors required by the non-IRR workload must be less than the size of the region R_k. However, this still leaves the question if there is another set of sizes of $R_1, \ldots, R_{k-1}$ such that the decomposition does exist. In our decompositions of workloads discussed in Section 4.8, we try different combinations of parameters and then see if the above condition is met to check if the decomposition

exists for that set of parameters.

This leads us to another open problem regarding the existence criteria for decomposition. Here, we first do a decomposition and see if there are enough blocks to hold the sectors for W_k to check if the decomposition is valid. The open problem is if there are other conditions or criterion for the workload and flash memory configurations that will tell us if a decomposition is possible or not.

4.6.4.1 Nature of Sub-Workloads

The TIs of the sub-workloads are different than they were in W because the TIs now correspond to the times of invalidation corresponding to the sub-workload rather than the entire workload. When operations are separated into sub-workloads, the TIs become smaller as operations are removed from the workload sequence and we do not leave gaps in the workload and count each write as one time step.

In our example of decomposing W into W_1 and W_2 where W_1 is an IRR workload, the operations in W_1 will have smaller TIs because operations between them have been removed. The operations in W_2 have a completely different TI distribution and we can think of W_2 as W_1 removed from W. Now, W_2 can be further decomposed into sub-workloads if desired based on the TI distribution of W_2.

4.6.4.2 Delete Operations In Sub-Workloads

Suppose a sector is written with a TI greater than the IRR range r and then later written again with TI less than r. Then, the sector will appear in both the W_1 and W_2 workloads and will shift between the workloads. Since we want W_1 and W_2 to be self-contained workloads and to not have multiple copies of the sector in the flash memory, we have to insert a delete operation whenever a sector switches between workloads. So, in our example, when the sector first appears in W_2 and then appears in W_1, at the time it moves to W_1, we insert a delete operation for the sector in W_2. In this way, the sector is

no longer stored in W_2 when it moves to W_1. Note that delete operations do not count as a time step and thus, inserting a delete operation does not affect the TI values in the workload. Thus, as a rule, whenever a sector moves from a one workload to the next, it must be accompanied by a delete operation for the sector in the previous workload.

So, even if the original workload did not have any delete operations, the sub-workloads can have delete operations if sectors move from one sub-workload to the other. This is needed to make the sub-workloads consistent with W.

4.6.5 Decomposition Algorithm

We next present the algorithm to perform the decomposition. The general idea is to mark operations as belonging to W_2 if their TIs are greater than r and update the TIs of operations remaining in W not marked for W_2. We are done when all the operations not marked for W_2 have TIs less than r. The basic algorithm is outlined in Algorithm 4.

We want to divide our flash memory $\mathcal{F}$ into k different regions such that each sub-workload W_i is written to one region. Except for the last region, none of the other regions should produce write amplification as they all have IRR workloads written on them.

The first step is to determine how to divide the n blocks into k regions. We need the lengths of the regions $R_1, \ldots, R_k$ of the flash memory so that we can start decomposing the workloads with IRRs such that the range of the IRR workload W_i is the size of the regions R_i. We decompose workloads by iteratively dividing it into two workloads at a time. We start with workload W which we then decompose it to sub-workloads W_1 and $\hat{W}_2$, W_1 being an IRR workload. Next, we decompose $\hat{W}_2$ into sub-workloads W_2 and $\hat{W}_3$ where W_2 is an IRR workload and, after $k-1$ decompositions, we have k sub-workloads where all but one of the workload are IRR workloads.

To define the entire decomposition algorithm, we just have to define decomposing a workload W into two sub-workloads W_1 and W_2 where W_1 is an IRR workload with range r. The algorithm finds the operation with the largest TI and removes it from W and

marks it as belonging to W_2. If the operation with largest TI is less than r, then we are done decomposing the workload. What is left after removing operations from W will be W_1, an IRR workload with range r. When we move operations to W_2, we have to adjust all the TIs in the operations remaining in W and insert delete operations if sectors switch between workloads as discussed in Section 4.6.4.2.

So, the algorithm can be defined as three sub-routines. The first algorithm is the DecomposeWorkload algorithm given in Algorithm 4 which is a while loop that removes operations from W to W_2. Here, `find_op_with_largest_TI` function iterates over the entire workload to find the write operation with the largest TI. The function `TI_of_op` returns the TI of the operation passed to it.

Algorithm 4 Decompose Workload $W = W_1 \oplus W_2$

```
1: procedure DecomposeWorkoad(W)
2:     o ← find_op_with_largest_TI
3:     m ← TI_of_op(o)
4:     while (m > r) do
5:         RemoveOp(o)
6:         o ← find_op_with_largest_TI
7:         m ← TI_of_op(o)
8:     InsertDels
```

Next is the removeOp function used in Algorithm 4 as shown in Algorithm 5. In removeOp, we mark an operation as belonging to W_2 but in addition, we also update all the TIs and indices of the operations left in W. We mark the operation as removed using the function `mark_as_W2` and update the TIs and indexes of the operations in W afterwards. The index of an operation in W is the order of the write operation which is needed for calculating TIs. The function `get_it` returns the invalidation time of an operation which is the exact time in the workload the operation is going to be invalidated and is calculated as the operation index added to its TI. Array I holds the index of an operation and `dec` decrements a value. We assume the TIs of the sectors are stored in the array ti and l is the length of the workload W.

Updating the TI values is necessary because as operations are removed, the index of

Algorithm 5 Remove an operation from W

```
1: procedure RemoveOp(o)
2:     mark_as_W2 (o)
3:     for i = 1 to o do
4:         it ← get_it (o)
5:         if it > I[o] then dec(ti[i])
6:     for i = o + 1 to l do
7:         dec(I[i])
```

the next sector update might decrease and TIs of all such operations must be decremented by 1. Some operations with TIs greater than r could now have their TIs go below r after the decrements and thus, remain in W_1.

The final step is to add the delete operations when sector writes move from W_1 to W_2 as discussed in Section 4.6.4.2. We execute this after the while loop in Algorithm 4 has finished and all the operations are marked either belonging to W_1 or W_2.

The final step insertDels is shown in Algorithm 6. The function `is_WL1` returns if the operation is marked to be in W_1. The functions `prev_sector_write_W1` and `prev_sector_write_W2` return if the previous sector write of the current operation was in W_1 or W_2. The simplest way to implement these functions is to create an array of m queues where each queue holds the sequence of times of sector updates for each of the m sectors. The function `add_del`(i, W_i) adds a delete operation to workload W_i of the sector in the ith operation.

Algorithm 6 Insert delete operations in W_1 and W_2

```
1: procedure InsertDels
2:     for i = 1 to l do
3:         if is_WL1(i) then
4:             if prev_sector_write_W2(i) then
5:                 add_del (i, W2)
6:         else
7:             if prev_sector_write_W1(i) then
8:                 add_del (i, W1)
```

Using the above algorithms, we can generate a workload decomposition $W = W_1 \oplus W_2$,

where W_1 is an IRR workload with range r. Both W_1 and W_2 are sub-workloads that is consistent with W. If we want to decompose into more sub-workloads, we decompose W_2 into another two workloads, one IRR workload and another non-IRR workload.

4.6.5.1 Complexity of Decomposition

The complexity of the above decomposition algorithm is $O(l^2)$ where l is the length of the workload. For each operation whose TI value is greater than r, we traverse the entire workload of length l. We expect the number of operations whose TI is greater than r to be proportional of the length of the workload and thus, the complexity of the decomposition is $O(l^2)$. We also have to find the operation with the maximum TI by traversing the entire workload but that can be achieved during the traversal for removing an operation.

4.6.6 Fast Decomposition Algorithm

The complexity of the algorithm can be reduced by removing all the operations at one loop instead of multiple loops through the workload. We next present an $O(l \log l)$ algorithm and utilizes the fact that the TI of an operation is only affected by the operations that are between the times it is valid and the time it becomes invalidated. The output of this algorithm is the same as our algorithm given in Algorithm 4.

Instead of finding the operation with the largest TI, removing it and repeating until there are no more operations with TI above the desired range, in the faster algorithm we update the TIs of all the operations in one loop and remove if above the range. The trick to making this possible is to iterate the workload in reverse. The TI of an operation can only be affected by the removal of operations that come after it and not before it and thus, if we iterate the workload in reverse, we can calculate the TI in a single loop. In algorithm 5, we updated the index I of all the operations after the deleted operation but since we are only performing a single loop, we do not need to use the index to keep track.

Figure 4.5 illustrates the operations of the algorithm. To calculate the TI of operation

Algorithm 7 $O(l \log l)$ Algorithm to Decompose Workload $W = W_1 \oplus W_2$

```
1: procedure FastDecomposeWorkoad(W)
2:     for i = l − 1 to 0 do
3:         it = i + ti[i]
4:         ti[i]− = get_Nremoved_ops(i, it)
5:         if ti[i] > range then
6:             Mark_as_W2 (i)
```

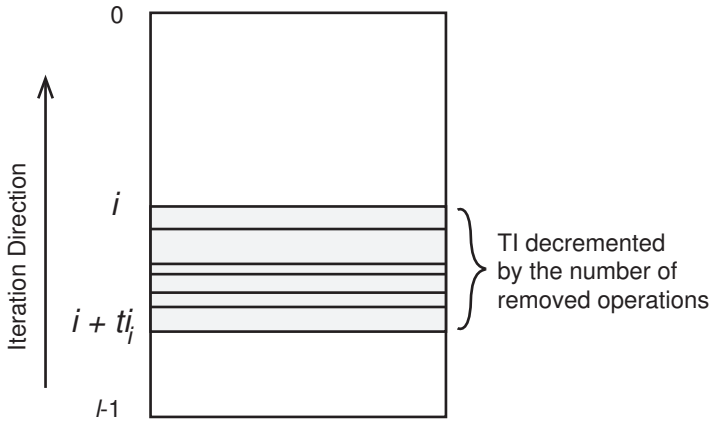


Figure 4.5: TI updates on the fast decomposition algorithm

i, we count the number of operations that have been removed to workload W_2 in the interval between i and ti_i and then, subtract it from the original ti_i for operation i in workload W. Since we are iterating from $l-1$ to 0, we already have processed the operations that come after i and know which operations have been removed to W_2. After calculating the new ti_i, we check if this is above the range. If so, then we mark it for W_2.

The function `get_Nremoved_ops` gets the number of operations that have been removed between i and it. As per our assumption, all the operations greater than i must have already have been processed by the algorithm. We keep a list of all the operations that have been removed to W_2 and this function simply searches through the list to find the number of removed operations.

The function `get_Nremoved_ops` takes $O(\log l)$ time since it searches through the list of removed operations to find the number between i and it using binary search. This function is performed l times as we iterate through the workload. Thus, the complexity

of this algorithm is $O(l \log l)$.

Note that, after performing Algorithm 7, we still need to perform Algorithm 6 to insert the deletes in W_1 and W_2.

4.7 Estimation Algorithm

Another main goal of our work is to estimate the minimal write amplification possible on a given workload on a particular flash memory using any layout management algorithm. Our estimation algorithm is based on the above workload decomposition. As we noted earlier, IRR workloads do not produce copybacks and thus, we can decompose our workload and use the number of copybacks generated by the non-IRR workload to estimate for the minimal write amplification. The layout management algorithm we will use will be the simple layout management algorithm where all writes are written to a single writeblock and the garbage collector erases the block with the least number of valid pages.

4.7.1 Optimal Decomposition

Given a workload W, there are many possible decompositions. The flash memory of n blocks can be divided into k regions in many different ways (with k also being variable). While not all the divisions would produce a valid decomposition, this is still a large number of possible decompositions. Since W_k varies for each decomposition, the number of copybacks it generates and hence the estimate for the minimal write amplification can vary between different decompositions.

Our solution to estimate the minimal write amplification is to find the decomposition that gives the lowest copybacks for W_k and use that as an estimate for the minimal write amplification for the entire workload. We call this the optimal decomposition. This optimal decomposition depends on the workload and is not fixed for all workloads.

4.7.2 Two-Way Optimal Decomposition

Finding the optimal decomposition for any k region decomposition is again a very computationally cost prohibitive operation. We consider a much simpler subset of the optimal decomposition problem where we divide W into two sub-workloads W_1 and W_2 only and estimate the minimal write amplification from W_2.

Here we take some proportion $p \in [0, 1]$ and divide the flash memory as $\lfloor np \rfloor$ and $n - \lfloor np \rfloor$. The value of p that gives the lowest number of copybacks for W_2 will be the two-way optimal decomposition.

4.8 Estimation Methods and Results

4.8.1 Workloads

We use three workloads; two synthetically generated traces (one uniformly random workload and the other a Zipf workload) and one real trace workload.

The synthetic workloads follow a uniformly random or Zipf distributiom. Given m sectors, the probability that an operation on sector i occurs is given by p_{u_i} for uniformly random and p_{z_i} for Zipf distribution where

$$p_{u_i} = \frac{1}{m} \quad p_{z_i} = \frac{\frac{1}{i^\alpha}}{\sum_{k=1}^{m} \frac{1}{k^\alpha}}$$

and for our simulation, we take $\alpha = 1$ for p_{z_i}. Uniformly random workloads have been used in many studies of flash memory and write amplification [14, 53]. A large number of real life workload distributions are Zipfian [25] or follow the power-law rule where some data are updated a lot more frequently than others and the frequency of updates follows the power law as given above.

We use the financial OLTP trace from Storage Performance Council [117]. The work-

load has a large percentage of writes suitable for our write amplification simulation and also have been previously used in flash memory studies [53, 98]. We limited the workload to at most 2^{22} sectors since we are simulating on a 128GB die. The synthetic workloads has sectors $m = 3,774,873$ and workload length $l = 41,943,040$. The trace workload has $m = 4,194,304$ and $l = 30,517,401$.

4.8.1.1 Cache

We assume that the cache is 0.1% of the size of the number of sectors in flash memory. While cache size is probably dependent on the size of the flash memory, for clarity during analysis, we kept the size of the cache constant across different over-provisioning amounts. Increasing cache sizes while increasing the number of blocks would make the analysis of the effects of increasing the number of blocks unclear.

Cache essentially removes all the operations whose TIs are less than the size of the flash memory. For all the workloads, we first pass it through a cache filter that simulates the operation of a LRU cache and outputs the resulting after-cache workload. We then generate the decomposition on this filtered workload.

4.8.2 Estimation Method

We compare the number of copybacks using a simulator from W against W_k, the non-IRR portion of the decomposed workload. The workload W gets m blocks whereas W_k gets $m - \lfloor mp \rfloor$ blocks. In both the simulations, we use a blind layout manager and the greedy garbage collector. The blind layout manager uses a single writeblock and writes all data to the single writeblock. The simple garbage collection algorithm chooses the block with the least number of valid pages.

4.8.2.1 Static Sectors

Static sectors are sectors that are written to once and then never updated again. Static sectors tend to get copied back multiple time in the blind layout management algorithm and thus, we separate out the static sectors. We pass the output of the after-cache filter through another filter that removes the static sectors. We then create a segment in the flash memory for static sectors, the size of the segment depending on the number of static sectors.

We can think of the separation of static sectors as a part of the decomposition where $W = W_s \oplus W_1 \oplus W_2$ and W_s is the workload of sequence of writes to static sectors. If the length of W_s is l_s (which is also the number of static sectors) and the size of the region dedicated to W_s is greater than or equal to l_s, then W_s is an IRR workload (since the range is longer than the length of the workload).

Thus, the algorithm we follow is that we decompose $W = W_s \oplus \hat{W}$ by filtering out the static sectors and then use the fast decomposition algorithm for $\hat{W} = W_1 \oplus W_2$ to get the IRR-workload W_1 and non-IRR workload W_2.

After passing through the cache filter and the static sectors filter, the length of the workload for Zipf was $21,932,714$, uniformly random was $41,883,529$ and trace was $18,762,727$. As we would expect, the uniformly random workload had very little filtered by the cache.

4.8.2.2 Over-Provisioning

We denote the over-provisioning by an over-provisioning factor o. If $o = 0.5$, then 0.5 proportion of the total flash memory blocks makes up the number of sectors (i.e. $onc = m$). In other words, over-provisioning ratio of 2 would be represented by $o = 0.5$ and in general, the over-provisioning ratio related to the over-provisioning factor by $\frac{1}{o}$.

For each over-provisioning factor, we have to find an optimal decomposition. We find the two-way optimal decomposition by taking a set of values for p between 0 and 1 and

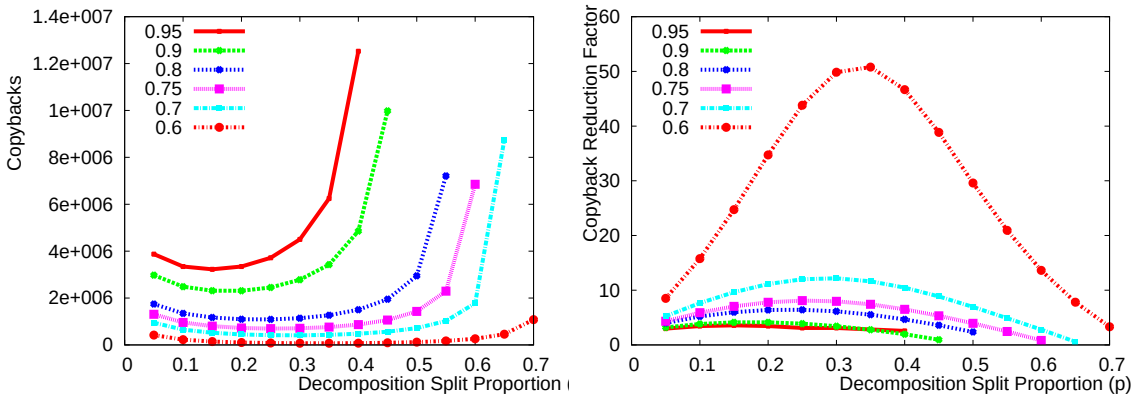
do two-way decompositions as in 4.7.2 for each point. We will use the values of p that produces the least number of copyback as an estimate of the optimal decomposition. Thus, we do decompositions for different values of o and p and find the best decomposition for each o from the different values of p for optimal two-way decompositions.

4.8.3 Results

In this section, we present and explore the results of our workload decomposition and simulation. The major results are that non-trivial workload decompositions do exist and these decompositions do result in effective reduction in copybacks. Additionally, there is a value of p that gives the optimal workload decomposition. For low values of p (indicating W_2 gets most of the flash memory), the number of copybacks reduced were low and for high values of p (indicating the IRR workload W_1 gets most of the flash memory), either the decomposition did not exist or the number of copybacks reduced were low. Optimal decompositions existed somewhere in the middle. A surprising result was that zero write amplification could be achieved for as little as over-provisioning ratio of 0.3-0.5 for the workloads we examined, a vast improvement over the worst case where the over-provisioning ratio is $\frac{1}{c} \approx 0.008$.

We look at the following:

1. Copybacks from W (m blocks using an SSD simulator).
2. Copybacks from W_k (after decomposition using simulator).
3. The ratio of copyback reduction $\left(\frac{W}{W_k}\right)$ when $W_k > 0$ which gives the rate/factor of copyback reduction. This is useful in comparing different over-provisioning amounts that yield different copybacks.
4. The value of p that gives the lowest number of copybacks as an estimate of the optimal two-way decomposition.



(a) Copybacks for various p (Zipf)

(b) Copyback Reduction Factors (Zipf)

Figure 4.6: Copyback Reduction Using Two-Way Workload Decomposition Estimation for Zipf, Uniformly Random and OLTP Trace Workloads

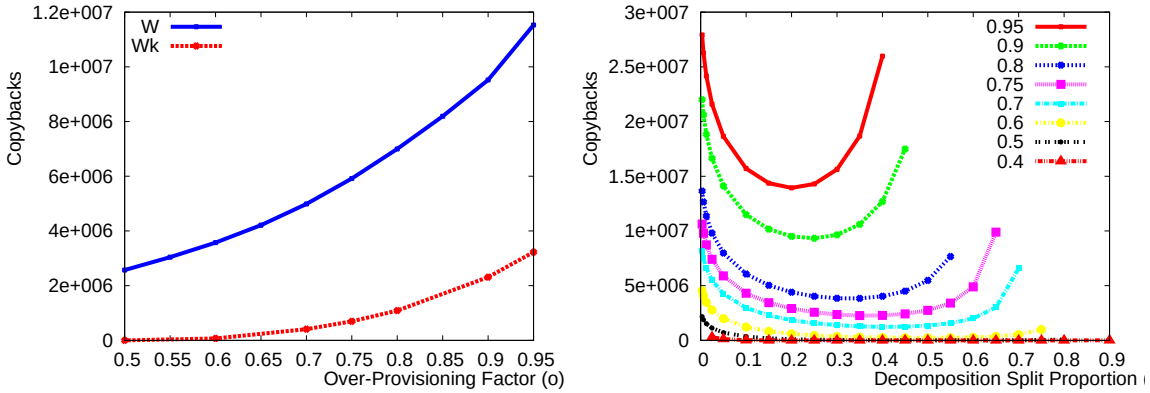
4.8.3.1 Zipf Workload

We expect Zipf workloads to have the most gains from using offline information. For the workload, $m = 3,774,873$ and thus, the cache was of size $3,775$ pages. Passing through the static filter as described in Section 4.8.2.1, we had 984,849 sectors which were static and thus, $l_s = 984,849$ and hence, $7,695$ blocks were allocated to the static region R_s.

For $o = 0.95$, the number of copybacks in W was $11,530,237$ while for $o = 0.5$, the number of copybacks was $2,570,203$ as we can see in Figure 4.7a. The number of copybacks from W_k for the optimal decompositions for $o = 0.95$ was 3,344,856 and for $o = 0.5$ was 0.

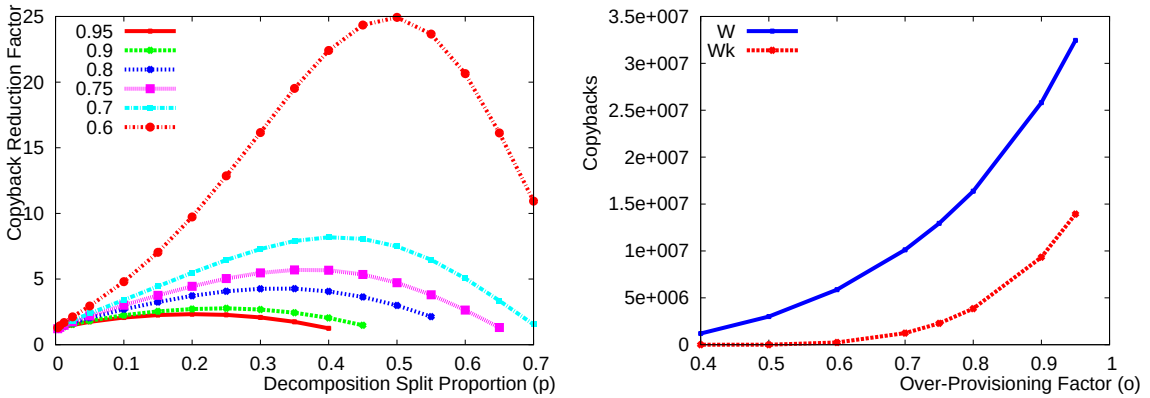
As expected, when decreasing o (i.e., providing higher over-provisioning), the number of copybacks dropped for both W and W_k, as we see in Figure 4.6a. However, the ratio at which they start to drop is higher for lower o as we can see in Figure 4.6b. Each curve in the figures represents one value of o. Figure 4.6a gives the number of copybacks for W_k with varying p. Figure 4.6b shows the ratio of the copybacks from Figure 4.6a with the copybacks from W.

In the figures, the x-axis represents the portion of the flash memory given to W_1 and W_2. The missing data points indicate that the decompositions were not valid for those



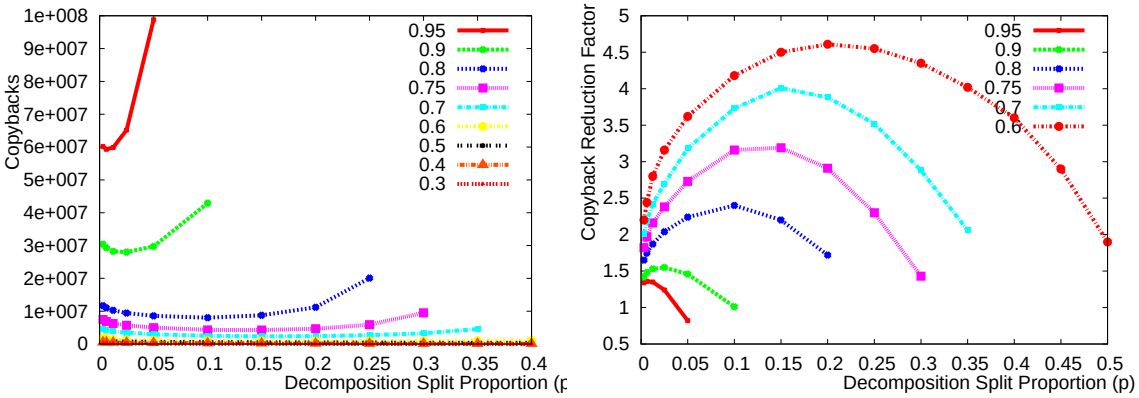
(a) Copybacks for Optimal Decomposition (Zipf) (b) Copybacks for various p (Uniformly Random)

Figure 4.7: Copyback Reduction Using Two-Way Workload Decomposition Estimation for Zipf, Uniformly Random and OLTP Trace Workloads



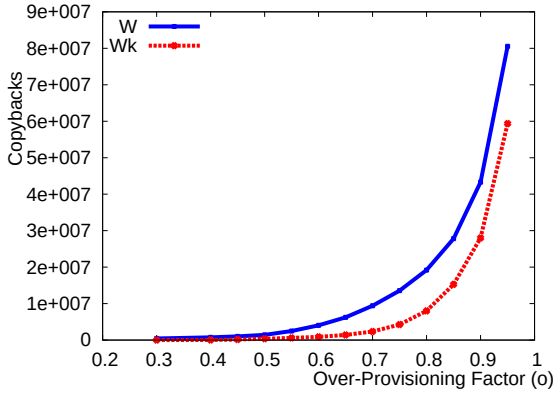
(a) Copyback Reduction Factors (Uniformly Random) (b) Copybacks for Optimal Decomposition (Uniformly Random)

Figure 4.8: Copyback Reduction Using Two-Way Workload Decomposition Estimation for Zipf, Uniformly Random and OLTP Trace Workloads



(a) Copybacks for various p (OLTP Trace) (b) Copyback Reduction Factors (OLTP Trace)

Figure 4.9: Copyback Reduction Using Two-Way Workload Decomposition Estimation for Zipf, Uniformly Random and OLTP Trace Workloads



(a) Copybacks for Optimal Decomposition (OLTP Trace)

Figure 4.10: Copyback Reduction Using Two-Way Workload Decomposition Estimation for Zipf, Uniformly Random and OLTP Trace Workloads

values as the number of sectors required for W_2 were greater than the capacity of R_2. The curves in Figure 4.6b go from $o = 0.95$ to $o = 0.6$ only because for $o \leq 0.5$, zero copybacks was achieved for W_k and the copyback reduction factor would be infinite.

From Figure 4.6a, we see that there is a unique optimal decomposition for each over-provisioning ratio o. The optimal copyback values are plotted in Figure 4.7a against the copybacks from W. These optimal decompositions were reached for different values of p for different o, and the optimal being reached with lower ps for lower os.

Overall, from Figure 4.7a we can see that large numbers of copybacks are reduced by using the workload decomposition algorithm. For $o \leq 0.5$, an over-provisioning ratio where up to half the flash memory is filled with valid data, the number of copybacks were zero and so, write amplification could be completely avoided in these cases. Note that this is far from the trivial over-provisioning value of $o = \frac{1}{c} \approx 0.008$.

4.8.3.2 Uniformly Random Workload

For uniformly random workload, the number of sectors were the same as the Zipf workload ($m = 3,774,873$) and thus, the same cache size. The number of static sectors were only 9 sectors and so only a single block needed to be allocated to R_s.

As expected, the uniformly random workload produced more copybacks as the cache filter took less of the workload out. For $o = 0.95$, there were $32,448,759$ copybacks for W and for $o = 0.4$, there were $1,194,298$ which we can see in Figure 4.8b.

Figure 4.8a shows the reduction factors in copybacks for the uniformly random workload. For high values of o (low over-provisioning), the copyback reduction is less than that of Zipf workloads. However, for $o < 0.5$, we can completely eliminate copybacks. Since all the sectors in the uniformly random workload have an average TI of m and very few static sectors, the number of copybacks fell faster than Zipf with more over-provisioning, for both W and optimal W_ks. This can be seen by the steepness of the plotted curves.

4.8.3.3 OLTP Trace Workload

The trace workload had $m = 1,144,230$ which is smaller than the one for m used for the synthetic workloads. The number of static sectors were $l_s = 99,183$ and thus, $R_s = 775$.

Figure 4.9b shows the reduction factors in copybacks for the trace workload. Here, small reductions of factor of 1.2-1.5 are possible for low over-provisioned systems and these optimal decompositions are achieved for low values of p. For $o > 0.5$, copybacks can be pushed down to zero.

The actual number of copybacks reduced in Figure 4.10a is much smaller than the synthetic workloads and the curves for W and W_k are much closer together. This is because, in the trace workload, writes to a large file creates a long string of sequential writes to sectors that are infrequently updated.

4.8.4 Analysis

The number of copybacks in the decompositions is affected by a multitude of factors:

1. the non-updateable erase block before page write architecture of flash memory,
2. the perfect knowledge of the write sequence and TIs from the offline workload,
3. errors because our algorithm is an estimation of the actual number of copybacks that can be reduced.

For low over-provisioning, the architecture of the flash memory is the major source of copybacks. As the over-provisioning is larger, our knowledge of the TIs of the workload becomes more effective in reducing copybacks. All copybacks are completely eliminated for higher over-provisioning of around $o = 0.3$ to $o = 0.4$. It is similar to keeping only half to one third of the flash memory occupied by valid data to get zero write amplification. This is much lower than the worst-case scenario of trivial over-provisioning where we had

$o = \frac{1}{128} \approx 0.008$ to get zero write amplification, i.e., only using 0.8% of the flash memory for valid data.

4.8.4.1 Limitations of the Estimation

While this decomposition method does give an idea of how much copybacks can be reduced, it is equal to or higher than the optimal minimal number. We currently do not have a method to estimate how far from the actual absolute minimal write amplification our algorithm estimate generates. Thus, if the number of copybacks is only reduced by a small amount, it could be very well due to the ineffectiveness of the algorithm rather than the real actual value of the minimal number of copybacks that is possible. The problem of finding a way to calculate how far from the optimal value our estimation algorithm generates is an open problem.

CHAPTER 5

Online Algorithms for Reducing Write Amplification and Wear Leveling

We examine the problem of repeated copybacks through multiple garbage collection cycles that creates additional write amplification. Here, we explore a method of combating this problem using multiple copyback blocks.

Data that is not updated for a long time remains valid through multiple garbage collection cycles and could be copied back hundreds or even thousands of times depending on the lifetime of the data. This is exactly the same data being read from flash memory and then being copied back somewhere else only to be read again and copied back somewhere else again. The cause of this type of wasteful increase in write amplification is when data of differing volatility, or data that has been valid for different times and is expected to be updated at different times in the future, are mixed in together in a block. When the garbage collection algorithm selects a block for erasing, all the valid data from the block is copied back to another block without looking at the characteristics of the valid data that is being copied back.

Our method assigns each data element in a page a copyback count, a simple count of the number of times a page has been copied back, and estimates data of differing volatility by its copyback count. Thus, each data can now be categorized by its copyback count, a simple measure that is an estimate of a more complex characteristic of the data like how long it has been valid for or how many write operations have occurred in the flash memory since the data was written.

We present an algorithm where we use the copyback count of the data to send valid

data of different copyback counts to different copyback blocks during the copyback phase of the garbage collection process. Thus, using these multiple copyback blocks, we are able to separate data of differing volatility into different blocks. This reduces write amplification as highly volatile data are grouped together and blocks containing more volatile data results in more invalid data together and consequently, less copyback required during garbage collection thereby reducing write amplificaton.

The main benefit of our algorithm is that it is low-overhead, simple and effective and does not require learning algorithms, complex book-keeping or a-priori knowledge about the workload. We show that our algorithm significantly decreases write amplification and improves SSD performance through simulation.

While the goals of decreasing write amplification and even wear is seen as conflicting goals, we present a simple addition to the garbage collector algorithm that solves the problem. The blocks in the garbage collector are sorted by their erase counts and the high erase count blocks are used for copyback blocks while the low erase count blocks are used for write blocks. The idea here is that data that is copied back are more static that data that has just been written.

5.1 Write Amplification

The cause of write amplification in SSDs is that in-place updates are not possible in flash memory and that the read/write granularity is different than the erase granularity in flash memory. Once flash memory pages are programmed, they cannot be updated unless they are erased, but an erase operation can only happen on a block that consists of many pages. In magnetic hard disks in place updates are possible because new data can simply be overwritten over the old data. The operating system or the storage system is completely oblivious to these flash memory characteristics and issues IO requests to update sectors and the FTL has to work in the background to transparently emulate

updates. Thus, write amplification in SSDs arises from basic flash memory characteristics and therefore an unavoidable side-effect, but with good designs and algorithms it can be minimized.

Write amplification is especially severe in devices that erase a block for every update request. For devices like USB drives that use a very simple FTL, each update to a sector is translated as follows: a read of the entire block that the data for the sector resides in minus the old sector data, an erase for the block and then a write back of the all data previously read from the block plus the updated data in the sector. This is done for simplicity as the FTL can directly translate IO requests to flash operations without maintaining any data structures. However, on top of being severely inefficient and slow, it also creates a tremendous amount of extra writes. A block consists of 32-128 pages and thus, every issued write creates 32-128 extra writes which is a massive amount of write amplification. Obviously, this is a very inefficient method and for more efficient flash memory usage, SSDs employ mapping tables.

To reduce write amplification and inefficiencies of the previous method, SSDs employ mapping tables such that sector updates are handled out of place [43]. So, instead of mapping each sector to a specific physical page in the flash memory, sectors are mapped through lookup tables in the FTL. Thus, the data in a sector could be stored in any page in the flash memory and the FTL keeps track of where the data for each sector is stored through the lookup tables. Additionally, when a sector is updated, the new data is written to a free page and the lookup tables updated so that the sector location now points to the new page. This is called out of place update and most updates just translate to a page write. Thus, using mapping tables and out of place updates improve efficiency but at the same time create further challenges and complexities like garbage collection.

Write amplification in these block-mapped FTLs arises during garbage collection. When sectors are deleted or updated, the old page that used to store the data for the sector now has out-of-date data and is marked invalid by the FTL. The idea of an invalid

page is that the contents of the page can be discarded without causing any data loss. With successive sector writes, the number of free pages in the SSD diminishes and at some point it reaches a low enough point that blocks have to be erased to free up some pages for future writes. This is done by a background garbage collection process which finds the best block to erase. However, blocks can contain pages that are valid and if we want to erase the block to free up the invalid pages in it, the valid pages must be copied somewhere else before erasing the block. These extra writes caused by copying back valid pages from blocks chosen to be erased by the garbage collection algorithm is the source of write amplification for FTLs equipped out-of-place updates.

5.1.1 Quantifying Write Amplification

To quantify and measure write amplification as described above, we can use the mathematical formulation given in [53]. Write amplification is defined as the multiplicative factor by which the user-issued writes increases to the actual system writes. Let A denote the write amplification and we have

$$A = \frac{V + I}{I} \tag{5.1}$$

where V is the number of valid pages copied back to the flash memory during garbage collection and I is the number of writes issued by the user. The total number of system writes is $V + I$ and we can rewrite the above equation as

$$V + I = AI$$

which shows that A is the multiplicative factor that gives the increase in system writes. Minimizing write amplification A entails minimizing V, the number of valid pages copied back. Thus, the above formulation is an easy way of quantifying write amplification especially for systems and simulations.

The probabilistic definition of write amplification A is a richer and analytically more useful formulation. It is also given in [53] as

$$A = \frac{c}{c - E[X]} \tag{5.2}$$

where c is the number of pages per block and X is a random variable that denotes the number of valid pages in a block prior to erasing. The above definition is similar to Equation 5.1 if we note that c is the total number of page writes per block, $E[X] = \sum_{k=0}^{c} kP(X = k)$ is the expected number of pages copied back and thus, $(c - E[X])$ is the expected number of user page writes. The usefulness of this definition is that we can look at the number of pages copied back in more detail.

For detailed analysis of write amplification, it is useful look at the distribution of the random variable X from Equation 5.2, which we call the *copyback distribution*. As we can see from Equation 5.2, write amplification is fully characterized by X as it is the only variable in Equation 5.2. The copyback distribution gives the probabilities or ratios of the number of pages copied back per block. Write amplification is just a single number while the copyback distribution gives a more detailed view of the write amplification by the shape of the copyback distribution. For our techniques we describe later, copyback distribution is an important tool for analysis of how write amplification varies.

5.1.2 Other Factors in Write Amplification

To aid garbage collection, flash chips provide a *copyback* operation. The requested valid pages are read to the internal buffer in the flash die and then programmed to a different requested block in the same die. The benefit of having a copyback operation is that once a page is read, the page is not transferred out of the die only to be immediately sent back in to be written. This frees us the channel between the controller and the flash package. Thus, copybacks provide an efficient way of performing garbage collection by internally

moving the valid pages around before erasing a block.

One of the factors of SSDs that heavily influences write amplification is the amount of *over-provisioning*. Over-provisioning is a technique where the SSD capacity is advertised to be only a fraction of the actual raw flash memory capacity. Simulation [53] and analysis [14, 119] have shown that higher over-provisioning results in lower write amplification. The reason is that over-provisioning allows for a longer time between when a block is fully written to the time it is erased during garbage collection. This longer time allows for more pages to become invalid and consequently, less valid pages being copied back thus reducing write amplification. For our designs and algorithms to reduce write amplification, we measure write amplification while varying over-provisioning to study its effects on our methods.

5.1.3 Reducing Write Amplification

Write amplification as we described above is caused by valid data being copied from one location in the flash memory to another. This happens when a block is marked to be erased and the valid pages in the block need to be copied back.

To reduce write amplification, we note that when copying back valid pages from a block that is about to be erased, the valid pages do not all have to be copied to one single block but can be copied to multiple different blocks.

In order to copy the pages of a block to different copyback blocks, we must have some criteria for differentiating between the pages. Our criterion is copyback count, the number of times the page has been copied back previously. Copyback count gives a rough estimate of how long the page has been valid for.

From the above observations, we apply them to reduce write amplification by separating out data of different volatility to different blocks by using multiple copyback blocks. Data of low volatility go to one block where they will remain untouched for a large length of time whereas data of high volatility will go to another block where they will be up-

dated and become invalid in the near future so that block can be selected for reuse by the garbage collection algorithm.

5.2 Related Techniques

A part of the reason for repeated page copybacks that causes write amplification is data that does not change for a length of time and these kind of data have been studied in flash memory previously as *static data* and *cold data*. Static data is the data in the storage system that does not change and cold data is the data that hasn't been modified for a length of time [1]. This is in contrast to *dynamic data* which is constantly being updated and *hot data* which has been recently written or updated. Static data is usually assumed to be known in advance while cold data is detected by keeping track of the time it was last updated. In our context, a block with cold and static data pages mixed with hot and dynamic data pages causes repeated page copybacks which we discuss in detail in Section 5.3.

An analysis of write amplification caused by static and dynamic data was done in [53] and an algorithm was given to reduce write amplification. In the analysis, an assumption was made that static and dynamic data is known a-priori or could be learnt before it is written to flash memory. This differs to our algorithm where we make no such assumption. Both algorithms attempt to separate the cold and static data from the hot and dynamic data to reduce write amplification but we do it in a different stages of the flash memory operation in a different way so that it is efficient while not requiring prediction or a-priori knowledge. We give more details on the difference between the algorithms after we describe our algorithm in Section 5.4.

The management of static or cold data has also been studied previously for the purpose of *wear leveling*. The need for wear leveling arises due to the fact that flash memory cells can only be erased a limited number of times before they are unable to reliably hold

data due to the breakdown of the insulation around the floating gate. Wear leveling is thus the process by which the FTL attempts to evenly wear the flash memory. In other words, the FTL attempts to keep the variance (or some other measure of dispersion) of the number of erases of each block as low as possible. The consequence of uneven wear is that parts of the flash memory get worn out more quickly than other parts which can lead to a reduced functional lifespan of the SSD. Cold and static data are important in wear leveling because blocks containing cold and static data are not selected for erasing during garbage collection as they contain a large number of valid pages. The end result is that all the wear occurs on the other blocks containing hot and dynamic data which leads to uneven wear. Thus, one of the goals of wear leveling algorithms is to manage the cold and static data so that the flash memory wears evenly throughout.

The method used by wear leveling algorithms to manage cold and static data is to periodically move such data in a block with low erase count to a block with a high erase count. One technique to achieve that is to periodically scan and find such blocks (cold and static data on a block with below average erase count) and move the data to a block with well above average erase count [1]. Efficient data structures and algorithms for this technique have been given in [21]. Another technique is to alter the garbage collection algorithm so that it chooses the block to erase not just by the number of valid pages in it but also by how cold or hot the data in the block is [43]. In either technique, data is only allowed to stay in a block for only so long before it gets moved to a different block and thus, these wear leveling algorithms actually increase write amplification and have to be managed such that the benefits of wear leveling outweigh the ill-effects of write amplification. Thus, even though wear leveling algorithms deal with cold and static data and affect write amplification, our methods of reducing write amplification is a completely different process that occurs in the copyback phase of the garbage collection with completely different goals.

The method we describe below for reducing write amplification is actually orthogonal

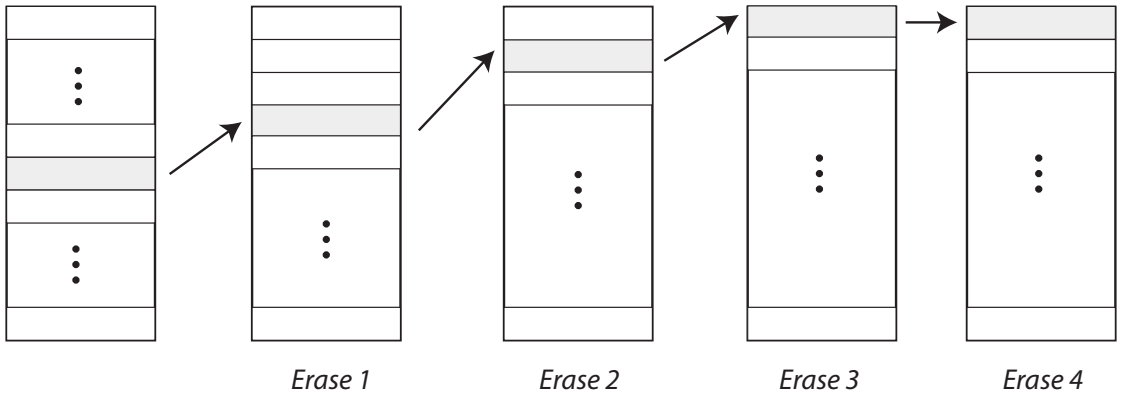


Figure 5.1: The same data in a page being copied back multiple garbage collection cycles

to wear leveling process. While wear leveling algorithms move static blocks from less worn blocks to more worn blocks, our algorithm for reducing write amplification moves pages with like data together. For better performance, both wear leveling and our algorithm can be used together which we describe in Section 5.5.3.

5.3 Repeated Page Copybacks on Multiple Garbage Collection Cycles

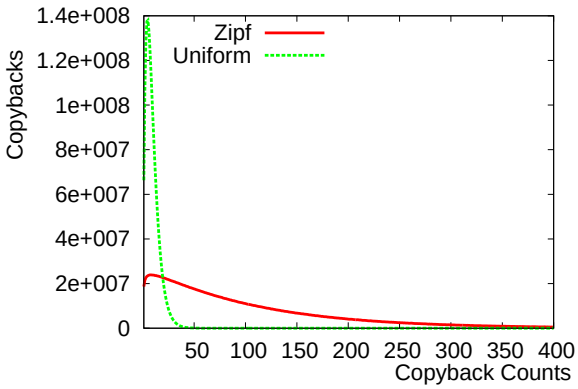
If the data in a page remains valid for a long time, it could be copied back a number of times during the time it stays valid. This is illustrated by Figure 5.1 where the same data in a page is copied back through four garbage collection cycles. This creates unnecessary write amplification as the same data is being copied around over and over again.

A *garbage collection cycle* is the process by which a block is erased and its invalid pages turned to free pages for rewrite. Garbage collection is triggered when the amount of free pages available in the flash memory goes below a certain pre-determined threshold. During garbage collection, the first step is selecting a block for erasure depends on multiple criteria but the most important one being the number of valid pages in the block. After the block is selected by the garbage collection algorithm to be erased, the valid pages in the block are copied to a clean block and then the block erased. The newly erased

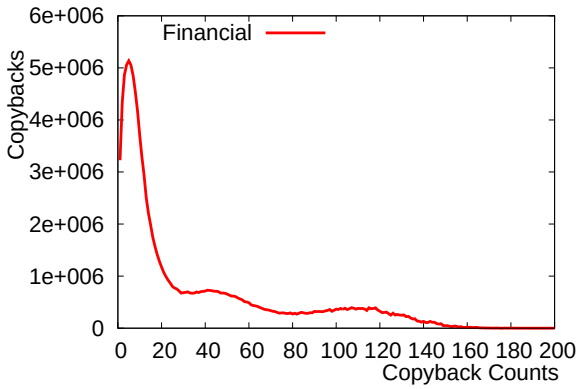
block is then written with pages copied back from other blocks and new pages from the user until it is full. It then waits for the garbage collection process to select it again for erasing. During this passage of time, valid pages in the block become invalid when the user updates or deletes the data in those pages and when the number of remaining valid pages becomes low enough it is selected again by the garbage collection algorithm for erasing. Thus, the garbage collection cycle repeats over and over to provide free pages to the user as the user consumes the free pages by issuing write requests.

The main reason why pages are copied back repeatedly is because the garbage collection process is completely oblivious to the nature of the data in the page. When the valid pages in the block are copied back before erasing, they are all copied back to the same clean block. If a page is valid during these two garbage collection cycles, then it will be copied back two times and if valid longer, copied multiple times. Thus, the garbage collection algorithm in the way it operates creates the repeated page copybacks.

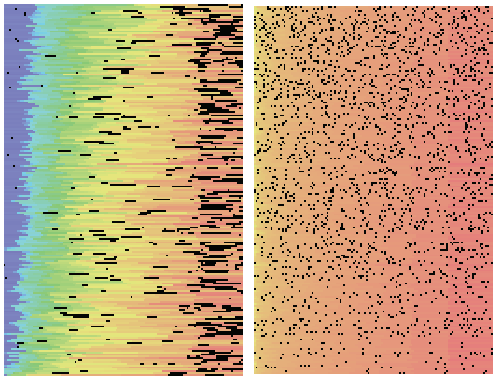
This phenomenon is clearly illustrated in Figure 5.2. Figure 5.2a and 5.2b shows the number of copybacks grouped by the copyback counts (number of times the data in the sector is copied back). In Figure 5.2a, we compare uniformly random and Zipf workloads (we describe workloads in more detail in Section 5.5.1) for a flash memory of 32768 blocks and 128 pages per block with 10% of the space used as over-provisioning, with a workload of length 4.2×10^8. As Zipf distributed workloads have sectors that are updated infrequently, some of these sectors are copied back hundreds of times while on the other hand, uniformly random workloads where all sectors are updated with equal frequency, most sectors are copied back fewer number of times, as shown by the narrower uniformly random curve versus the Zipf curve which has a long tail of hundreds of copybacks. Real trace workloads feature static data along with data that is updated infrequently and thus, also results in a long tail in the copybacks as shown in Figure 5.2b. In either case, we see that workloads with data of varying rates of being updated (which we define and quantize as data volatility in Section 5.4.3) can result in the static or rarely updated sectors being



(a) Uniform and Zipf Workloads Copybacks



(b) Trace Workloads Copybacks



(c) Heat Map for Trace Workload and Uniformly Random Workload

Figure 5.2: Impact of Multiple Page Copybacks

copied back a large number of times which increases in write amplification.

5.3.1 Heap Maps

An alternate way of looking at the problem of multiple copybacks is through the flash memory heat maps as in Figure 5.2c. In heat maps, the pages whose data that haven't been updated for some time is denoted by a square colored bluish hue while pages that have been recently updated are denoted by a reddish square. Thus, each page is given a color in the color continuum from blue to red depending on how long the data in the page has been valid for; for example, yellow/orange being more recently updated than bluish/green pages. Additionally, pages that contain invalid data are colored black. Figure 5.2c shows the heat map for the financial OLTP trace workload on the left and the heat map for a uniformly random workload on the right. Each row represents a block with each column representing a page from page 0 to the left to page 127 on the right. Additionally, the blocks are sorted such that the blocks with the oldest pages are at the top and only a random sample of blocks are shown in Figure 5.2c for size consideration. The goal of this heat map form of visualization is to observe the layout of the hot and cold data in the flash memory.

5.3.2 Sedimentation

The heat map in Figure 5.2c shows that static sectors tend to pool to the low numbered pages of the block and dynamic data to the high numbered page of the block. When writing to a block, pages must be written from page 0 onwards incrementally. During a copyback, each valid page in the block is copied to a lower or equal numbered page in the new copyback block and so static or cold pages who get copied back multiple times ends up occupying the low numbered pages. For the trace workload, the heat map shows blue on the left and for higher page numbers, a more reddish hue to the right with more black dots. The blue pages are the static pages and the greenish ones are cold pages that

haven't been updated for a while and the red/black on the right are frequently updated pages. In contrast, the heat map for the uniformly random are mostly reddish and the invalid black sectors are uniformly distributed over the map. Thus, for workloads with static or cold data, it tends to clog up in the low pages of the blocks and essentially reduce our block of capacity 128 pages to only a fraction of the pages being available to use after each copyback. This is because during garbage collection, the top part of the block where the static and cold data resides, is copied back and over again to multiple blocks on different garbage collection cycles even though the contents of those pages never changes. This is not a problem where the data is uniformly random since any page could be updated at any time but with workloads with static and cold data, it creates a large amount of write amplification. Thus, with workloads that have static and cold pages, the layout of the data in the flash memory using a single copyback block creates multiple copybacks and high write amplification. We call this phenomenon *sedimentation* and our algorithm breaks up the sedimentation.

To quantify the repeated copybacks, we express it through V, the number of pages copied back, from Equation 5.1. Let s be an index of the sector from the possible cu sectors in the SSD (i.e., $s \in \{1, \ldots, cu\}$), and u is the number of blocks available to the user. Let s_i be the different versions of the sector s that have been written to the flash memory over time and let n_{s_i} be the number of times the data in sector s_i has been copied back during the garbage collection process. Then,

$$V = \sum_{s=1}^{cu} \sum_{i=1}^{\infty} n_{s_i} \tag{5.3}$$

Thus, the above equation gives that the number of copybacks as the total number of times each version of each sector has been copied back.

Although the above equation is useful for calculating V during simulation, we will use a slightly different formulation based on *copyback counts* for analysis. Let r_j be

the copyback counts or the number of times a page is copied back j times. Thus, in mathematical notation,

$$r_j = \sum_{s=1}^{cu} \sum_{i=1}^{\infty} I_j(s_i) \tag{5.4}$$

where

$$I_j(s_i) = \begin{cases} 1 & n_{s_i} = j \\ 0 & \text{otherwise} \end{cases} \tag{5.5}$$

from which we can write V as

$$V = \sum_{j=1}^{\infty} j r_j$$

Using this formulation we can evaluate the copyback counts r_j for different j and analyze how many times blocks are being copied back.

5.4 Using Multiple Copyback Blocks

The main goal of our algorithm is to separate the data based on when we expect them to be updated, for example separate cold and static data from the hot and dynamic data. Blocks with hot and dynamic data are updated frequently and thus, these blocks tend to have a large number of invalid pages. Blocks with cold and static data will rarely updated and have a large number of valid pages. Thus, separating the data will result in lower write amplification as a block with hot and dynamic data will have less valid pages to copyback than a block with mixed static and dynamic pages. Thus, in this way we lower overall system write amplification.

We need to perform the separation of dynamic and static data as efficiently as possible and with as little overhead as possible. Because of that, we do not use learning or prediction algorithms or maintain large in-memory data structures. The only data we maintain on the page is n_{s_i}, the number of times it has been copied back, but this is stored in the page itself and not in memory, as we describe in Section 5.4.1. This makes

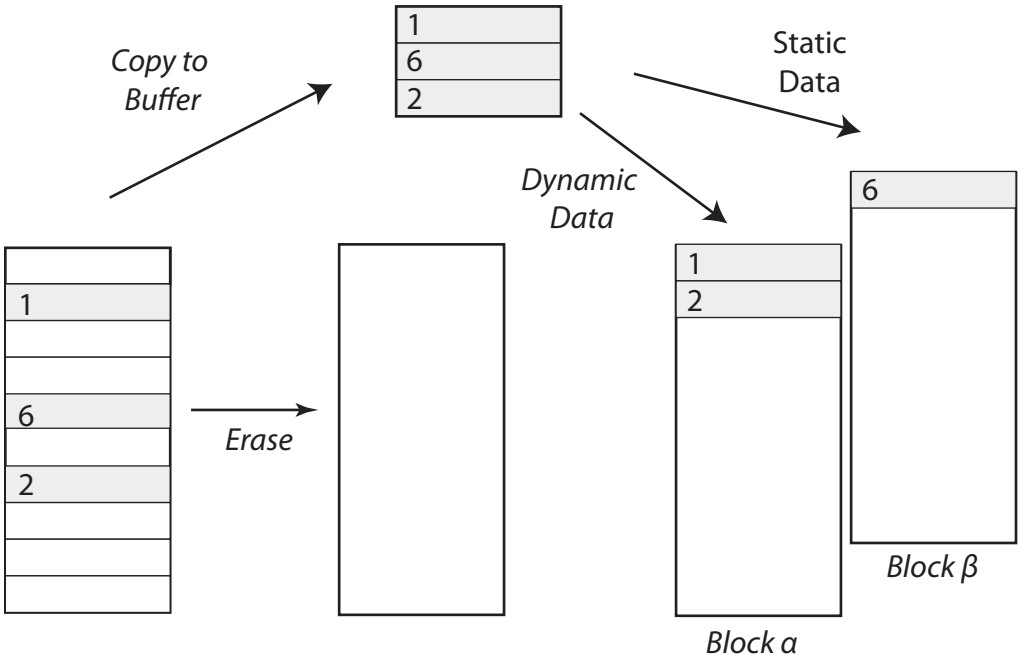


Figure 5.3: Copyback to two blocks, one with dynamic data and the other with static-like data

the algorithm very efficient for use in the FTL while providing good results for practical use which we show in Section 5.5.

In order to separate out the data, we propose the following:

1. Each page is stored with its copyback count; this is the number of times it has been copied back as a result of a block erasure (n_{s_i} from the above definition that is used in Equation 5.3 and Equation 5.5),

2. We maintain multiple copyback blocks, i.e. the valid pages of the block that is marked for erasing could be copied back to different blocks depending on the page's copyback count.

We illustrate how the algorithm works with two examples, the first using two copyback blocks and the second using a copyback block for each copyback count. These represent the ends of the spectrum of algorithm parameters.

Our first example is using two copyback blocks, block α for pages with low copyback counts and block β for storing pages with high copyback counts. After a block is selected

for erasing by the garbage collection algorithm but before a block is erased, its valid pages are copied to the buffers in the flash die and written to some other block in the same die. By our assumption above, each page maintains a copyback count and pages with copyback count below some specified threshold N_{cb} are copied to block α and those above the threshold N_{cb} are be copied to block β. This is illustrated in Figure 5.3 where $N_{cb} = 6$, the copyback counts given in the left of the rectangle representing the page and pages with copyback count 6 or higher is copied to a different block than the the pages with the copyback counts less than 6. In this way, dynamic data is moved to block α and static data to block β, separating the dynamic and static data.

The number of copyback blocks is not limited to 2 and can be as many as required to handle different workloads. Then, each copyback block will have its own threshold parameter to determine which pages are copied to which block. The parameters of the number of copyback blocks and their impacts will be analyzed in Section 5.5.

In our second example, we use a copyback block for each copyback count i.e., during copyback the page's copyback count is checked and sent to a different block for each possible copyback count. Let us assume we have the copyback blocks $\beta_1, \beta_2, \ldots, \beta_k, \beta_\infty$ where k is some large number of copybacks possible. We copy a page of copyback count i to block β_i for $1 \leq i \leq k$ and if $i > k$ then we copyback to block β_∞. Though not a copyback block, we denote block β_0 as the current writing block where new page writes and page updates are written to. This example is illustrated in Figure 5.4. Thus, each of the copyback block stores pages of equal copyback counts.

The result of the above setup for the algorithm is that during copyback pages migrate from a block that used to be β_i to β_{i+1}. New writes go to block β_0 and then a copyback from those block will go to β_1, and when the pages of the block that was β_1 is copied back, it goes to β_2 and so on.

From the above two examples of our algorithm with different parameters, we illustrate how we can separate data depending on its copyback counts. This separation of data

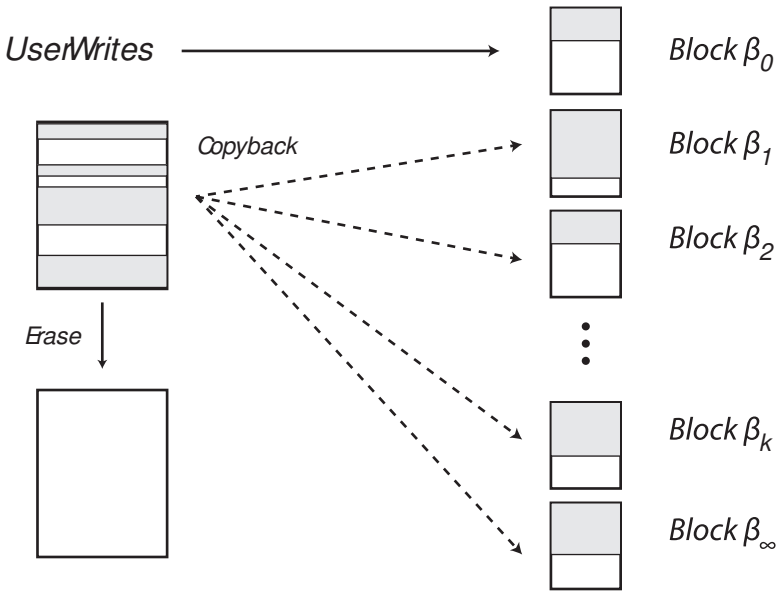


Figure 5.4: Copyback block for each Copyback Count

leads to reduction in write amplification which we will show in Section 5.5.

5.4.1 Utilizing Existing Flash Memory Hardware Features

To be able to maintain copyback counts and copyback to multiple blocks, we need hardware support from the flash memory die. The hardware features that we utilize are the spare area of a page and the copyback operations in flash dies, which are supported by most flash dies [110, 95].

Each page of flash memory contains 128-256 bytes of additional area called the *spare area* which is used to store parity data for error correcting and other information used by the FTL. To keep track of the copyback counts, we can allocate 4-8 bits in the spare area. Whenever the page is copied back, the count in the spare area is read from the flash memory and incremented in the buffer and then the incremented value written back. Thus, we can keep track of the copyback counts of a page by using the spare area in the page.

The copyback operation in a flash die is a two step process, a copyback read command first and then a copyback program command [95]. The copyback read command is func-

tionally identical to the read operation where the requested data in the given address is copied to the flash die cache registers. The copyback program is also functionally identical to the program operation where data in the flash memory cache registers is moved to a given address in the flash address. The difference between the set of copyback commands and the read/program command set is that data is not expected to be moved in or out of the flash die for copyback commands. Since each copyback command is given its own addresses, moving pages in a block to multiple different blocks is possible by giving different block addresses to the copyback program command. Thus, supporting multiple copyback blocks for different copyback counts is possible using the primitive copyback commands available in flash memory dies.

5.4.2 Data Models for Selecting Parameters

To reduce write amplification, our algorithm takes advantage of the nature of the workload and we would like to quantify this property of the workload that our algorithm utilizes. The primary assumption we make about the data is that the probability that it will be updated in the very near future is determined by how recently the data was updated. For example with static and dynamic data, if the data hasn't been updated for a while, the probability that it will be updated in the very near future is smaller than if it was recently updated. Thus, when we move the pages with higher copyback counts or lower write-stamps to a different block than the ones with lower copyback counts or higher write-stamps, we expect the pages with higher copyback counts to be updated less often than the other blocks. Then, we expect write amplification to go down since the block with low copyback count pages end up with more invalid pages. Thus, copyback counts or timestamps act as a predictor for if in the near future the page will be updated and deleted which is the basis from which our algorithm reduces write amplification.

In order to select the right parameters, we need to quality the above property of the workload data. We use data volatility which we define in Section 5.4.3. In order to make

our calculations and simulation easier, we utilize the reductions possible in the workload which we discuss in Section 5.4.2.1.

5.4.2.1 Workload Structural Reductions

When analyzing write amplification, there are two workload reductions possible that simplify the workload for our simulation but do not take anything away from our analysis of write amplification. In essence, we are able to remove some structure from the workloads that are orthogonal to the write amplification process.

Firstly a very useful assumption that can be made is regarding the timings of the workload. Write amplification is only influenced by the sequence of write and delete operations and not by the times that these operations were issued by the user. Thus, the write amplification is the same if a sequence of operations occurred in a burst of operations or if the operations were issued over a long period of time. Using this fact, we can simplify the workload such that at each interval of time there is exactly one operation issued by the user or that the arrival time of the operations in the flash memory is constant. In real workloads, the distribution of the arrival time of operations varies but the write amplification is independent of the arrival times and thus when analyzing the workloads for our algorithm, we simply take the simplest arrival time, the constant arrival time. By making this assumption about workloads, we can simplify a large part of our analysis.

Additionally, we can remove all read operations from the workload since only write and delete operations affect write amplification.

While we simplify arrival times in the workload for our write amplification analysis, the arrival times of workloads are very important in modeling and simulation of performance of flash memory, performance of the garbage collection mechanism and the latency and throughput of the system. For such analysis, arrival times need to be considered.

5.4.3 Data Volatility

We next give a quantitative formulation of the property of the workload that is utilized by our algorithm called data volatility. We define *data volatility* as the probability that given a page has been valid for some given time period, it will become invalid in the next time period. For example, static data has low volatility since there is low probability it will be invalid in the next time period and dynamic data has high volatility since it is quite likely that it will be updated quickly. Thus, data volatility is the measure of the tendency of a page to become invalid in the near future.

However, when data first comes in, we don't know if the data is static or dynamic and we have to estimate the volatility. In our case, our estimate of volatility is based on how long the data has been valid for. We get this estimate of how long a page has been valid for coarsely from its copyback count or more fine grained using its writestamp, a form of a timestamp which we describe below.

Using write-stamps and copyback counts are indirect ways of measuring how long the page has been valid for. The timestamp of the page would be a more accurate method of determining how long the data in a page has been valid for but we use copyback counts because it is simple and efficient to work with while also being effective. One of the major problems of using timestamps is that it has to be normalized against the number of operations that has occurred in the time frame as operations can come in bursts. Thus, the normalization would be different for each page and would require a lot of overhead to keep track of. The write-stamp is the count of the number of write operations that have occurred. Using write-stamps is another estimate for the length of time a page has been valid for and is a more fine grained method than copyback counts. The drawbacks with using write-stamps is that selecting a copyback block based on this measure requires a more complex copyback rules and the write-stamp itself requires a larger number of bits to store in the spare area than the copyback count. Thus, the algorithm uses the

write-stamps and copyback counts as quick and efficient ways to get a rough estimate of how long the page has been valid for and we use data volatility to create the rules to select the copyback blocks.

We discuss the usage of copyback counts and write-stamps in more detail in Section 5.4.4.

We next give the equations for calculating data volatility. As defined before in Section 5.3, let s_i be the ith version of sector s. Let $Z_\gamma^{s_i}$ the random variable that gives if the sector s_i is valid(v) or invalid(i) after time γ in the flash memory. Then, the data volatility $\upsilon_\gamma^{s_i}$ is the probability that a block that has been valid for γ steps will be invalid at $\gamma+1$ time step and is given by

$$\upsilon_\gamma^{s_i} = P(Z_{\gamma+1}^{s_i} = i \mid Z_\gamma^{s_i} = v)$$

and we can write the above as

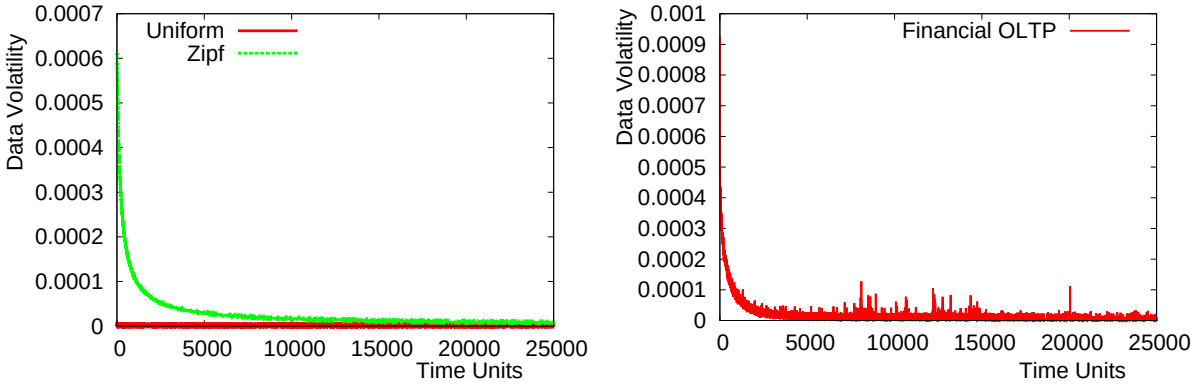
$$\upsilon_\gamma^{s_i} = \frac{P(Z_{\gamma+1}^{s_i} = i \ \cap \ Z_\gamma^{s_i} = v)}{P(Z_\gamma^{s_i} = v)}$$

A convenient method to calculate data volatility for generated or trace workloads is using the following formulation. Let V be the random variable for the length of time a page is valid for. Then, $P(V = \gamma)$ gives the probability that a page will be valid for exactly time γ or equivalently, the probability that if a page has been valid for time γ, it will become invalid in the next time interval. Thus, we can write data volatility as

$$\upsilon_\gamma^{s_i} = \frac{P(V = \gamma)}{P(V \geq \gamma)}$$

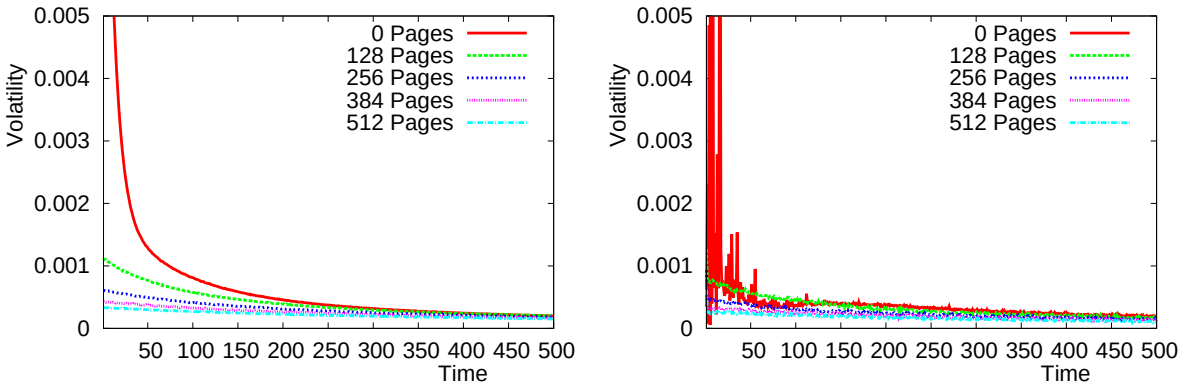
In this form, just keeping track of how long sectors are valid for in the workload is sufficient to calculate the data volatility and thus, is a convenient formulation.

The data volatility of various workloads is given in Figure 5.5 (we give the definitions of the workloads in Section 5.5.1) where the x-axis gives how long the sector has been



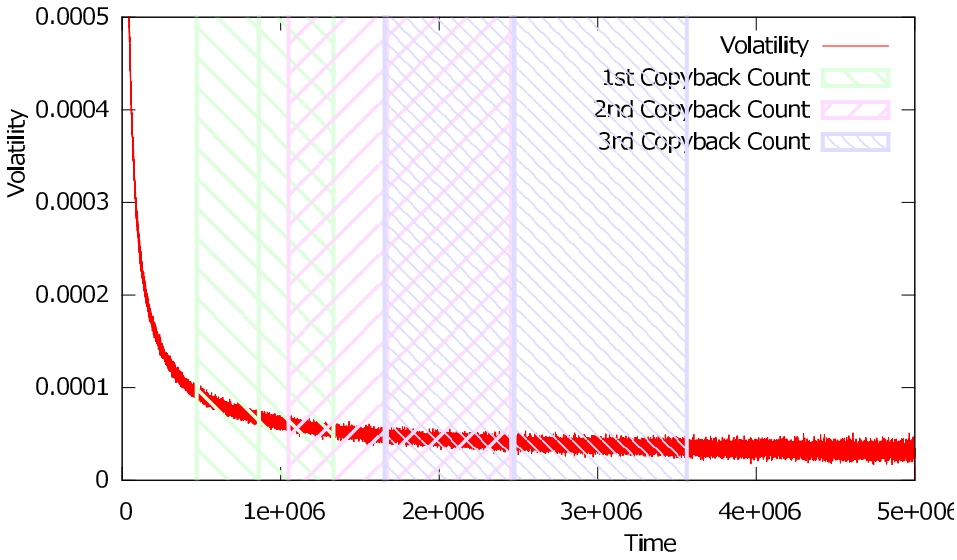
(a) Uniform and Zipf Workloads (b) Trace Workloads

Figure 5.5: Data Volatility of Various Workloads

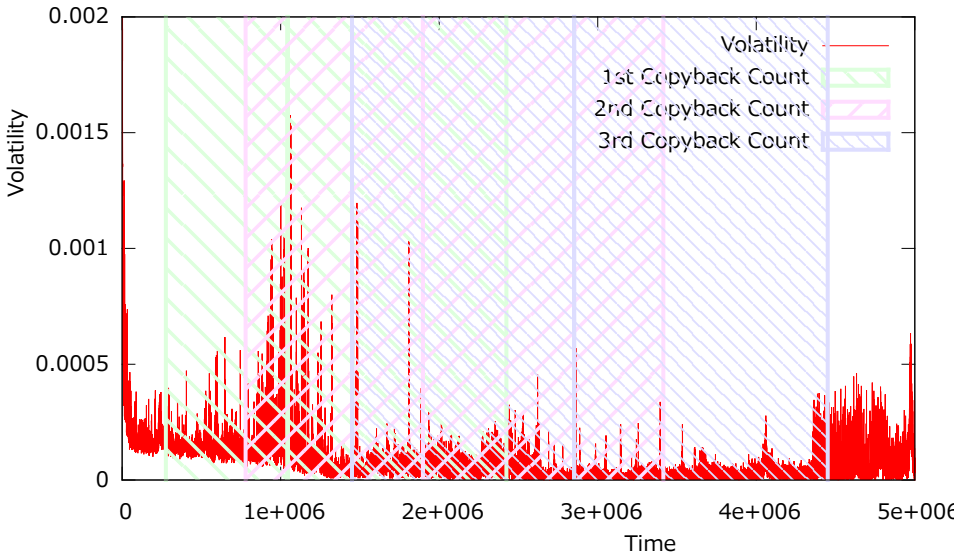


(a) Cache size and Volatility for Zipf (b) Cache size and Volatility for Trace

Figure 5.6: Data Volatility with different Cache Sizes



(a) Copyback Counts and Write-Stamps



(b) Copyback Counts and Write-Stamps for Trace

Figure 5.7: Data Volatility of Various Workloads

valid for (in time units of intervals based on the total number of operations) and the y-axis gives the data volatility. Figure 5.5a gives the volatility for uniformly random and Zipf workloads. Note that the data volatility of the uniformly random workload is constant throughout while the Zipf distribution has high volatility for the sectors that are just written which goes down dramatically the longer the sector stays valid. The volatility of the uniform curve is very low and close to zero because it is constant throughout time and thus, evenly distributed over the duration of the simulation. Figure 5.5b gives the data volatility of real system traces and we can see that they are close to that of the Zipf distribution.

Volatility depends on the size of the cache used in the system. From simulations results in Figure 5.6a and 5.6b, the volatility decreases with larger cache size but with diminishing decreases in volatility with increasing cache size. The simulations in Figure 5.6a were done with workloads of 9932 blocks with 128 blocks per page and cache size of $0, 128, 256, \ldots$ pages. Figure 5.6b uses the Financial OLTP trace to generate the volatilities with varying cache size and a small cache size irons out the high volatility during initial times after writes. Thus, as larger cache sizes provide diminishing volatility decreases, we run our simulations with cache size of 256 pages (2 blocks) without cache size being a major variable.

Volatility can also be calculated for intervals of time rather than fixed point in time and we see the volatility of the synthetic and trace workloads in Figure 5.7a and 5.7b calculated with intervals of 100 time units.

5.4.4 Parameter Selection

Workloads are different depending on the usage patterns of the SSD and we can adjust the parameters of our algorithm to adapt under the varying conditions to keep reducing write amplification. If the workload is uniformly random, then the probability that a page is updated in the near future is the same at any time for any page and in such a

case, write amplification cannot be reduced and we would only use one copyback block. On the other hand, workloads with static data that is never updated or is Zipfian where some pages are rarely updated while others are updated more frequently have a variety of properties that interact with our algorithm in different ways and we should be able to adjust the parameters to adapt to these workloads.

During a copyback operation, when we have to choose the location to place a page that has been copied-back, we have its copyback count or its write-stamp to determine the best location. However, we can only use a write-stamp after a copyback has occurred and so thus the write-stamp that we encounter after a copyback is closely related to the copyback count. Figure 5.7a and 5.7b shows the relationship between copyback counts and write-stamps. Each rectangular block shows the write-stamps when the page is copied-back with the left edge the 5th percentile and the right edge the 95th percentile. The total rectangular area shows the write-stamps for the copyback-count in 90% of the time. As we can see from the figures, the range of write-stamps for each copyback count increases and there is certain overlap between the write-stamps of the different copyback counts. Thus, we can use write-stamps for a finer grain of control and we describe the methods to select the parameters for the copyback blocks next.

First, let us consider the parameter selection when using copyback counts for selecting where to copyback pages. If there was a copyback block for each copyback count then the algorithm would perform the best in write amplification reduction (assuming that the number of blocks in the flash memory is large). The reason for this is because the goal of the algorithm is to group data of similar volatility together and the smallest grouping we can do is through a single copyback count (when using copyback counts). On the other hand, the cost of using a large number of multiple copyback blocks is that as more blocks are used for copybacks, the over-provisioning slightly goes down as fewer blocks are used to store data which can increase write amplification. Thus, having a block for each copyback count gives the most reduction in write amplification if we have a large number

of blocks available for copyback blocks but a large number of copyback blocks can result in lower over-provisioning which leads to higher write-amplification.

The method to do parameter selection is through the data volatility distributions we discussed in Section 5.4.3. When the volatility distribution is flat we can group the copyback counts together, the extreme being uniformly random where we can group all the copyback counts together to one. This is because when the volatility curve is flat for a time interval, there is no difference in volatility of the pages that have valid until that time region and thus, have about equal probability of getting updated in the near future. However, the correspondence between the copyback counts and the different regions of the data volatility curve depend on the parameters of the SSD like the over-provisioning and the garbage collection algorithm. Thus, the easiest way to do parameter selection is to use first copyback block for each copyback count and merge copyback blocks if the volatility of the data in them are the same. For example, in the volatility curve overlaid with where the copyback counts in Figure 5.7a, the curve is flat after the first two copyback counts and thus, all the remaining copyback counts can be grouped into one to use only a single copyback block in the region where the volatility curve is flat. Thus, the method for parameter selection for copyback counts is to group the copyback counts together where the data volatility curve is flat.

5.5 Simulation and Results

We use our simulator to evaluate our layout management methods for decreasing write amplification. The simulator is an event driven simulator that emulates the various modules of the FTL. For instance, the IO controller translates IO requests to flash memory operations, the mapping table module handles the logical to physical maps, the layout manager determines where to place each new block and invokes the garbage collector module when the free pages are needed and the state manager keeps track of the state

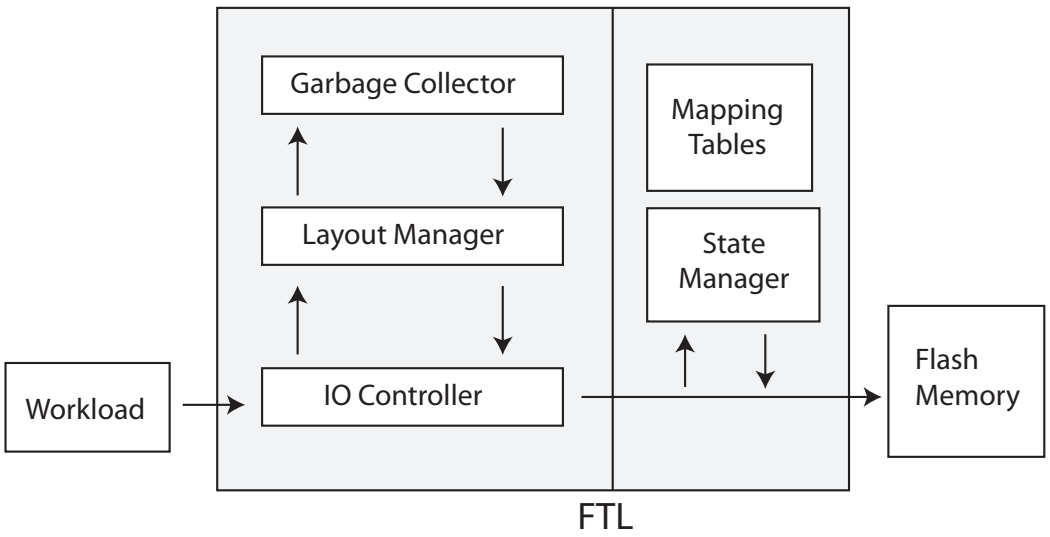


Figure 5.8: Flash Memory SSD Architecture

of the data in the flash memory. Workloads are either simulated or from a trace but we do apply the workload structural reductions as described in Section 5.4.2.1. Thus, using these emulated modules that make up the simulated FTL, we perform an analysis of the effectiveness of our analysis and algorithm for reducing write amplification.

The simulator works in the following chain of events. At each time interval, a workload event is processed. If the operation is delete, the sector is marked invalid in the mapping table. If the operation is write, then the old version of the sector is marked invalid if it resides on the flash memory and the layout manager is asked for the location to write the new page. Here, the layout manager uses one of our algorithms to determine where to write the page. When the free pages in the flash memory runs out, the layout manager request the garbage collector to erase blocks to acquire free pages to write to. When the garbage collector is copying back pages, it also requests the layout manager where to locate the valid pages in the block to be copied-back. Here, our copyback layout management algorithms are used. The modules and its relationships in the simulator are illustrated in Figure 5.8. In this way, the layout manager plays a central role in our simulator and we can effectively test our algorithms against write amplification from the simulation results.

Since copyback operations are restricted to copying back data inside a flash die, we

performed our simulation assuming a single flash die. A 128Gb die [95] has 32,768 blocks and each block has 128 pages and serves as our primary model. When comparing the results of synthetic workloads to trace workloads, we use smaller number of blocks for the synthetic workloads so that the number of blocks used in the synthetic workload simulation matches the ones that are present in the trace workload.

5.5.1 Workloads

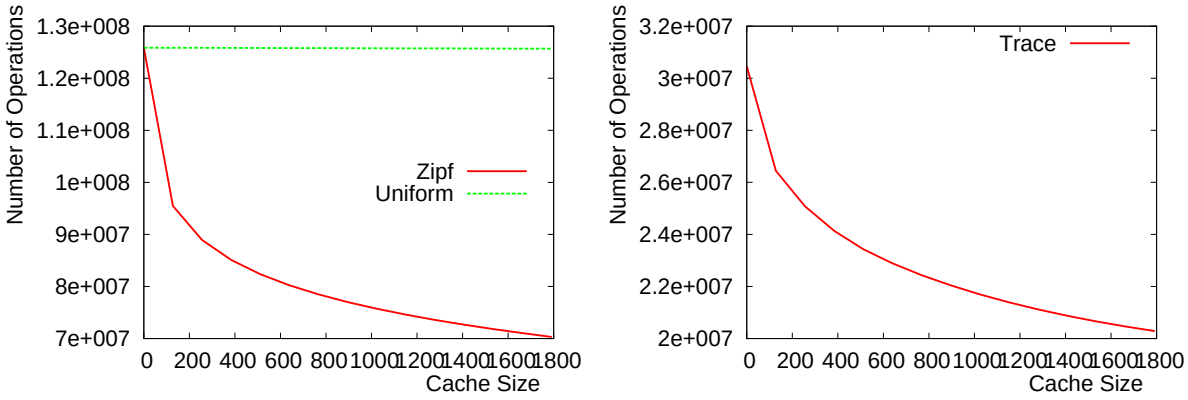
For our simulation, we use both synthetically generated workloads and real world system traces.

For synthetic workloads, we use workloads as described in [112] which has a write as well as a delete (TRIM) operation where the workloads follow either a uniformly random or Zipf distribution. In other words, given n sectors, the probability that an operation on sector i occurs is given by p_{u_i} for uniformly random and p_{z_i} for Zipf distribution where

$$p_{u_i} = \frac{1}{n} \quad p_{z_i} = \frac{\frac{1}{i^{\alpha}}}{\sum_{k=1}^{m} \frac{1}{k^{\alpha}}}$$

and for our simulation, we take $\alpha = 1$ for p_{z_i}. Uniformly random workloads have been used in many studies of flash memory and write amplification [14, 53]. A large number of real life workload distributions are Zipfian [25] or follow the power-law rule where some data are updated a lot more frequently than others and the frequency of updates follows the power law as given above. Real system traces have similar volatility curves to Zipf distributions, which we discussed in Section 5.4.3, and thus synthetic Zipf workloads are an important predictor for real system performance.

For workload traces from real systems, we use the financial OLTP traces from Storage Performance Council [117] and the MSN Storage metadata server [114]. These workloads have a large percentage of writes suitable for our write amplification simulation and also have been previously used in flash memory studies [53, 98]. We limited the workload to



(a) No. of Operations for Synthetic Workloads

(b) No. of Operations for Trace Workload

Figure 5.9: No. of Operations and Copybacks after Cache

at most 2^{22} sectors since we are simulating on a 128Gb die.

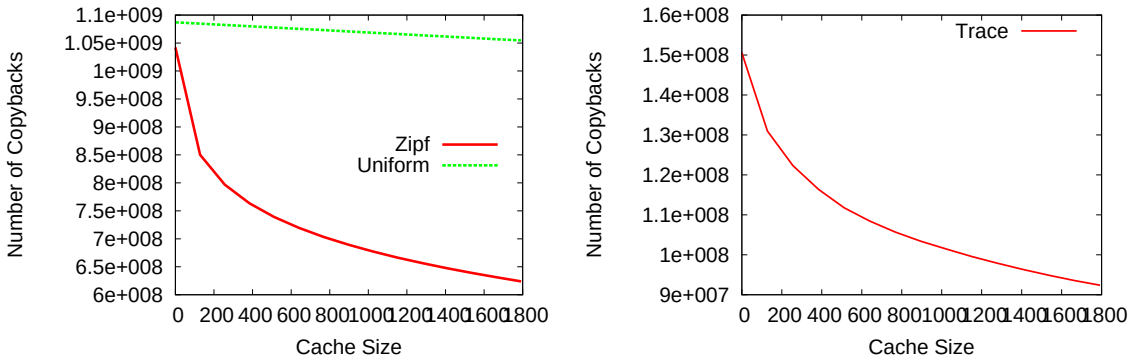
5.5.1.1 Cache Size

Before we can get relevant and meaningful results about write amplification, we must first determine the cache size we are going to use.

We use a simple LRU cache where each cache entry is a page. The cache can be thought of as a filter on the workload where updates of sectors that occur in a short time are removed. Updates that occur in less than the size of the cache is removed but some updates that are slightly larger can also be removed. To first look at the effect of cache, we look at how many operations are removed from the workload for various cache sizes.

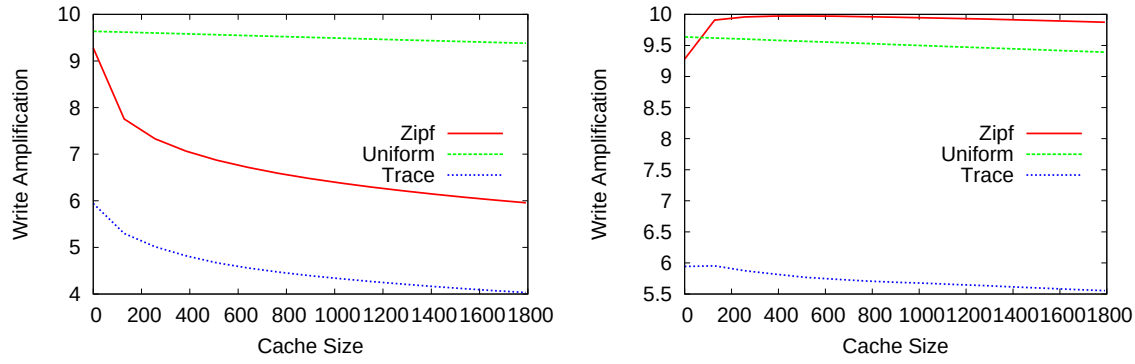
As we discussed in Section 5.4.3, as we increase the cache size, the volatility decreases and consequently the write amplification also decreases but does so at diminishing levels. Therefore, we must choose a cache size large enough that increasing the cache size will have very little effect on the write amplification but small enough so that the cache size is only a tiny fraction of the flash memory.

Figures 5.9a and 5.9b shows the number of operations that are not removed by the cache for different cache sizes (the cache size zero representing the original number of operations). For uniformly random workload, there is hardly any operations removed



(a) Copybacks for Synthetic Workloads (b) Copybacks for Trace Workload

Figure 5.10: Number of Copybacks vs Cache Size



(a) System Write Amplifications (b) SSD Write Amplifications

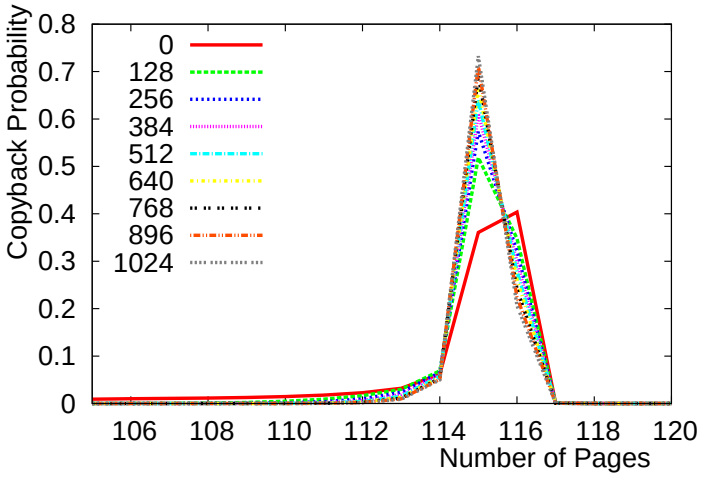
Figure 5.11: Cache Size and Write Amplification

by the cache. This is expected because each sector is updated randomly with equal probability and there are a few cases when the same sector is updated in a short while. For Zipf workload, some sectors are updated more frequently than others and the trace workload has locality as well. In these workloads, we get large decreases for small cache sizes. When we increase the cache size, the operations allowed through are still decreasing but at a lower rate.

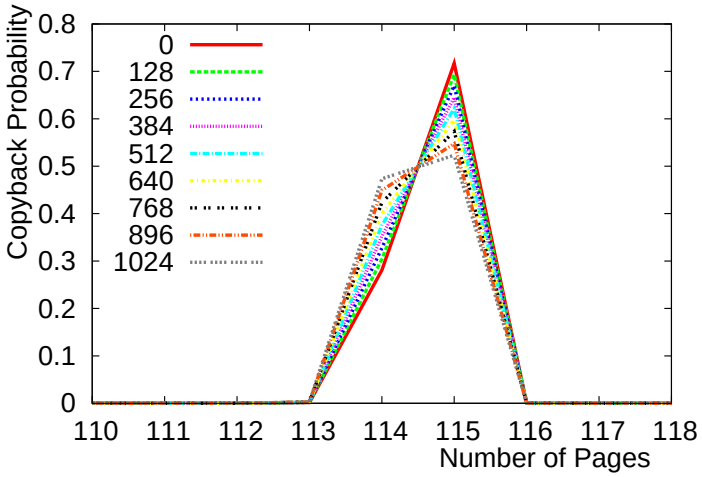
Figures 5.10a and 5.10b show the number of copybacks in the simulation from the cache filtered workloads. The Zipf and trace workloads show a similar trend to the number of operations. For uniformly random workload, we can see the number of copybacks linearly decreasing with cache size even though the number of operations did not show that trend. We can infer that removing short updates has some effect on the number of copybacks it results in the system.

Figure 5.11a shows the system write amplification (the amount of write amplification compared to the number of operations before the cache). As with the number of copybacks, it decreases linearly with the size of the cache for uniform workloads. For Zipf and trace workload, it decreases rapidly at first and then decreases slower with larger cache sizes. Another interesting view of write amplification is to view it after the cache as in Figure 5.11b where we measure the write amplification from the operations that have reached the SSD after the cache. By removing the short updates, the cache changes the nature of the workload that reaches the SSD. The write amplification increases initially for small cache sizes for Zipf and trace workloads showing that very small updates actually increase write amplification for the layout management strategy we are using. For larger cache sizes, write amplification decreases.

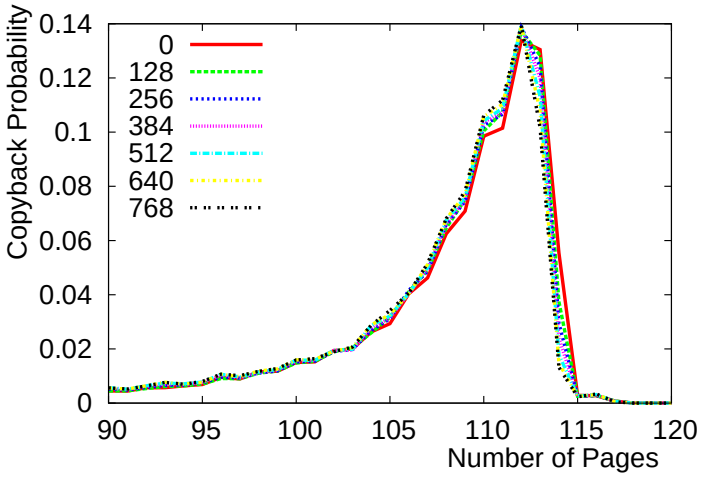
To determine the appropriate cache size to use, we have to balance the cost and benefit of larger cache sizes. Larger cache sizes offer lower write amplification but at falling rates. A general rule of thumb is to use 0.1% size of the flash memory as the cache size that comes out to around 1280 pages for our simulation setup. At this size, the decreases in



(a) Zipf Distribution



(b) Uniformly Random Distribution



(c) Trace Workload

Figure 5.12: Copyback Distributions with Different Cache Sizes

write amplification aren't as steep for larger sizes and so, we use this as our cache size.

Another interesting view is how the copyback distribution changes with cache sizes. Figure 5.12 shows the different copyback distributions of Zipf, uniformly random and trace workloads for different cache sizes. With larger cache sizes and lower write amplification, the curve moves to the left.

5.5.1.2 Convergence of Write Amplification and Copyback Distribution

While we will use write amplification values and copyback distributions as the foundations of our analysis of the characteristics of the SSD system, we must be careful to use meaningful values for them from the simulation since these numbers can vary with time with the course of the simulation. If possible, we must use their values when they converge after a length of time running the simulation. If the values converge, the rates of convergence vary depending on the nature of the workload, the size of the SSD and other system parameters. Thus, before using these values for analysis, we must first analyze the convergence properties of the simulation so that we can precisely define the values obtained from simulation of write amplification and copyback distributions even though they vary with time.

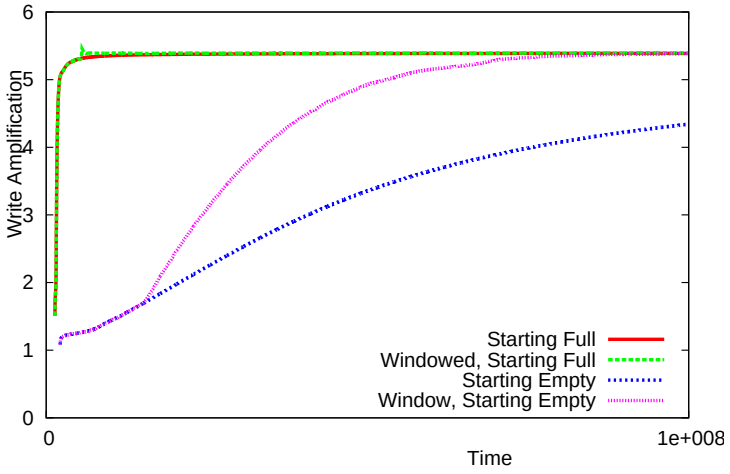
When we perform the simulation, we have various options about calculating the write amplification. The first option is that we can calculate the write amplification from every copyback event that has occurred during the entire simulation, i.e., the entire history of copybacks. Alternately, we can calculate it only from the latest set of copyback events from some specified time window, i.e., only a brief preceding history. The drawback of using the entire history is that values are always influenced by prior events in the simulation while the drawback of using recent history is that events in the small time window might influence the write amplification calculated too much. We will analyze the two different write amplification calculation methods as it relates to convergence and the speed of convergence.

A significant part of the simulation history is the start of the simulation and we can start the simulation with all the sectors already on disk (we call this *starting full*) or we start the simulation assuming that the blocks are empty and clean (we call this *starting empty*). When we start full, we start the simulation with all the sectors occupied and placed randomly in the flash memory. When we start empty, all the sectors are written as they come in. Starting empty leads to an initial period where the write amplification is low as sectors with low probability of being written are not on disk and this creates an effect like over-provisioning where the number of sectors stored on disk is significantly lower than the number of possible sectors. Staring full or empty has more effects to consider for trace workloads on static sectors and convergence. Trace workloads may omit writing to certain sectors during the course of the recorded trace as for example the OLTP trace does. If those sectors are assumed to be empty, then this lowers write amplification as those free pages act as over-provisioning. If we start full, we assume that those sectors that are never written to during the course of the trace are static sectors and which results in increased write amplification. Starting full or empty affects write amplification calculations and convergence and we must select the appropriate start for accurate values.

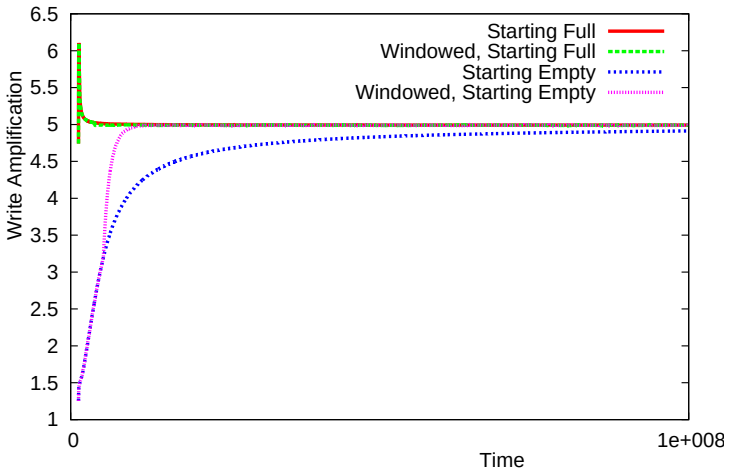
The convergence of write amplification is illustrated in Figure 5.13 where we plot write amplification against the time the simulation has been running. Each figure has four curves for starting full or empty, or using a windowed or complete history of copyback history to calculate the write amplification.

The first question is if convergence of write amplification values does occur. For synthetic workloads, the workload distribution stays the same and we expect the write amplification to converge eventually. For trace workloads, the workload distribution might change over time and we might not get convergence.

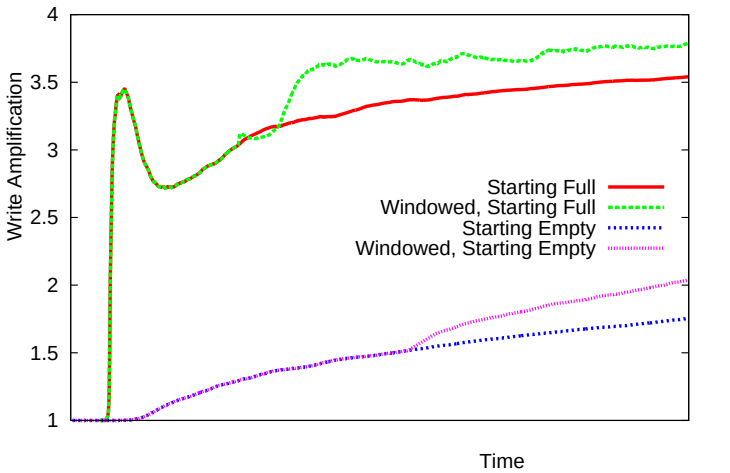
The second question is if it does converge, how fast does it converge and hence, how long should our simulations last. Synthetic workloads lengths can be increased arbitrarily



(a) Zipf Workload



(b) Uniform Workload



(c) Trace Workload (OLTP)

Figure 5.13: Convergence of Write Amplification

until we reach convergence but if the convergence is very slow then might take billions of operations before it reaches convergence and might not be practical to run the simulations for that length of time. Rare events in the workload that affect copybacks will take longer to convergence, for example in Zipf workloads sectors that are rarely updated. Workloads without any rare events converge fast, for example uniformly random workloads where each update of the sector is equal. As we can see in Figure 5.13a, the convergence of the Zipf workload is slower than uniform workloads than that in Figure 5.13b. However, factors like starting full or empty and using or not using windowed write amplification as our write amplification values affects the rate of convergence and we will use the ones that give us the fastest convergence. For trace workloads, the issue of convergence is trickier and more complicated as mentioned before. As we can see from Figure 5.13c, the windowed write amplification values do not converge and it reflects that the changing nature of the trace workload. However, the full cumulative write amplification values are smoother that average out the temporal fluctuations.

Next we want to look at which is the best method of looking at write amplification values from the various choices we have. For synthetic workloads starting full or empty should lead to the same write amplification value as there are no hidden static sectors. However, starting full, simulations for synthetic workloads converges much faster than starting empty and using the write amplification from a window of recent history converges even faster and thus, for the fastest convergence we use window history with full start to calculate the write amplification. In Figure 5.13b, we see that the uniform workloads starting full, empty, full or window history converge to the same value but starting empty converges much slower but is faster by using a window history. For Zipf workloads as show in Figure 5.13a, starting empty converges to the same value as starting full but doing so very slowly that by the end of the simulation period, it hasn't converged even though we have processed 1^8 operations. Though using the windowed history does converge after around 0.75^8 operations. In starting empty, each step increasing the write amplification

by a tiny amount but this adds a significant amount to the total write amplification over millions of operations. The rate of convergence is dependent on the number of sectors as higher number of sectors leads to more and more rarer events and the simulation in Figure 5.13a and 5.13b was done with $32,768$ sectors. Thus, for synthetic workloads, we start the simulation with all the sectors on disk and look at the recent history of copybacks for quicker convergence.

For trace workloads, as illustrated in Figure 5.13c, the write amplification on starting full is much higher than starting empty owing to the effect of sectors that are never written to during the course of the trace. The windowed values of write amplification varies because the trace workload varies with time which we can see as the non-converging curves which indicates that it does not make sense to talk about convergence with trace workloads. However, using the full history of the trace workload creates a smoother write amplification curve as it averages all the events that have occurred during the course of the trace. Thus, for trace workloads as well, we start the simulation with all the sectors on the disk but use the full history of write amplification at the end of the simulation.

In the case we do not achieve convergence in write amplification and copyback distribution, we use the values at the end of the simulation. This is not a problem since we are comparing algorithms and its effectiveness but these values are not valid for comparing across workloads. Thus, our analysis will deal with convergent values of write amplification when possible that elucidates the structure of the algorithms in the system, or the values at the end of the trace workloads when convergence is not reached.

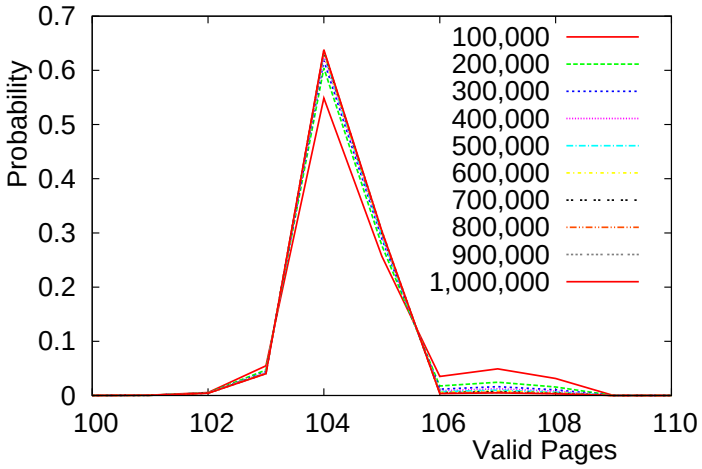
Another interesting relationship is the number of copyback against time. In Figure 5.17b, we see that for most curves that the relationship is linear with around 1 copyback in 100 time units except for the Zipf empty start. This is because for the starting empty curve, the sectors haven't been written to the disk and thus, the copyback per time unit time or the slope of the curve is lower first which increases to match the rest of the curves. Similarly, for the OLTP trace in 5.17c, the slopes of starting full and empty are different

since some of the sectors are not written to.

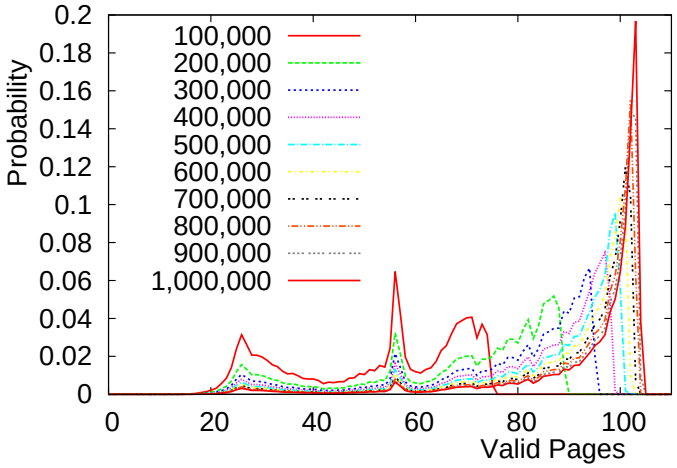
Next, we look at the convergence and properties of copyback distribution in the different simulation settings. This is shown in Figures 5.14, 5.15, 5.16 and 5.17, with each curve showing the copyback distribution curve after a certain number of copybacks have taken place. The conversion between the number of copyback and time is given in Figure 5.17. In Figure 5.14a, we look at the copyback distribution of the Zipf distribution with full start and it very quickly converges to its copyback distribution. With the empty start, the copyback distribution using the complete history slowly moves towards convergence in Figure 5.14b with the history of the empty start leaving a long left tail. Without the history moves to the convergence quicker in Figure 5.14c which ignores the history of the empty start. Similarly for the uniform workload with full start, the copyback converges quickly but faster than the Zipf workloads as in Figure 5.15a. Figure 5.15b shows the copyback distributions for an empty start and Figure 5.15c for an empty start but distributions calculated on a recent history window which follows a similar pattern to Zipf workloads. However, owing to the different distributions in the workloads, we have different shapes in the copyback distribution graphs.

For trace workloads, we see in Figures 5.16 and 5.17 that the copyback distributions for the OLTP trace workload is far more spread over the number of valid pages copied back. In terms of convergence, we see that with the full start and complete history, the convergence is the best. With the empty start and windowed, the distribution is moving to the right increasing write amplification and hasn't converged by the end of the trace. Since the trace is limited to what was recorded, we cannot expend the experiment further and thus, using the full start with complete history presents the best results for trace workloads.

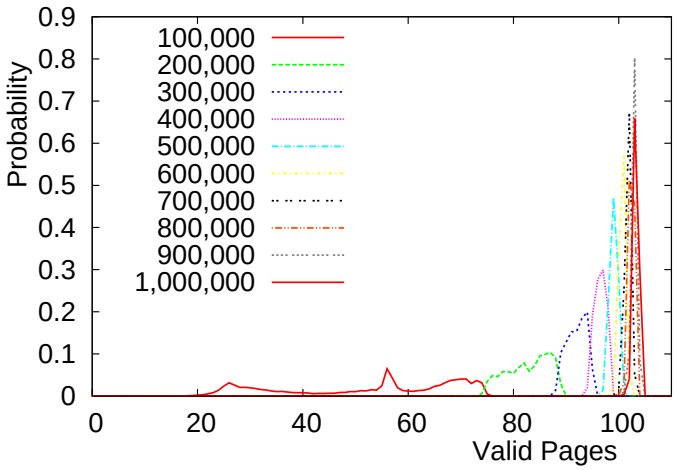
In this way, we can use a single write amplification number and a copyback distribution curve to analyze the operation of flash memory through a workload with millions of operations.



(a) Full Start, Complete History

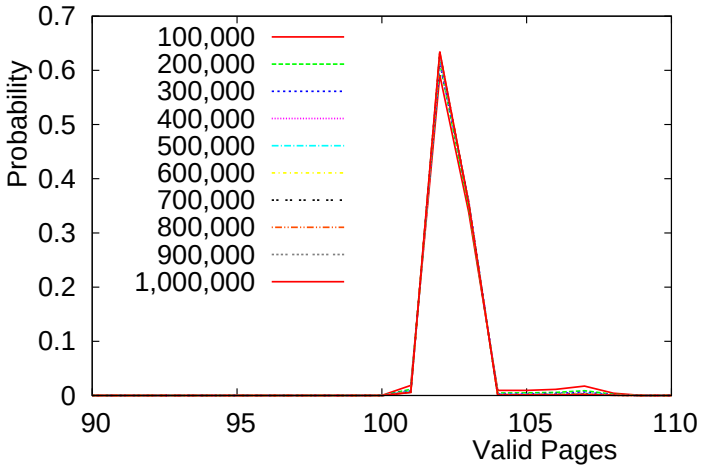


(b) Empty Start, Complete History

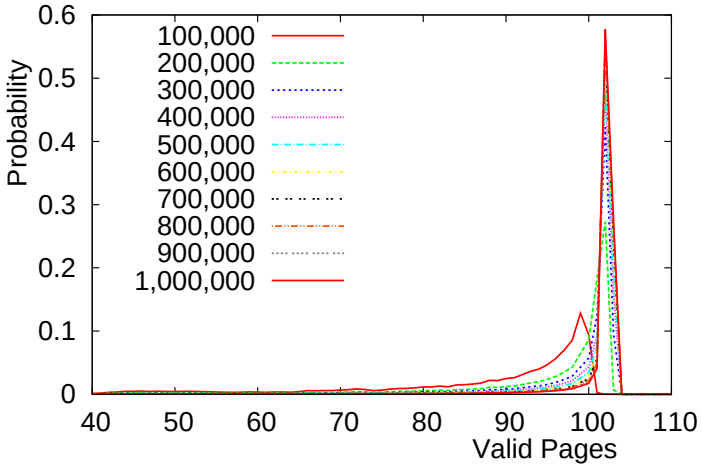


(c) Empty Start, Window

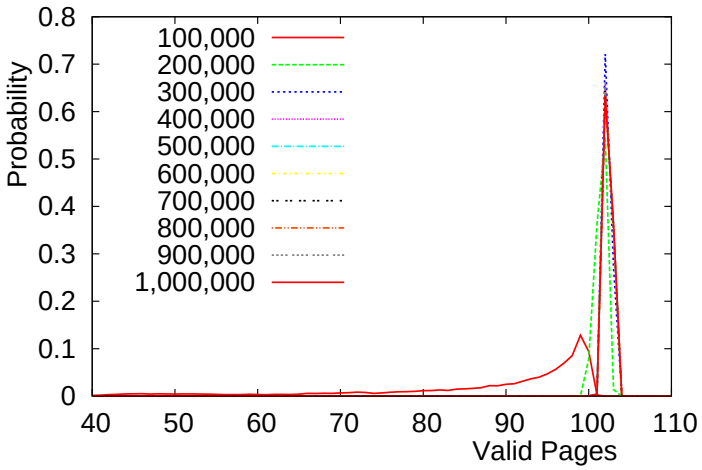
Figure 5.14: Convergence of Copyback Distributions of Zipf Workloads



(a) Full Start, Complete History

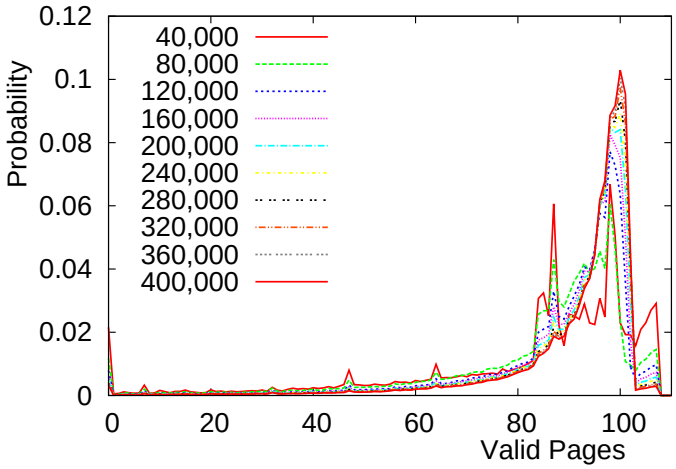


(b) Empty Start, Complete History

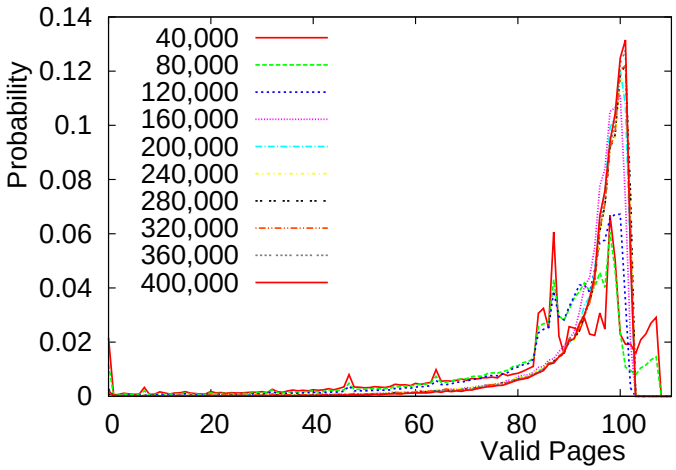


(c) Empty Start, Window

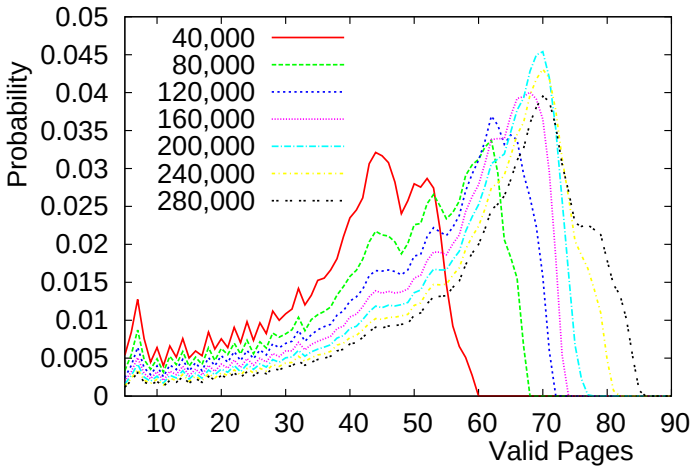
Figure 5.15: Convergence of Copyback Distributions of Uniformly Random Workloads



(a) Full Start, Complete History

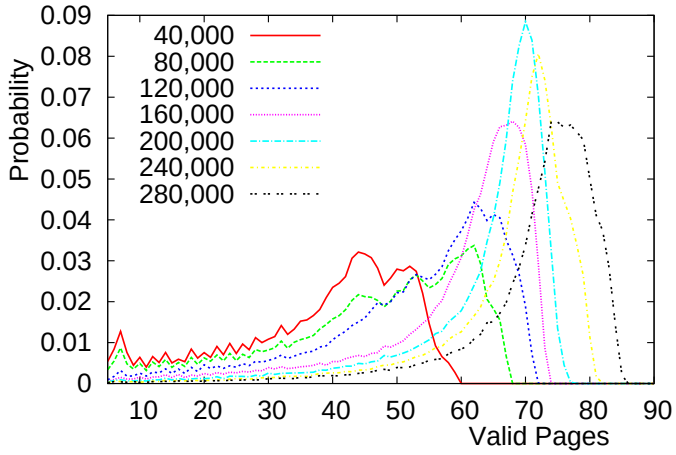


(b) Full Start, Window

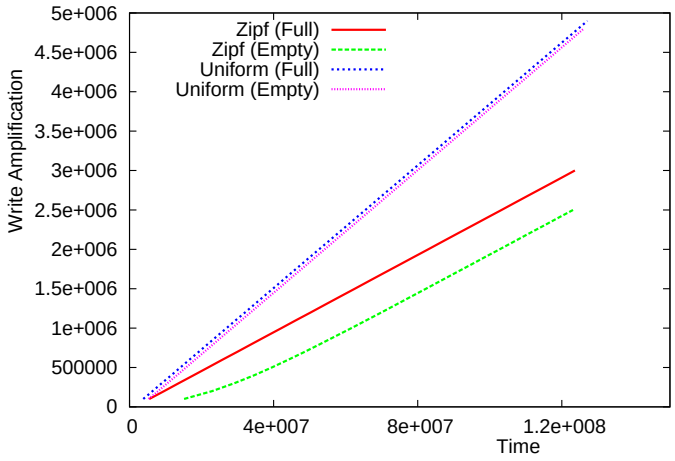


(c) Empty Start, Complete History

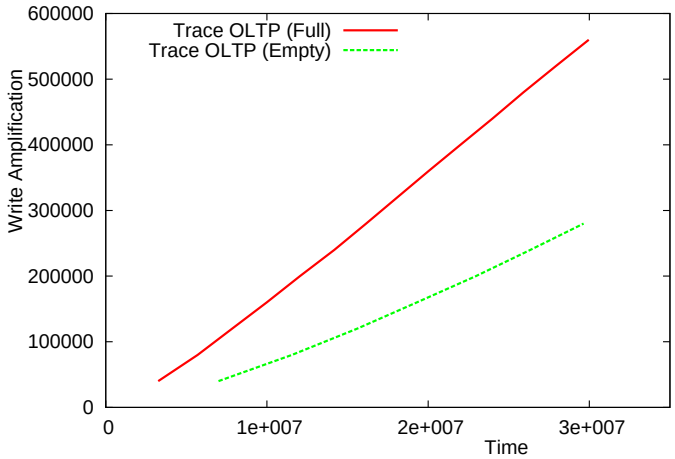
Figure 5.16: Convergence of Copyback Distributions of OLTP Trace Workload



(a) Empty Start, Window



(b) Copybacks vs Time for Zipf and Uniform



(c) Copybacks vs Time for Trace OLTP

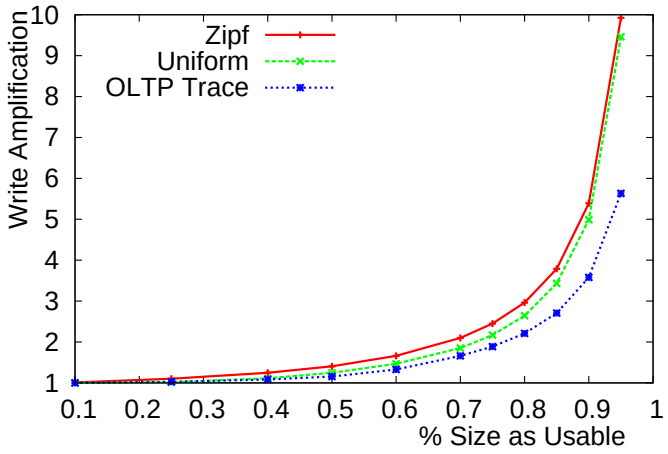
Figure 5.17: Convergence of Copyback Distributions of OLTP Trace Workload, Total Copybacks vs Time

5.5.1.3 Over-Provisioning

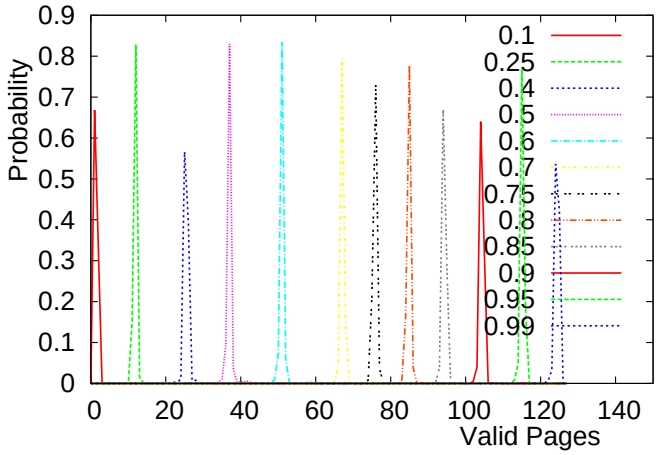
The factor that most influences write amplification in SSDs is over-provisioning and we will investigate our layout management algorithms across various degrees of over-provisioning. Figure 5.18a shows the write amplification versus the degree of over-provisioning for synthetic (uniformly random and Zipf) and trace workloads. We control the degree of over-provisioning by the percentage of the SSD size available. For synthetic workloads, we assume a flash die size and that only a certain fraction of the actual capacity is advertised as available which thus leads to lower number of sectors that can be stored in the disk. For trace workloads, the number of sectors remain the same and we change the size of the SSD so that the number of sectors represents the given percentage of the SSD. Though we have included the synthetic and trace workloads in the in the same graph, they write amplification values are not comparable because the number of sectors vary and the definition of how we measure over-provisioning varies.

From the simulation of write amplification and degree of over-provisioning, we see that the write amplification increases with less over-provisioning and the increase is very high for very low over-provisioning. For 0.95% useable level, write amplification is almost 10 while for 0.8%, it is ten times less at around 3 (for the synthetic workloads). In all levels of over-provisioning, the write amplification for uniformly random is slightly lower than for the Zipf workload.

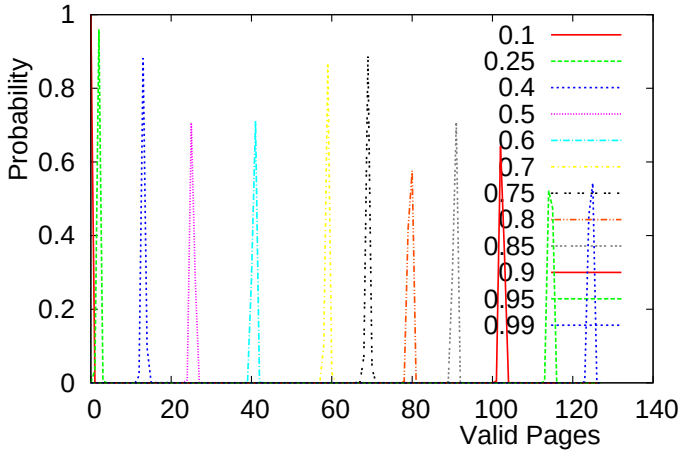
Figures 5.18b, 5.18c and 5.19a shows the copyback distributions for Zipf, uniformly random and OLTP trace workloads for various over-provisioning amounts. With less over-provisioning, the curves shift to the right and result in higher write amplification. The copyback distributions for the Zipf and uniformly random workloads are narrow and non-overlapping whereas the OLTP trace workload has wider over-lapping distributions which slightly change shape with higher over-provisioning.



(a) Write Amplification vs Over-provisioning

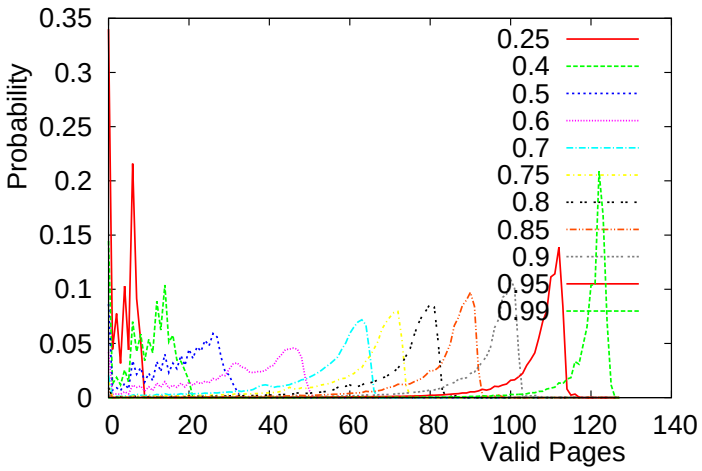


(b) Copyback Distributions for Various Over-provisioning for Zipf



(c) Copyback Distributions for Various Over-provisioning for Uniformly Random

Figure 5.18: Over-provisioning vs Write Amplification



(a) Copyback Distributions for Various Over-provisioning for OLTP Trace

Figure 5.19: Over-provisioning vs Write Amplification Additional

5.5.2 Using Copyback Counts

We compare the write amplifications

1. No copyback blocks and the writeblock used for copyback (None)
2. single copyback block and copyback blocks are different to writeblocks (2B)
3. multiple copyback blocks using our copyback algorithm (*n copyback blocks*).

We denote the rule of the multiple copyback blocks as

$$x_1, x_2, \ldots, x_N$$

where x_i is the copyback counts that make up the boundary of the copyback blocks. From the rule, we have $N+1$ copyback blocks $b_1, b_2, \ldots, b_{N+1}$: all pages with less than or equal to x_1 copyback count is copied back to block b_1, more than x_N is copied back to b_{N+1} and if between x_i and x_{i+1} copied back to block b_i. For example, a rule like $1, 2, 4$ would mean 4 copyback blocks b_1, b_2, b_3, b_4 where pages with 1 copyback count is copied to b_1, 2 copyback count to b_2, 3 and 4 copyback count to b_3 and more than 4 to b_4.

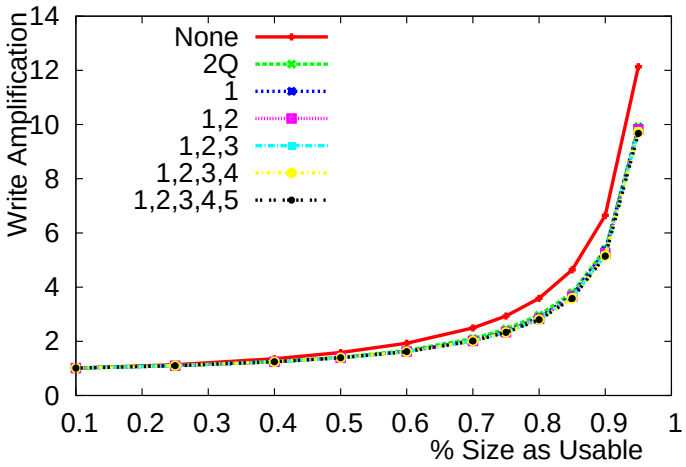
In our experiments, we start with a single copyback block and then add additional copyback blocks with rules as the following: none; 1; 1,2; 1,2,3; 1,2,3,4; and so on. The smallest granularity we can have between copyback counts is 1 and since we have a large number of blocks in the flash memory in the simulation, we give each copyback count its own block. This will give us the highest reduction in write amplification as each copyback count is copied back differently.

5.5.2.1 Simulation Results

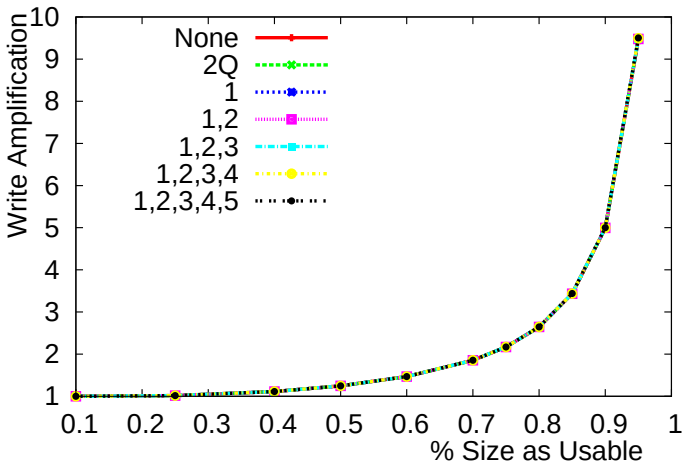
Figure 5.20 show the write amplification for various levels of over-provisioning and for using different copyback rules for Zipf, uniformly random workload and the OLTP trace workload. The standout characteristic is that for the Zipf and trace workloads switching from no copyback blocks to one or more copyback blocks give a significant decrease in write amplification but additional copyback blocks produce only very small decreases in write amplification. For uniformly random workloads, the write amplification is almost identical for all the different copyback blocks.

The same information is shown in a different format in Figure 5.21 where we plot with the number of copyback blocks in the x-axis. We see that when we go from no copyback block to one copyback block, i.e., switch from using the simple copyback algorithm to the multiple copyback block algorithm starting with a copyback block, the write amplification noticeably decreases. However, after that increasing the number of copyback blocks from 2 to higher values, the write amplification decreases only slightly.

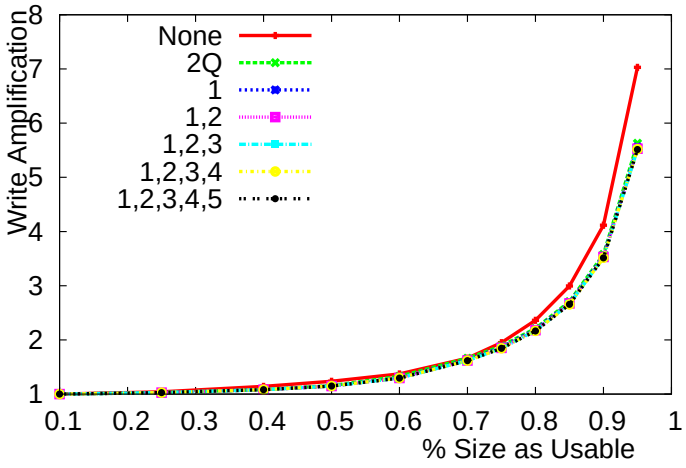
Tables 5.1, 5.2 and 5.3 shows the percentage decrease in write amplifications as we increase the number of copyback blocks. Going from top to bottom, we have various over-provisioning values of the flash memory in terms of the percentage of flash memory advertised as the capacity. Going from left to right, we have the percentage decrease in the write amplification by going from n copyback blocks to $n + 1$, starting with going from no copyback blocks (using writeblocks) to a single copyback block.



(a) Copyback Count Algorithm for Zipf

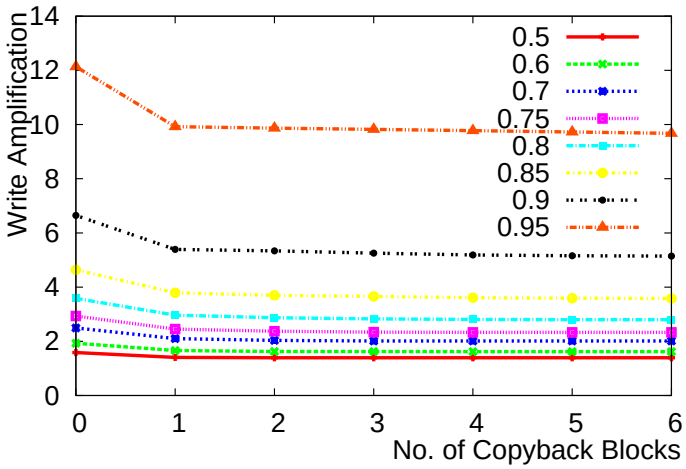


(b) Copyback Count Algorithm for Uniformly Random

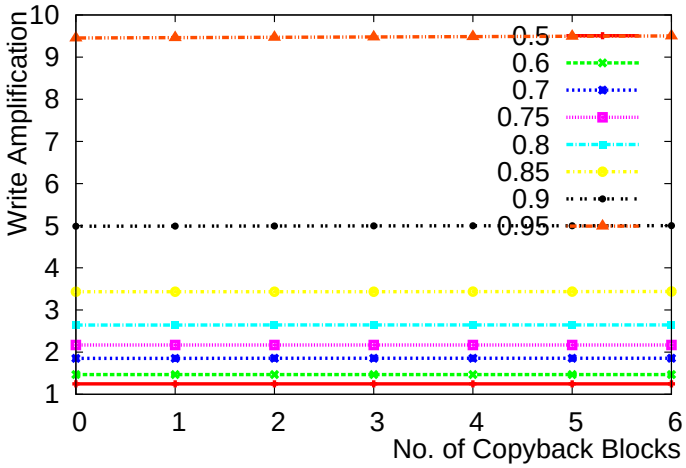


(c) Copyback Count Algorithm for OLTP Trace

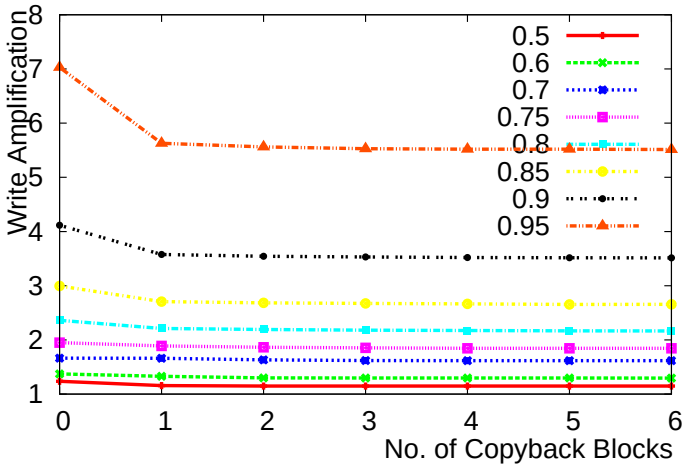
Figure 5.20: Over-provisioning and Copyback Count Algorithm vs Write Amplification



(a) Zipf Workload



(b) Unif. Random Workload



(c) Trace Workload

Figure 5.21: Write Amplification vs No. of Copyback Blocks for Various Over-provisioning

	None to 2B	1	1,2	1,2,3	1,2,3,4	1,2,3,4,5	Multi CBB	Total Reduced
0.1	7.51	-0.12	-0.10	0.08	-0.07	-0.12	-0.33	7.20
0.25	23.55	0.21	-0.02	-0.02	-0.00	-0.02	0.14	23.66
0.4	30.49	1.23	0.04	-0.02	-0.01	-0.01	1.24	31.35
0.5	30.80	3.17	0.16	0.01	-0.02	-0.01	3.30	33.08
0.6	28.84	5.52	0.50	0.04	-0.00	-0.00	6.02	33.12
0.7	26.25	6.05	1.91	0.28	0.04	-0.00	8.14	32.25
0.75	24.97	5.57	2.43	0.42	0.12	0.04	8.39	31.27
0.8	23.95	4.86	2.20	1.31	0.26	0.07	8.47	30.39
0.85	23.30	3.26	1.63	1.76	0.74	0.23	7.42	28.99
0.9	22.26	1.23	1.90	1.59	0.64	0.37	5.61	26.62
0.95	19.85	0.61	0.55	0.51	0.56	0.57	2.77	22.07

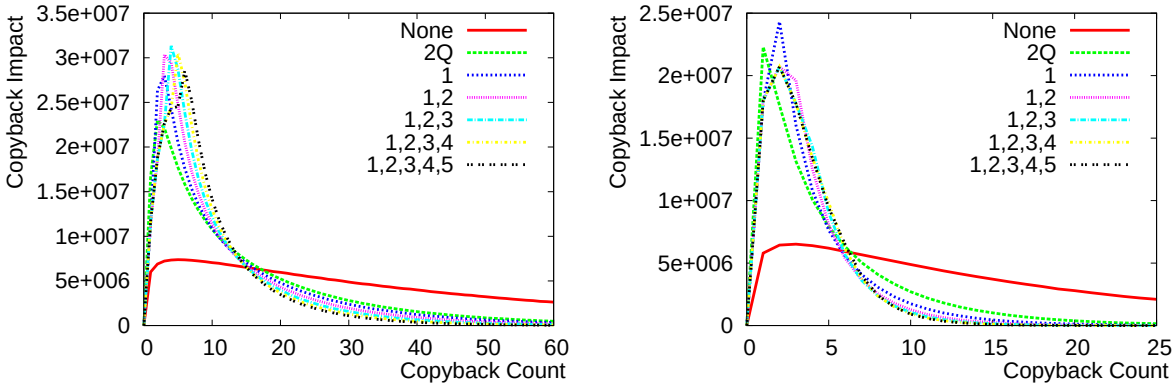
Table 5.1: Percentage Write Amplification Decreases for Zipf Workloads

	None to 2B	1	1,2	1,2,3	1,2,3,4	1,2,3,4,5	Multi CBB	Total Reduced
0.1	-	-	-	-	-	-	-	-
0.25	0.00	-0.02	-0.03	-0.09	-0.03	-0.05	-0.22	-0.22
0.4	-0.02	-0.03	-0.02	-0.03	-0.02	-0.04	-0.14	-0.15
0.5	-0.01	-0.02	-0.02	-0.02	-0.02	-0.01	-0.09	-0.10
0.6	-0.03	-0.02	-0.02	-0.02	-0.02	-0.02	-0.09	-0.12
0.7	-0.02	-0.02	-0.03	-0.02	-0.02	-0.03	-0.12	-0.14
0.75	-0.02	-0.04	-0.02	-0.03	-0.02	-0.03	-0.13	-0.15
0.8	-0.03	-0.02	-0.03	-0.03	-0.03	-0.03	-0.14	-0.18
0.85	-0.03	-0.05	-0.03	-0.05	-0.03	-0.04	-0.19	-0.22
0.9	-0.06	-0.06	-0.05	-0.06	-0.06	-0.05	-0.27	-0.33
0.95	-0.10	-0.10	-0.10	-0.10	-0.09	-0.11	-0.49	-0.59

Table 5.2: Percentage Write Amplification Decreases for Uniformly Random Workloads

	None to 2B	1	1,2	1,2,3	1,2,3,4	1,2,3,4,5	Multi CBB	Total Reduced
0.1	-	-	-	-	-	-	-	-
0.25	37.19	-0.12	0.11	-0.21	0.13	0.18	0.09	37.25
0.4	41.90	0.08	-0.02	0.01	-0.11	0.32	0.28	42.06
0.5	33.95	4.42	0.44	-0.32	0.12	-0.09	4.57	36.96
0.6	11.97	8.84	0.51	0.51	-0.29	-0.21	9.33	20.18
0.7	0.49	4.31	1.90	0.35	0.28	0.00	6.72	7.18
0.75	6.34	2.76	1.32	0.62	0.45	-0.21	4.87	10.90
0.8	11.10	1.59	0.80	0.69	0.43	0.06	3.53	14.24
0.85	14.45	1.51	0.68	0.51	0.22	0.15	3.04	17.06
0.9	17.30	1.38	0.56	0.19	0.17	0.18	2.45	19.32
0.95	23.24	1.58	0.51	0.29	0.03	-0.03	2.37	25.06

Table 5.3: Percentage Write Amplification Decreases for Trace Workload



(a) Copyback Impacts for Zipf at 0.9 Size Usable

(b) Zipf Workload, 0.75 Size Usable

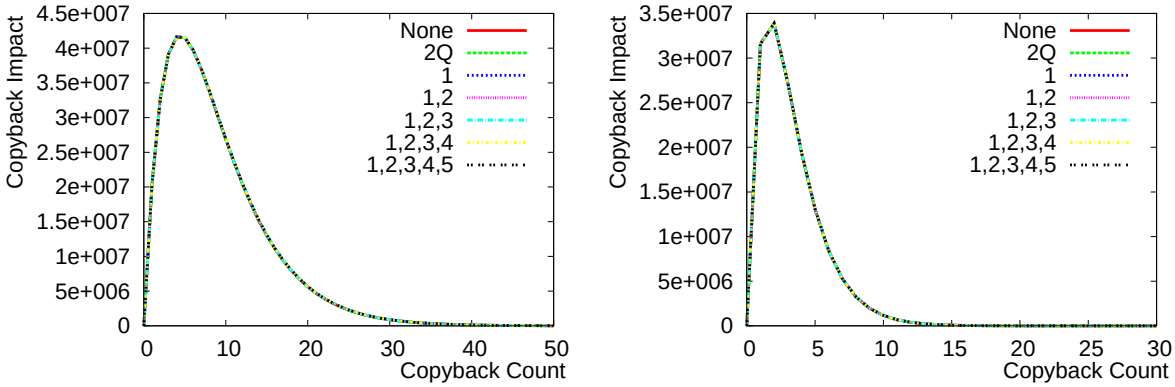
Figure 5.22: Copyback Impacts Graphs for different number of Copyback Blocks for Zipf Workload

As we saw from the graphs for Zipf and trace workload, adding the first copyback block created a large decrease in write amplification. The further addition of copyback blocks created a small decrease in write amplification with each additional copyback block being less effective. For uniformly random workloads, there was an increase in write amplification (as shown by a negative decrease). As we used a block for copybacks, it minutely decreased over-provisioning and thus, resulted in higher write amplification.

5.5.2.2 Analysis of Copyback Impacts

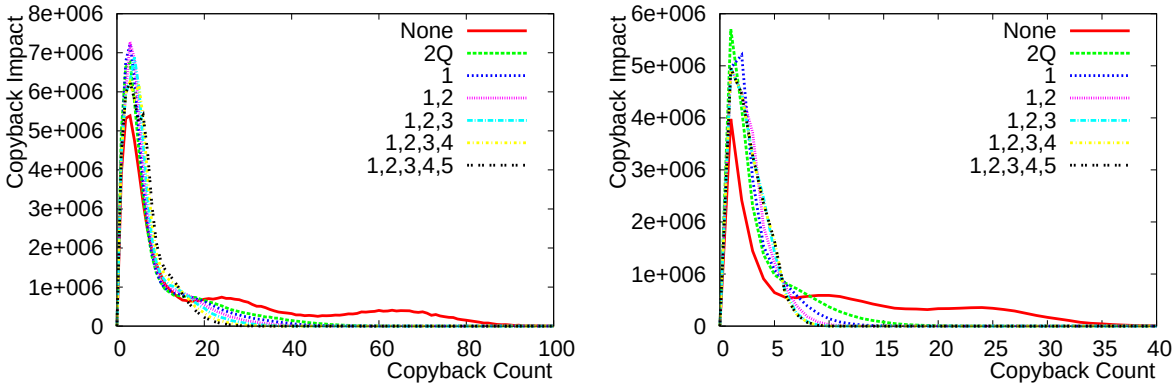
Next we look at how our copyback count algorithms affect the functioning of the SSD to explain the reduction in the write amplification. The main tool that we will use is *copyback impacts*.

In the simulation, every time a page is copied, we keep track of the copyback count of the page. This tells us how many times previously the data in the page has been copied back. Next, we take the histogram of the recorded copyback counts during the copyback of a page and this graphed histogram is the copyback impact. In this way, we have a quantitative measure of the repeated copyback of data and the graph a visualization of that.



(a) Unif. Random Workload, 0.9 Size Usable (b) Unif. Random Workload, 0.75 Size Usable

Figure 5.23: Copyback Impacts Graphs for different number of Copyback Blocks for Unif Random Workload



(a) OLTP Trace Workload, 0.9 Size Usable (b) OLTP Trace Workload, 0.75 Size Usable

Figure 5.24: Copyback Impacts Graphs for different number of Copyback Blocks for OLTP Trace

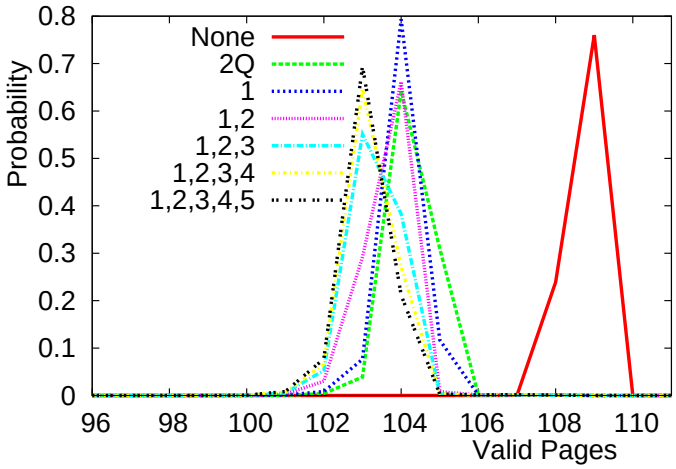
First, we look at how the copyback impacts differ between the algorithms and the rules of the copyback counts algorithm. In Figures 5.22a and 5.22b, we have the copyback impact graphs for the Zipf workload with 90% and 75% of the space advertised for storage and 10% and 25% used for over-provisioning.

When not using copyback blocks, the copyback impact distribution is flatter meaning that copybacks of pages with high copyback counts are occurring. When we add copyback blocks, the copyback impact distribution develops a much larger peak in the lower copyback counts. This means that there are more copybacks with low copyback counts and less with high copyback counts. This is the intended result of our design where we wanted to decrease the repeated copybacks of static pages.

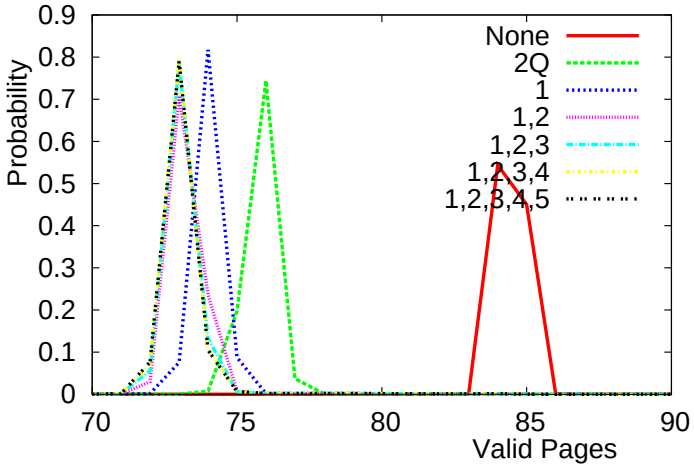
With multiple copyback blocks, the pages are copied back slightly less and the entire copyback impact graph moves to the left resulting in lesser copybacks and hence, lower write amplification. As we increase the number of copyback blocks, the copyback impact graphs do move to the left a bit more but a very small amount and thus, only a small change in the write amplification.

For uniformly random workloads, the copyback impact distributions is show in Figure 5.23a and 5.23b which shows only minute changes in the distributions when we change copyback blocks. For the OLTP trace workload, we see a similar trend as for Zipf workloads where the multiple copyback blocks reduce the impact from high copyback counts as given in Figures 5.24a and 5.24b.

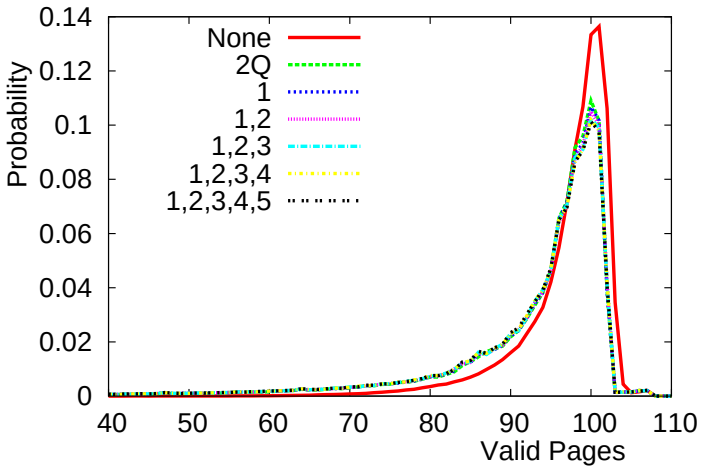
Thus, from the copyback impact distributions, we can see that the algorithm achieves what we intended with our algorithm. By using copyback blocks separate from write-blocks, we have essentially removed the sedimentation problem. After removing the sedimentation problem, the increased copyback blocks however yield only a smaller change in the copyback impacts.



(a) Zipf Workload, 0.9 Size Usable

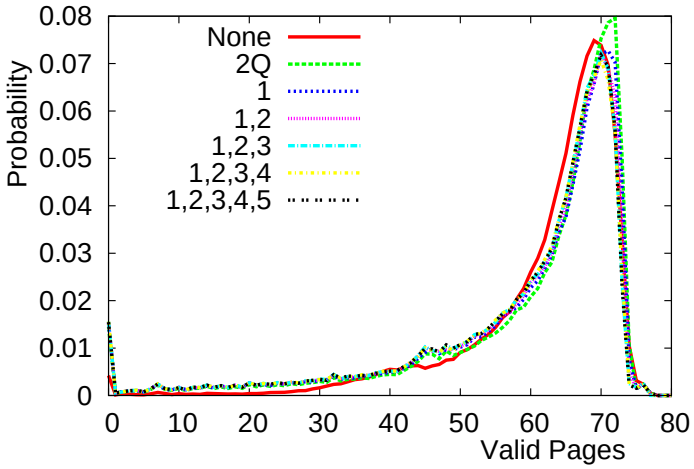


(b) Zipf Workload, 0.75 Size Usable

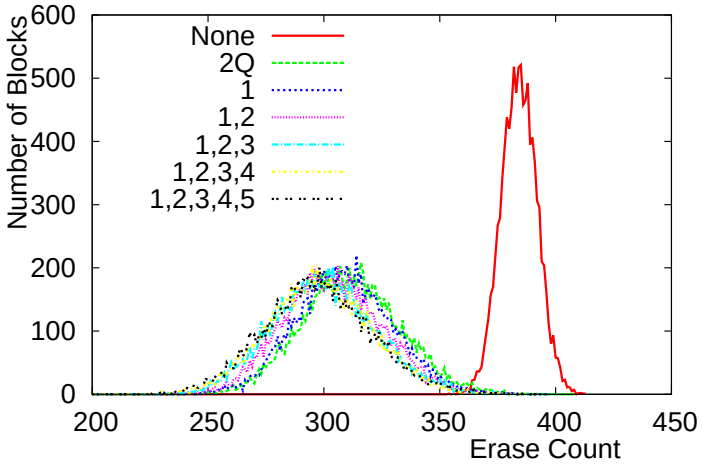


(c) OLTP Trace Workload, 0.9 Size Usable

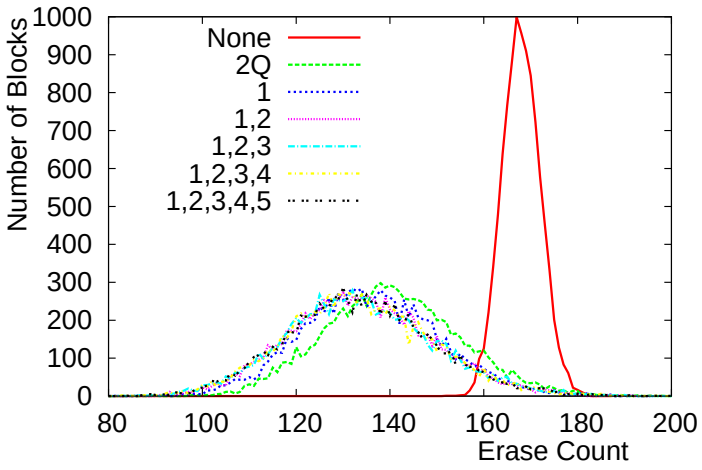
Figure 5.25: Copyback Distributions for different number of Copyback Blocks



(a) OLTP Trace Workload, 0.75 Size Usable

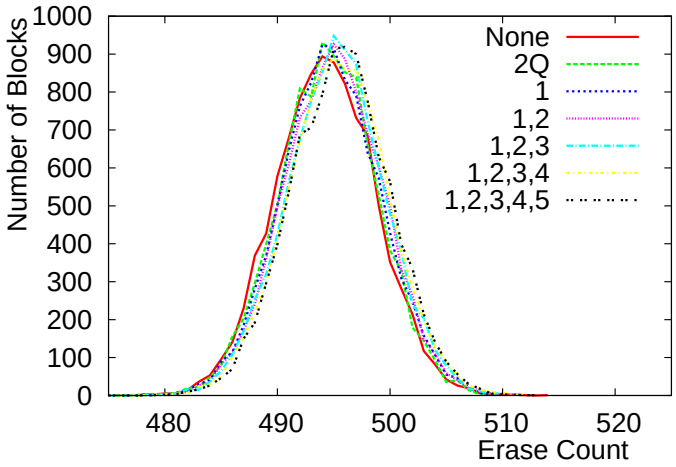


(b) Zipf Workload, 0.9 Size Usable, Wear Distribution

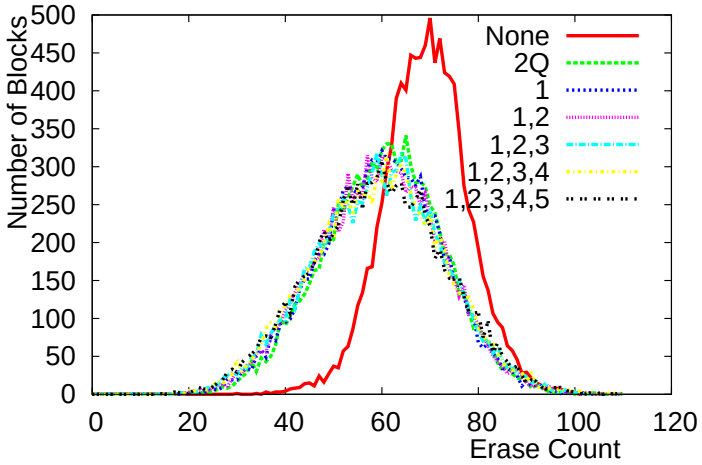


(c) Zipf Workload, 0.75 Size Usable, Wear Distribution

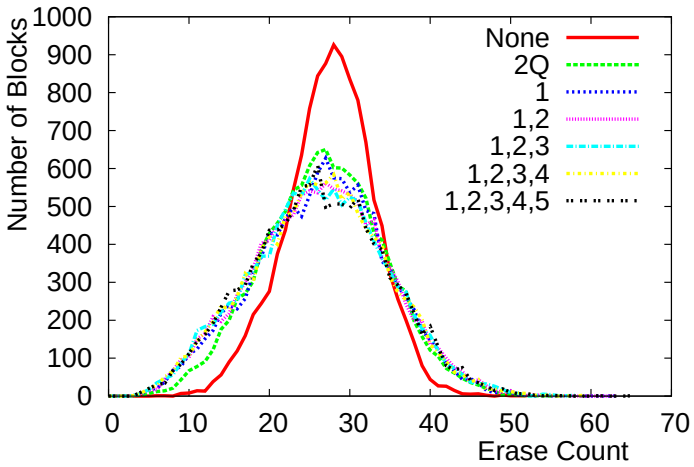
Figure 5.26: Copyback and Wear Distributions for different number of Copyback Blocks



(a) Unif Random Workload, 0.9 Size Usable



(b) OLTP Trace Workload, 0.9 Size Usable



(c) OLTP Trace Workload, 0.75 Size Usable

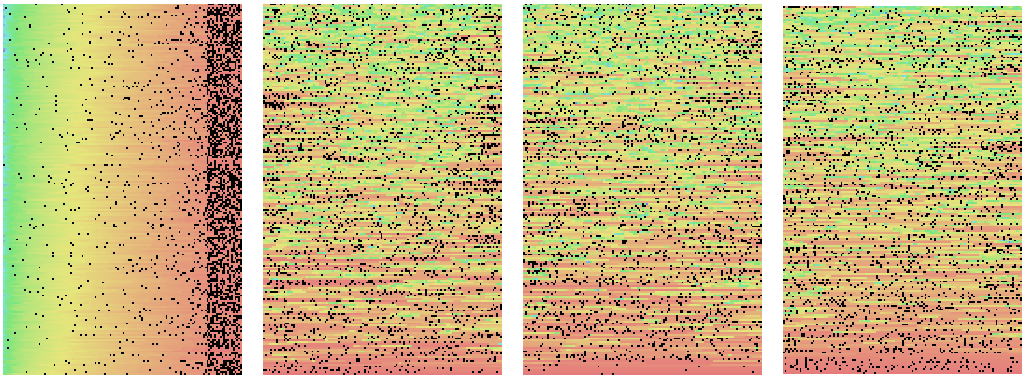
Figure 5.27: Wear Distributions

5.5.2.3 Copyback Distributions and Heatmap Analysis

Another view of the effects of the copyback blocks algorithm is the changes in the copyback distribution. In Figures 5.25a and 5.25b, we have the changes for the Zipf distribution with the over-provisioning set for 0.9 and 0.75 usable and in Figures 5.25c and 5.26a for the OLTP trace workload with over-provisioning set for 0.9 and 0.75 usable. From previous data, we know that the write amplification is decreased by using the copyback blocks algorithm. For the Zipf workload, the copyback distribution moves to the left with the introduction of the copyback blocks. For the trace workload, instead of a move of the distribution, we see a change in the shape of the distribution. With copyback blocks, there are more blocks that are copied back with lower number of valid pages and thus, the left tail of the copyback distributions are bigger for copyback blocks based algorithms.

The most striking illustration of using multiple copyback blocks over a single copyback block comes from heat maps. We have introduced heat maps in Section 5.3 and we analyze the heat maps in more detail here. In Figure 5.28a, we see the typical heat map of a no copyback block where the cold and static sectors sediment to the low pages of the block while the hot pages are written to the higher pages of the block. With the use of separate copyback blocks, this process is disrupted as seen in Figures 5.28b to 5.28g where some blocks contain cold data and some blocks hot data. Thus, instead of forming sediments to the bottom of each block and being copied over and over again, static data goes into a static block which can stay untouched and not result in multiple copybacks. The blue/green colored static or cold data is seen across blocks horizontally rather than as sediments vertically. As the number of copyback blocks increases, there is a slight improvement in the number of blocks with hot pages as seen with more reddish hue at the bottom of the figures. This results in the slight decrease in write amplification.

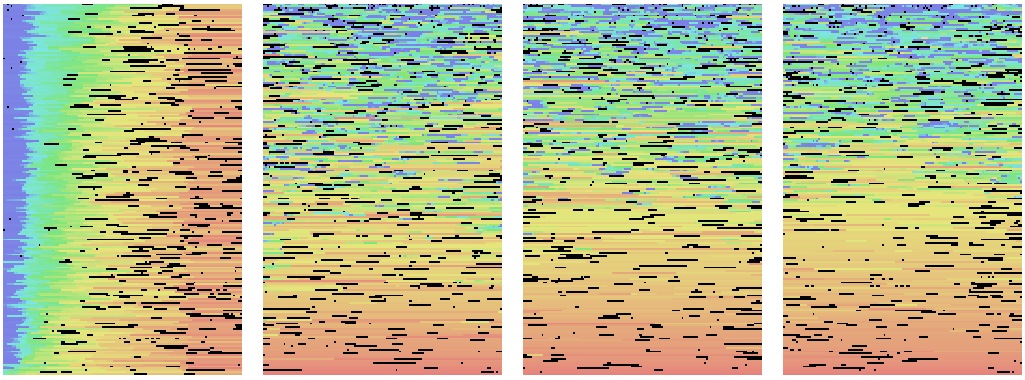
The heat maps for the trace OLTP workload is similar to the synthetic Zipf workload as in Figure 5.30. Since the trace workload has static data, there is more blue or cold data in the heat map for the workload. The multiple copyback algorithm again shifts static



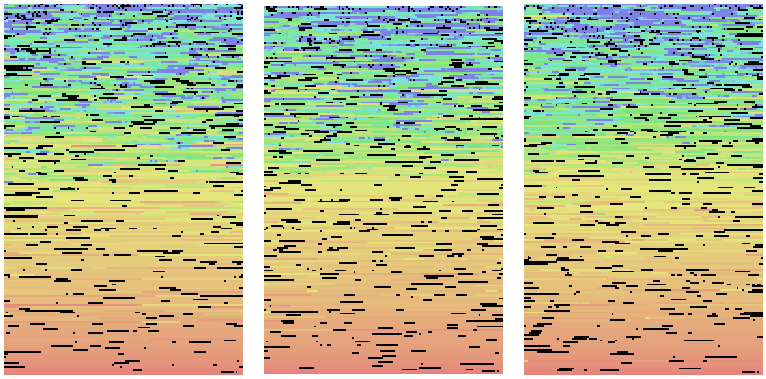
(a) None (b) 2B (c) 1 (d) 1,2

(e) 1,2,3 (f) 1,2,3,4 (g) 1,2,3,4,5

Figure 5.28: Heat-Maps: Zipf Workload and 0.9 Size Usable

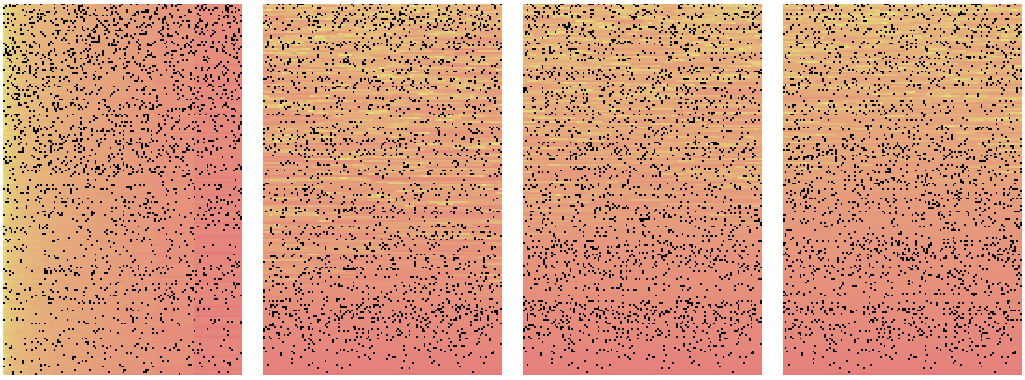


(a) None (b) 2B (c) 1 (d) 1,2

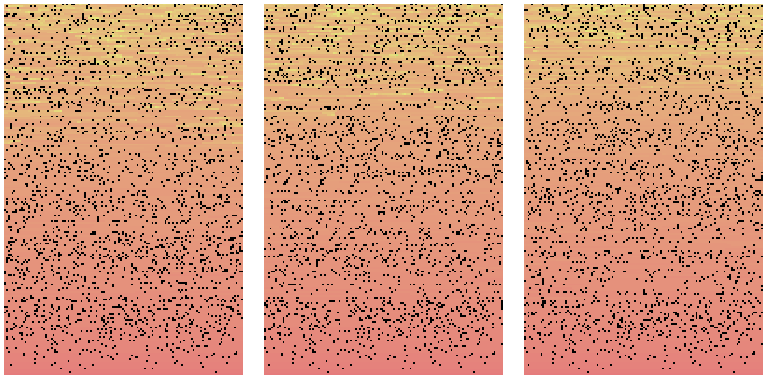


(e) 1,2,3 (f) 1,2,3,4 (g) 1,2,3,4,5

Figure 5.29: Heat-Maps: OLTP Trace Workload and 0.9 Size Usable



(a) None (b) 2B (c) 1 (d) 1,2



(e) 1,2,3 (f) 1,2,3,4 (g) 1,2,3,4,5

Figure 5.30: Heat-Maps: Unif Random Workload and 0.9 Size Usable

and dynamic data to different blocks instead of allowing the static blocks to sediment to lower pages in the blocks.

The heat maps for the uniformly random workload is given in Figure 5.30. There is very slight sedimentation in Figure 5.30a which is disrupted with the copyback blocks based algorithms. However, since there is no real static data, this does not translate to any write amplification reduction as each page has equal chance of being updated at every step.

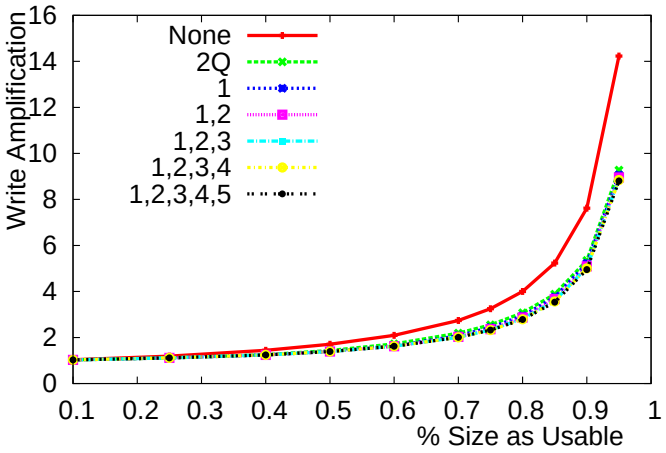
5.5.2.4 No Cache

When using a cache, operations that update a sector in less than the size of the cache are removed. By looking at the results from removing the cache altogether, we can observe the effects of operations that update the sectors quickly and also understand the role of caches within the context of copyback blocks.

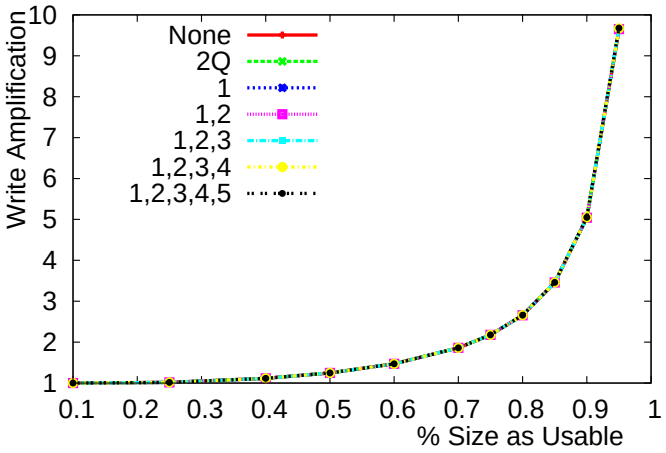
Figures 5.31 and 5.32 shows the write amplification against over-provisioning for the different copyback blocks algorithm parameters (the equivalent of Figures 5.20 and 5.21 when we used the cache size of 1280 sectors. The percentage changes in write amplification is given in Tables 5.4, 5.5 and 5.6. We see that the write amplification is higher without the cache and the percentage decreases in write amplification are also higher.

When sectors that are very quickly updated are added to the workload from the removal of the cache, static sectors end up getting copied back more often. By using copyback blocks and separating out static or cold data, we are more effective at reducing write amplification. The highly volatile sectors are invalidated before they are copied back and thus never make it to the copyback blocks. This gives more separation to the static and dynamic data and thus more effective write amplification decrease.

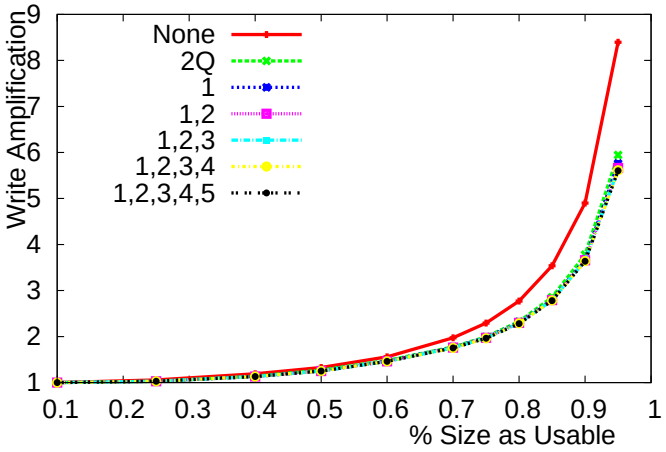
The increased copybacks of static or cold sectors is shown in copyback impact graphs in Figures 5.33a and 5.33c for the Zipf and trace workloads. The copyback impacts graphs without the cache are flatter and this is because of the larger number of repeated



(a) Copyback Count Algorithm for Zipf

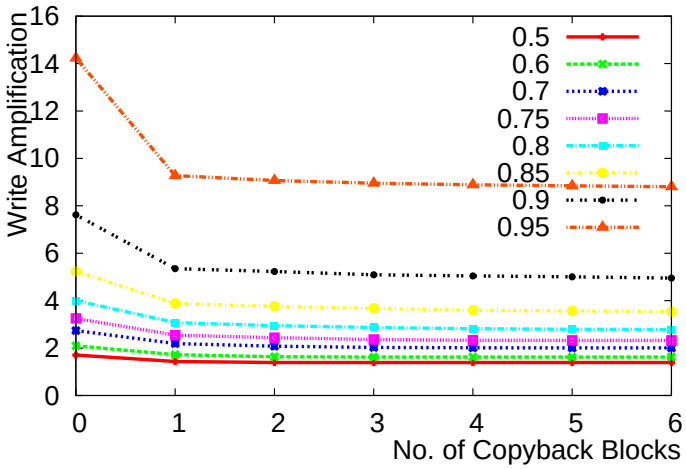


(b) Copyback Count Algorithm for Uniformly Random

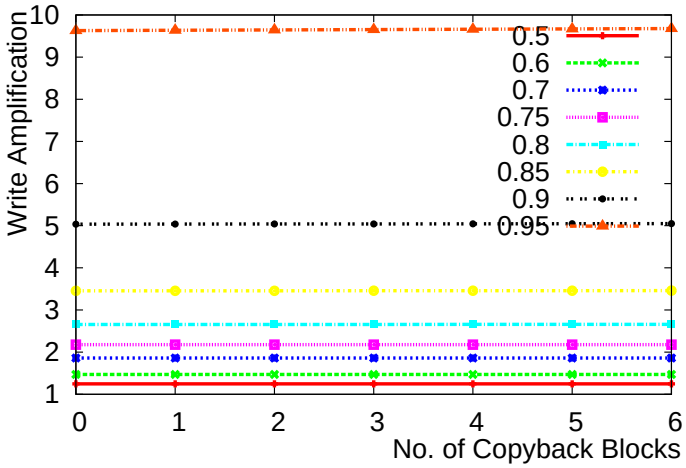


(c) Copyback Count Algorithm for OLTP Trace

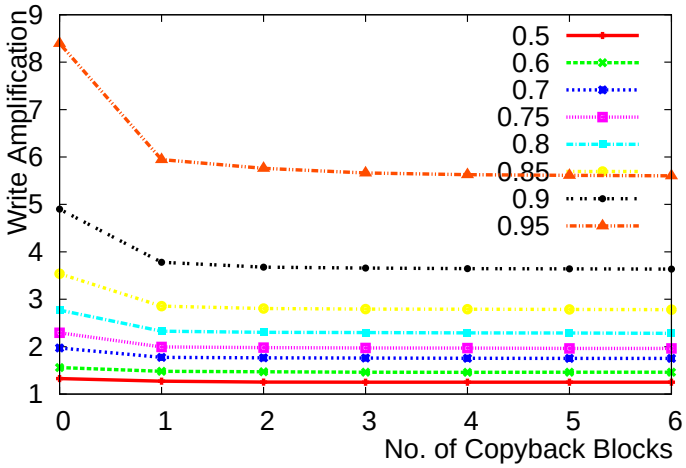
Figure 5.31: Over-provisioning and Copyback Count Algorithm vs Write Amplification (no Cache)



(a) Zipf Workload



(b) Unif. Random Workload



(c) Trace Workload

Figure 5.32: Write Amplification vs No. of Copyback Blocks for Various Over-provisioning

	None to 2B	1	1,2	1,2,3	1,2,3,4	1,2,3,4,5	Multi CBB	Total Reduced
0.1	21.37	0.00	-0.04	-0.01	-0.00	-0.08	-0.12	21.27
0.25	40.38	0.62	0.02	-0.01	0.00	-0.05	0.59	40.73
0.4	43.06	4.38	0.10	-0.01	-0.01	-0.02	4.44	45.59
0.5	38.60	9.71	0.48	-0.01	0.01	-0.07	10.08	44.79
0.6	34.09	11.74	2.40	0.16	-0.04	0.00	13.96	43.29
0.7	31.49	9.44	4.84	1.41	0.23	-0.01	15.22	41.92
0.75	31.06	7.26	5.42	2.21	0.35	0.10	14.61	41.13
0.8	30.91	6.17	4.00	2.82	1.38	0.50	14.10	40.65
0.85	31.94	4.36	3.19	2.84	1.14	1.08	12.02	40.12
0.9	34.29	2.79	3.29	1.14	0.94	1.30	9.13	40.29
0.95	37.45	2.51	1.44	0.81	0.54	0.48	5.65	40.99

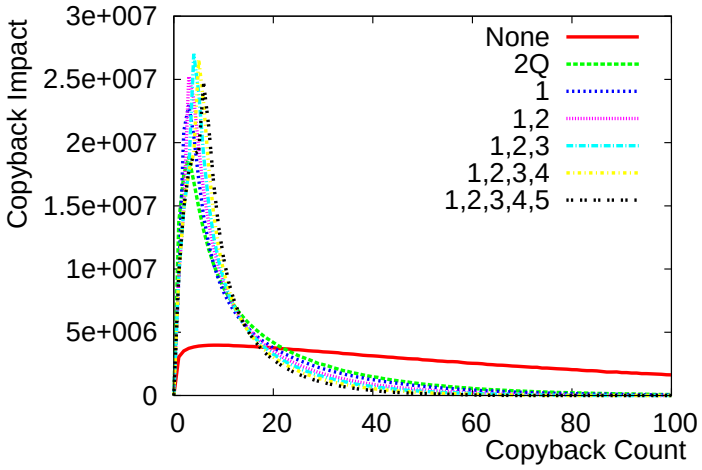
Table 5.4: Percentage Write Amplification Decreases for Zipf Workloads (no Cache)

	None to 2B	1	1,2	1,2,3	1,2,3,4	1,2,3,4,5	Multi CBB	Total Reduced
0.1	-	-	-	-	-	-	-	-
0.25	-0.07	-0.02	-0.09	-0.03	-0.07	-0.06	-0.26	-0.34
0.4	-0.01	-0.02	-0.02	-0.02	-0.05	-0.03	-0.13	-0.14
0.5	-0.02	-0.02	-0.01	-0.02	-0.02	-0.01	-0.09	-0.11
0.6	-0.04	-0.00	-0.04	-0.01	-0.02	-0.01	-0.07	-0.11
0.7	-0.02	-0.01	-0.04	-0.03	-0.00	-0.03	-0.11	-0.14
0.75	-0.02	-0.03	-0.02	-0.04	-0.02	-0.04	-0.14	-0.16
0.8	-0.03	-0.03	-0.03	-0.03	-0.04	-0.03	-0.15	-0.18
0.85	-0.04	-0.03	-0.06	-0.02	-0.04	-0.03	-0.18	-0.22
0.9	-0.05	-0.07	-0.05	-0.06	-0.05	-0.05	-0.27	-0.32
0.95	-0.11	-0.09	-0.09	-0.10	-0.10	-0.09	-0.47	-0.58

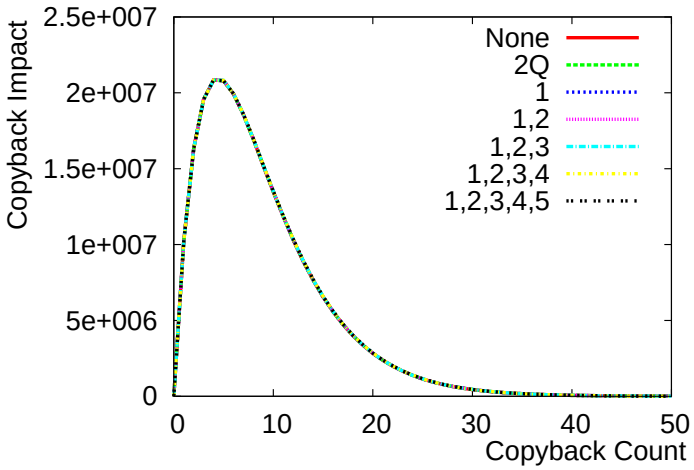
Table 5.5: Percentage Write Amplification Decreases for Uniformly Random Workloads (no Cache)

	None to 2B	1	1,2	1,2,3	1,2,3,4	1,2,3,4,5	Multi CBB	Total Reduced
0.1	5.08	-0.11	0.09	-0.15	0.11	-0.06	-0.12	4.96
0.25	49.02	-0.06	-0.05	0.05	0.02	0.03	-0.01	49.01
0.4	27.87	4.43	-0.11	-0.12	0.23	-0.31	4.14	30.86
0.5	16.66	7.62	0.32	0.18	-0.09	-0.01	7.99	23.32
0.6	14.26	1.87	2.36	0.07	-0.05	-0.04	4.16	17.83
0.7	20.31	1.31	0.65	0.65	0.27	0.06	2.91	22.63
0.75	23.18	1.48	0.59	0.33	0.66	0.01	3.05	25.52
0.8	25.10	1.78	0.42	0.48	0.31	0.44	3.38	27.64
0.85	26.92	2.73	0.68	0.12	0.21	0.35	4.05	29.88
0.9	28.69	3.73	0.73	0.42	0.20	0.19	5.20	32.40
0.95	33.11	3.75	2.02	0.81	0.27	0.24	6.94	37.76

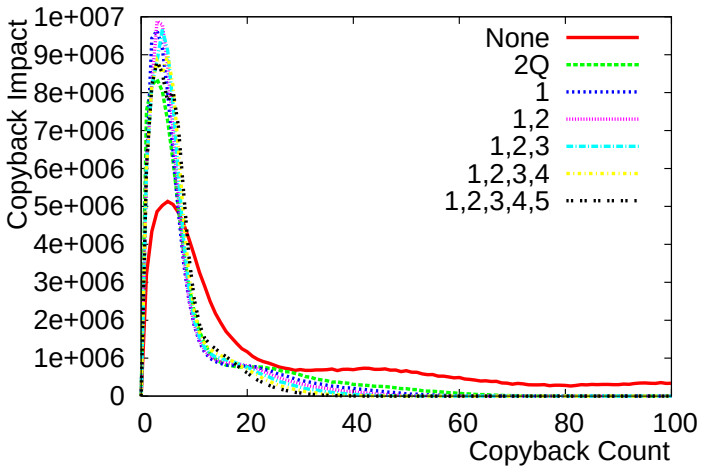
Table 5.6: Percentage Write Amplification Decreases for OLTP Trace Workload (no Cache)



(a) Zipf

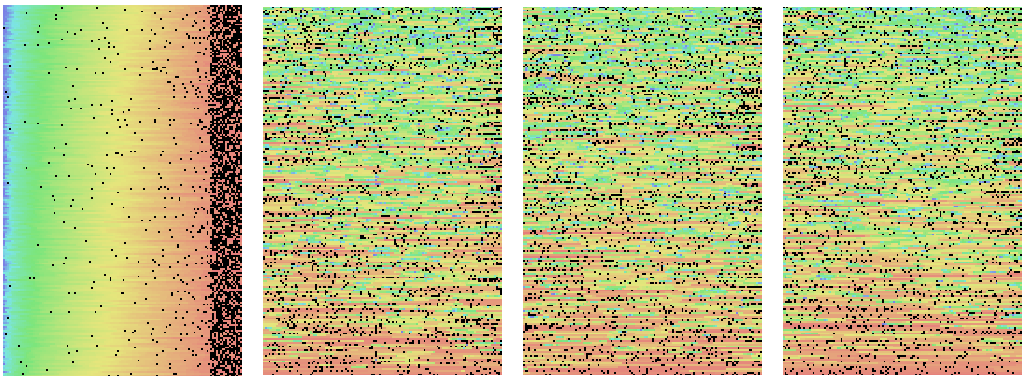


(b) Uniformly Random

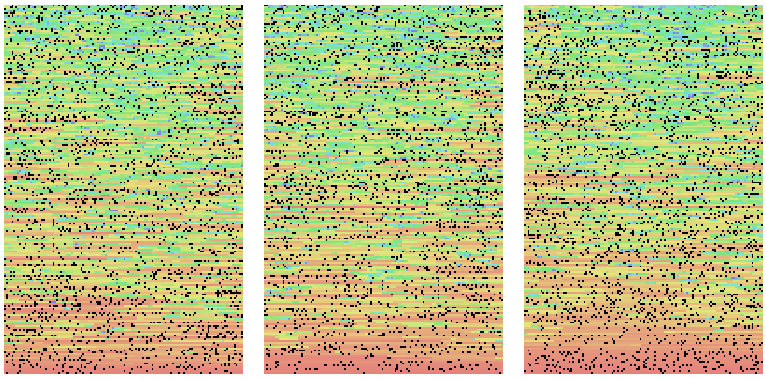


(c) Trace

Figure 5.33: Copyback Impacts at 0.9 Usable (No Cache)



(a) None (b) 2B (c) 1 (d) 1,2



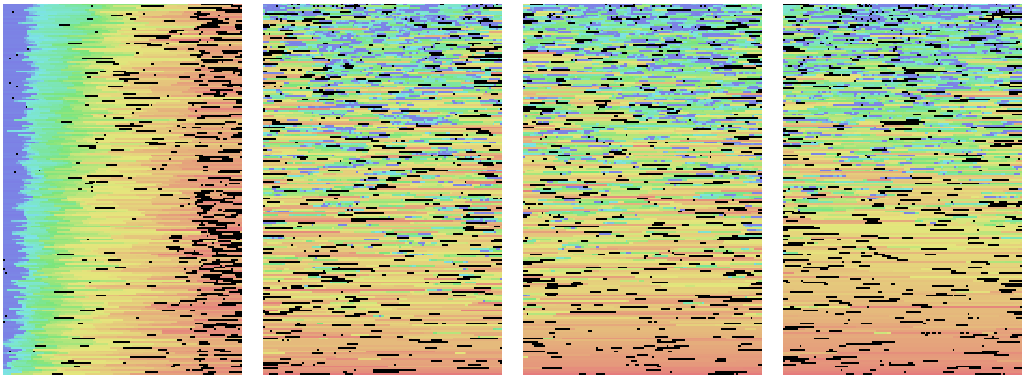
(e) 1,2,3 (f) 1,2,3,4 (g) 1,2,3,4,5

Figure 5.34: Heat-Maps: Zipf Workload and 0.9 Size Usable (No Cache)

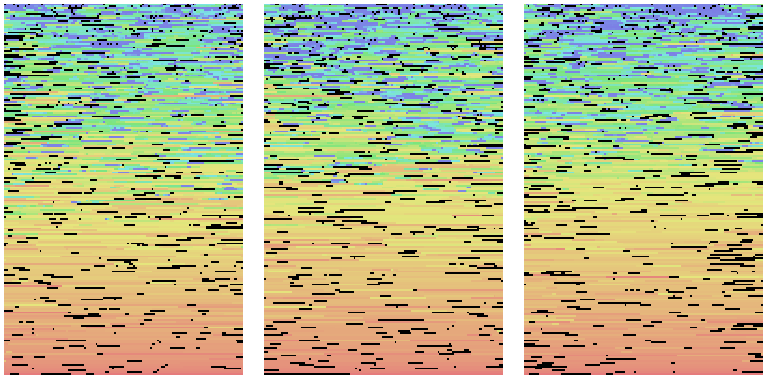
copybacks of data. As we can see in the figure, there are pages that are being copied back hundreds of times over. By using copyback blocks, we push a large number of pages to be copied back a few times only to predict how non-volatile the data is.

For the case of uniformly random workloads, there are no static sectors and each page or sector is as volatile as the other. Thus, the copyback blocks algorithm has no effect here as well.

Figures 5.34, 5.35 and 5.36 show the heat-maps at the end of the simulation. We see that without using separate copyback blocks, there is sedimentation and using separate copyback blocks removes it. The difference we can see between using a cache and not using a cache is that the dark pages are more concentrated to the right when there is no cache. This means that volatile pages are being updated more quickly and hence more

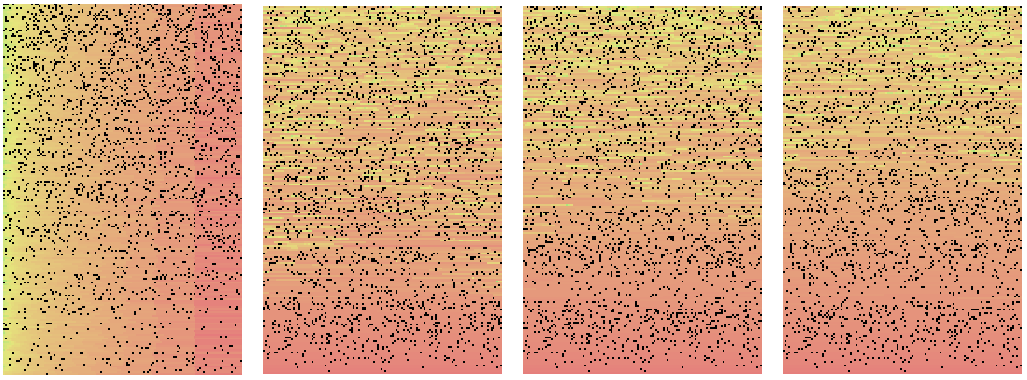


(a) None (b) 2B (c) 1 (d) 1,2

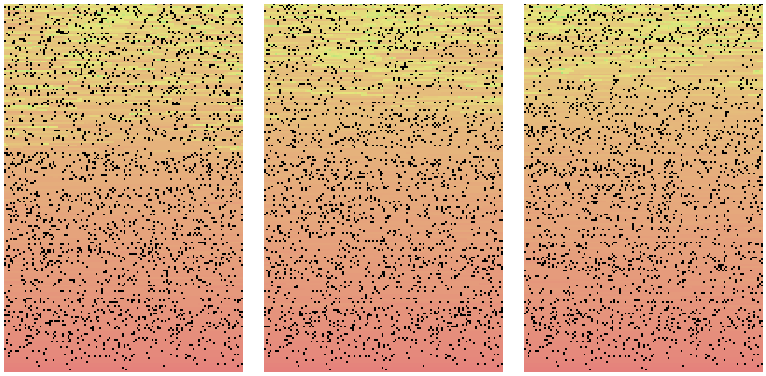


(e) 1,2,3 (f) 1,2,3,4 (g) 1,2,3,4,5

Figure 5.35: Heat-Maps: OLTP Trace Workload and 0.9 Size Usable (No Cache)



(a) None (b) 2B (c) 1 (d) 1,2



(e) 1,2,3 (f) 1,2,3,4 (g) 1,2,3,4,5

Figure 5.36: Heat-Maps: Unif Random Workload and 0.9 Size Usable (No Cache)

invalid pages at the high pages of the blocks.

For the uniformly random workload, there is some sedimentation and our algorithm does break up the sedimentation. However, the black dots are uniformly distributed over the heat map as any page can become invalid with equal probability. Thus, there is no effect on write amplification even though our algorithm does show a small de-sedimenting of the data. The loss of a block to copybacking and thus the small reduction in over-provisioning has more an effect than what is gained by de-sedimenting the data.

Thus, while cache does remove highly volatile sectors from the workload, it does not remove the problem of sedimentation and should be used for all cache sizes. However the smaller the cache, our separate copyback blocks algorithm is slightly more effective.

5.5.3 Performance Improvements and Wear Leveling

We are interested in the improvement in two aspects of flash memory: longevity and performance. By reducing write amplification, we reduce the total number of block erasures for the workload. This results in a longer lifetime of the flash memory due to lower average block erasures and a higher performance since less time is spent performing the slow block erasures.

5.5.3.1 Performance

When we reduce the number of copybacks, we reduce the number of system writes and this saves on the extra time required to do extra block erases to accommodate the extra writes caused by the excess write amplification. This savings in time is directly proportional to the reduction in write amplification because reduction in write amplification is just less writes in the system.

Taking the example of Zipf workloads and write amplifications of 7.2 for no copyback blocks and 5.5 for multiple copyback blocks, there is a savings of 2.2 extra writes per user write and an extra block erase at the end of 64 user writes.

5.5.3.2 Wear Distributions

The wear distribution, or the distribution of the number of times a block has been erased, is given in Figures 5.26 and 5.27 for the various workloads and over-provisioning factors. The mean wear of the block decreases proportionally to the decrease in write amplification as mean wear is simply total system writes divided by c (the number of pages per block).

In the case of the Zipf workload, as shown in Figure 5.26b for over-provisioning of 0.9 and Figure 5.26c for over-provisioning of 0.75, the wear distribution of the no copyback blocks is to the right of the multiple copyback blocks indicating a greater wear on the flash memory because of higher number of block erases. This follows directly from the difference in write amplification where using copyback blocks lowered the write amplification implying less copybacks and then less writes and thus, less block erases. For example, for over-provisioning factor of 0.9 usable, the write amplification decreases from 6.6 to 5.4 that results in an increased lifespan of the flash memory by a factor of around 22%.

In the case of the OLTP trace, for the over-provisioning of 0.9 as shown in Figure 5.27b, the wear curve for no copyback blocks is to the left of the copyback block algorithms. When the over-provisioning is 0.75m there is about 6% decrease in write amplification but the wear distribution curves are on top of each other. The wear distribution graph of the no copyback blocks is much narrower and taller than the wear distribution of the copyback blocks. We will look at the consequence of this in terms of wear leveling next.

5.5.3.3 Wear Leveling

Reducing write amplification reduces the mean wear on the blocks. However, using copyback blocks leads to a different problem. In all the wear distribution graphs for using copyback blocks, the graphs become wider and thus points to wear that is more uneven. Some blocks are heavily erased while some other blocks are only lightly erased leading to uneven wear.

The reason for uneven wear is that by separating out the static and dynamic data

into separate blocks, we end up with some blocks naturally updated less frequently than other blocks. When there are no copyback blocks and there is data sedimentation in the flash memory, all blocks have roughly equal amounts of static and dynamic data and each block is erased roughly the same amount. This leads to a very even wear in the flash memory. Now, when we separate out the data and put the static data on some blocks, what ends up happening is that the other blocks have more dynamic or hot data and are updated and erased more frequently. This leads to the situation of some blocks having low erase counts and some blocks having high erase counts, which is uneven wear.

This phenomenon is perfectly illustrated in Figure 5.27c. When we are not using copyback blocks, the wear distribution is narrow and tall. When using copyback blocks, the wear distribution becomes wide and spread out which indicates uneven wear. We will discuss a simple technique using a modified garbage collector to counter this in Section 5.5.3.4.

A common metric to measure the wear leveling is the wear variance, which is simply the variance of the wear distribution. It can be calculated as follows

$$\mathrm{Var}(W) = \frac{1}{n} \sum_{i=0}^{\infty} i^2 \, \eta(W_i) - \overline{W}$$

where W is the random variable that gives the erase count of a block and $\eta(W_i)$ is the number of blocks with erase count i. $\overline{W}$ is the average erase count or mean wear in the flash memory, n is the number of blocks and we iterate i from 0 (meaning no wear) to the maximum erase count in a block. The other way of denoting the range of i can be

$$i = 0 \ldots [\max \eta(W_i) > 0]$$

From the histogram and the above equation, we can easily calculate the wear variance.

Figures 5.37a and 5.37b shows the wear variances across different over-provisioning factors and with different copyback blocks algorithm. The apparent trend in the graph

is that using copyback blocks increases the wear variance and thus, creates wear that is more uneven. In many cases, the wear variance increased multiple folds of the original number and oven tenfold its original number in over-provisioning factor of $0.7 - 0.8$. We will next look at a modified garbage collector called sorted pool garbage collector that not only solves the problem but actually reduces the wear variance to below the original values.

5.5.3.4 Sorted Pool Garbage Collector

A simple method to counter uneven wear due to copyback blocks is to slightly modify the garbage collector. The garbage collector maintains a pool of erased blocks to use as future write or copyback blocks. We do this because erasing blocks is very slow and garbage-collecting blocks as they are needed would not be very efficient. The pool of erased blocks is usually kept as a queue with the first erased being used the first. We can modify the free blocks pool so that it is now stored as a sorted queue by its erase count. Then, copyback blocks will get erased and free blocks from the head of the sorted queue which has the smallest erase count and the writeblocks will get free blocks from the end of the queue which has the highest erase count. The idea is that when we keep static data in the block with low erase count and keep dynamic data in a block with high erase count, it will somewhat average out the erase counts and lead to more even wear. When we are using multiple copyback blocks, they will all get the new block from the beginning of the queue and we do not differentiate between the different copyback blocks here.

Note that we have not modified the garbage collector selection criteria. Therefore, all write amplification decreases will remain the same even with the sorted pool garbage collector. It is possible to modify the actual garbage collection criteria by making a more complicated method than just looking for the block with the least number of valid pages to make the wear more level but it will be a much more complicated algorithm. This method just requires the change in the free pool from a queue to a sorted queue. The

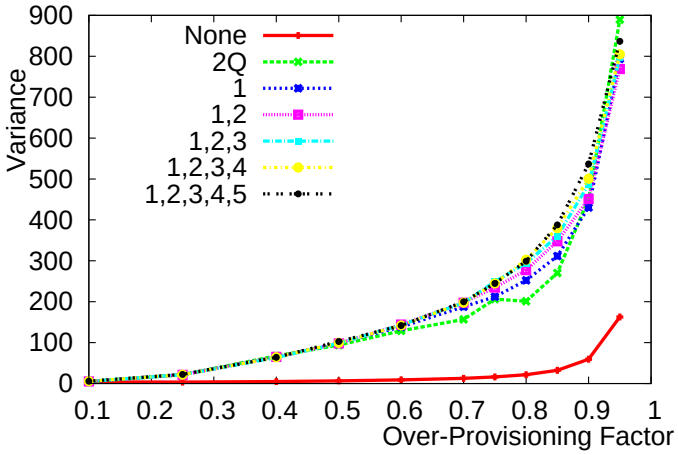
garbage collector's free block queue is 10 blocks in size in the simulation and so inserting into a sorted queue is very fast if implemented in a real system.

Figures 5.37c, 5.38a, 5.38b and 5.38a shows the wear distributions using the sorted pool garbage collector algorithm. They are the matching wear distribution curves from Figures 5.26b, 5.26c, 5.27b and 5.27c. Compared to the previous wear distribution curves, the sorted queue garbage collector has a taller and thinner wear distribution and hence, a more even wear. The average wear remains the same when we use the sorted queue garbage collector but the shape of the wear distribution considerably changes.

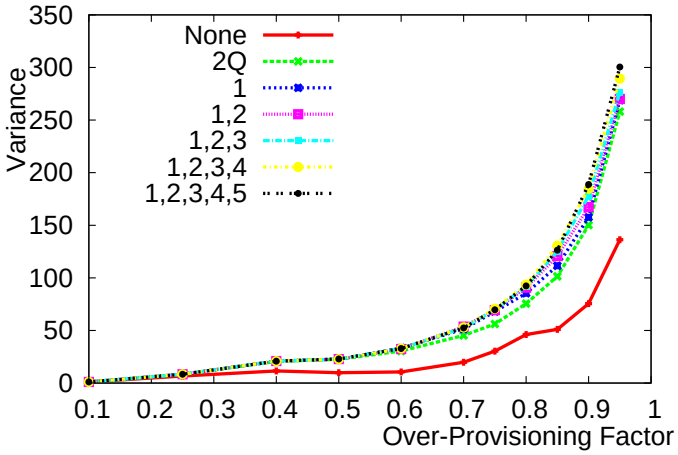
For the Zipf workloads, we see that the wear distribution curves are now very thin and tall and even make the non copyback-block wear distribution curve look short and wide. When not using the sorted pool garbage collector, the opposite was happening - the non copyback-block wear distribution curves looked thin and tall and the copyback blocks wear distribution short and wide. This highlights how much of an improvement in wear leveling is achieved by the sorted pool garbage collector. For the trace workload, similarly but to a lesser degree, the copyback blocks curves are now taller and thinner than the non-copyback block curves. This is the opposite of what we saw when not using a sorted pool garbage collector.

Figures 5.39a and 5.39b gives the wear variance using the sorted pool garbage collector for various over-provisioning amounts. We see that the wear variance values for the new copyback block algorithms are much lower than the non-copyback block algorithm. For the Zipf workload, it is well below and up to over-provisioning factor of 0.8, it very close to zero. For the OLTP trace workload, the wear variance curves are all closer together. Regardless of the workload, as we increased the number of copyback blocks, the wear variance became larger. Thus, just using a single copyback block gave us the lowest wear variances.

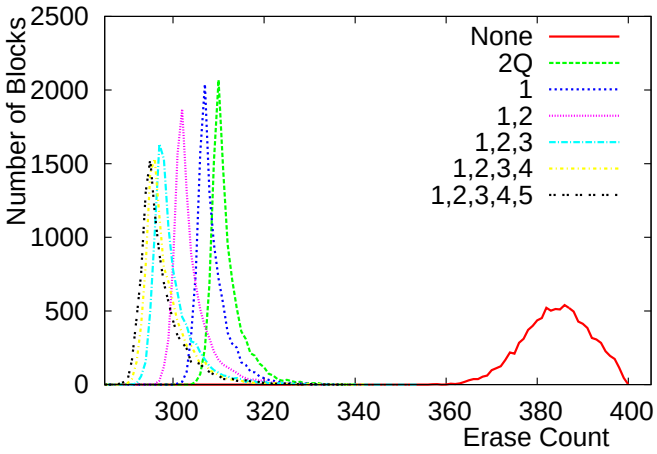
We omitted putting the wear variance curves from Figure 5.37 to Figure 5.39 because the differences are so vast that the difference between using the sorted pool garbage



(a) Zipf Workload, Wear Variance

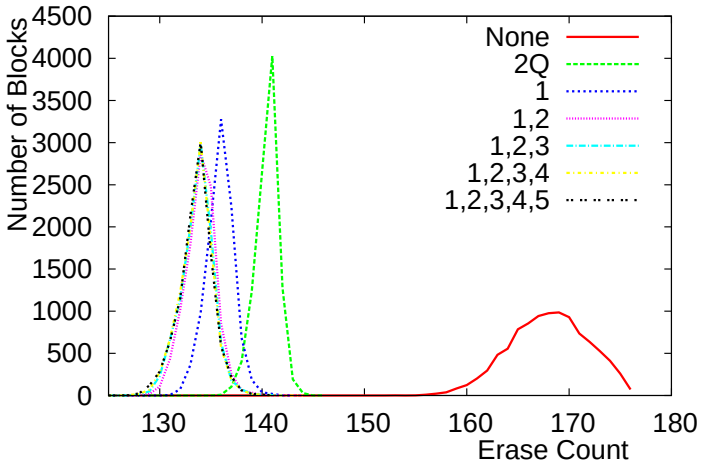


(b) OLTP Trace Workload, Wear Variance

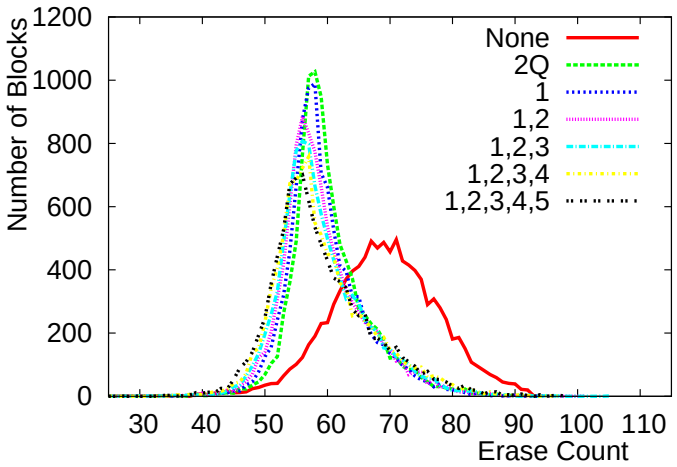


(c) Zipf Workload, 0.9 Size Usable, Wear Distribution

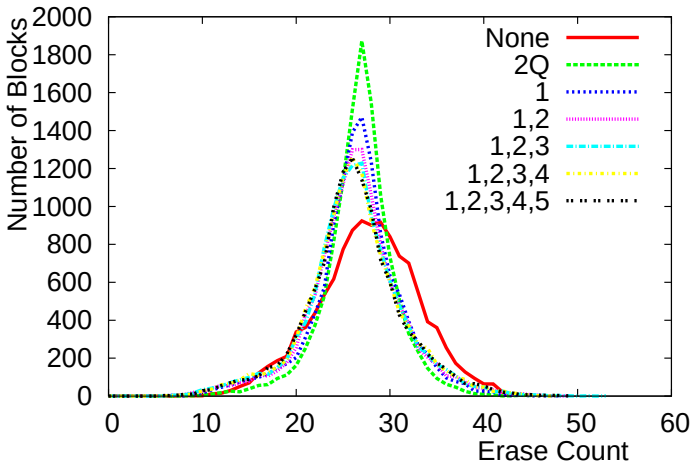
Figure 5.37: Wear Variance and and Wear Distributions Using Sorted Pool Garbage Collector



(a) Zipf Workload, 0.75 Size Usable, Wear Distribution

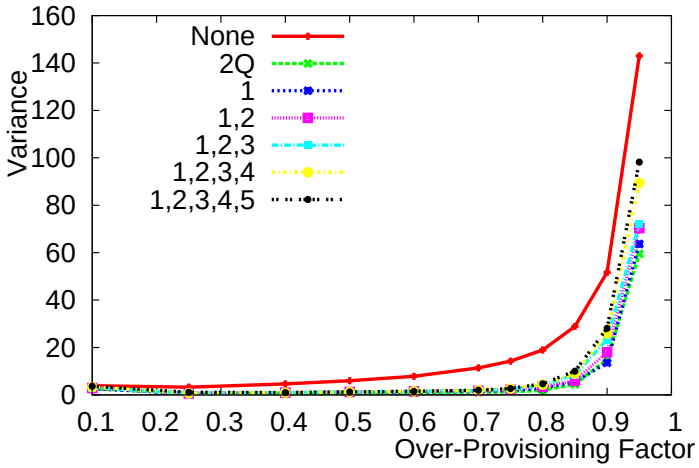


(b) OLTP Trace Workload, 0.9 Size Usable

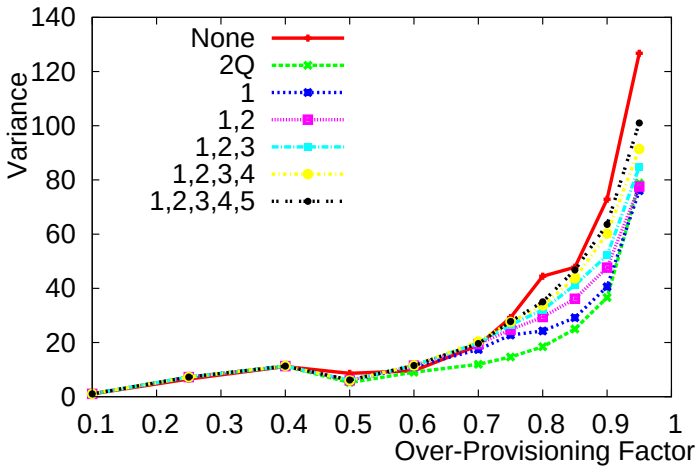


(c) OLTP Trace Workload, 0.75 Size Usable

Figure 5.38: Wear Distributions Using Sorted Pool Garbage Collector



(a) Zipf Workload



(b) OLTP Trace Workload

Figure 5.39: Wear Variance Using the Sorted Pool Garbage Collector

collector would be obscured.

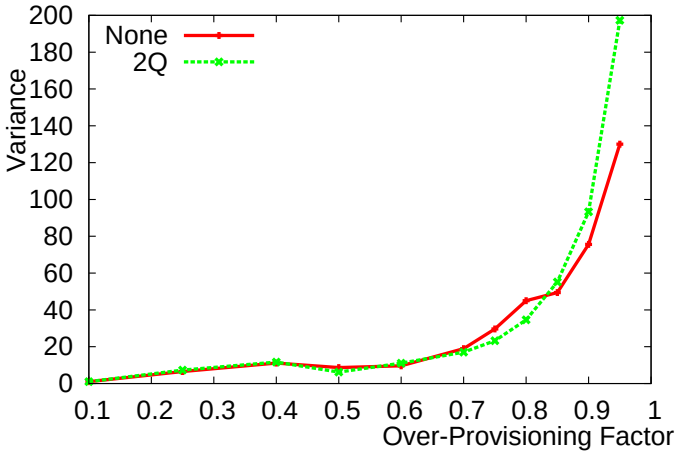
5.5.3.5 Size of the Garbage Collector Pool

In the above experiments, we have set the size of the garbage collector to 10. The next natural question is if the size of the garbage collector pool affects the wear leveling. When the pool is smaller, there is slightly more over-provisioning and thus less wear on the blocks. When the pool is larger, there are more blocks to choose from to use as the next write or copyback block. We next test with different sizes of pool sizes to see how this affects the wear leveling.

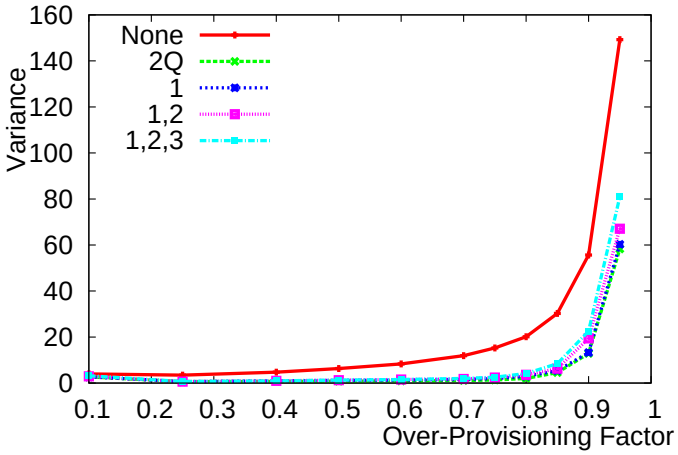
There is a minimal size of the garbage collector pool that is required that depends on the number of writeblocks and copyback blocks that we are using. The size of the pool of the garbage collector must be at least the total number of write and copyback blocks combined together. For example, if there were using a single writeblock and 4 copyback blocks for the 1,2,3 algorithm, we would need the size of the garbage collector pool to be at least 4. This is due the chance at some point during the simulation there is a chance that all the write and copyback blocks need clean blocks. For example, a write can fill up the writeblock and trigger a need for a new write block. This then triggers a need for a new block to be reclaimed for the pool. During copyback of the contents of the reclaimed block, all the copyback blocks could be filled up. In such a case, there needs to be at least as many blocks in the garbage collection free blocks pool as there are writeblocks and copyback blocks so that we don't run out of clean blocks.

Figures 5.39 and 5.40 shows the wear variance for the Zipf and OLTP trace workloads for various pool sizes. Note that the smaller pool sizes limit the number of copyback blocks we can use. For example, when the pool size is 2, we can only use a single copyback block and when the pool size is 5, we can only use 4 copyback blocks and thus, only limited to the 1,2,3 algorithm.

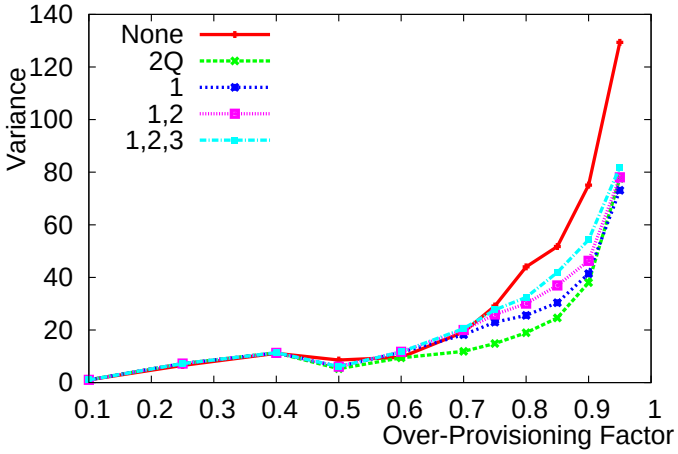
Next, we want to analyze the wear variance across pool sizes. This will give us an



(a) OLTP Trace Workload, Pool Size 2



(b) Zipf Workload, Pool Size 5



(c) OLTP Trace Workload, Pool Size 5

Figure 5.40: Wear Variance Using the Sorted Pool Garbage Collector For Various Pool Sizes

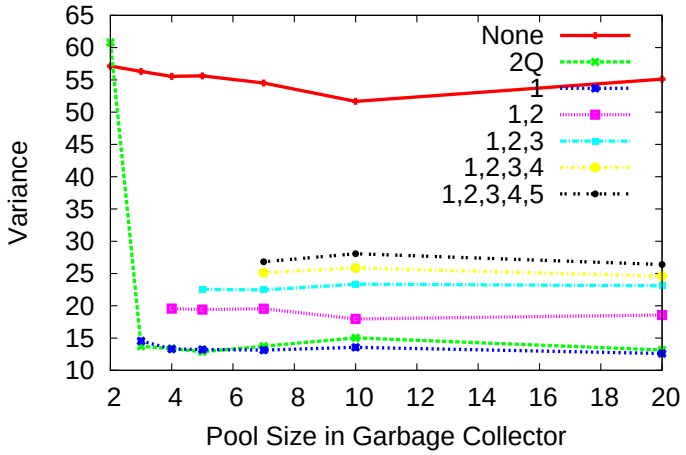
idea of how the wear variance is affected by the size of the sorted pool in the garbage collector. When the pool is larger, there is less over-provisioning and thus, higher write amplification and wear. When the pool is smaller, there is less choice in the free blocks pool for the writeblocks and copyback blocks to choose the next block from.

Figues 5.41 and 5.42 shows how the wear variances changes as we increase the number of blocks in the garbage collector pool. Note that the data point is not in the figure when the size of the pool is smaller than the total number of writeblocks and copyback blocks.

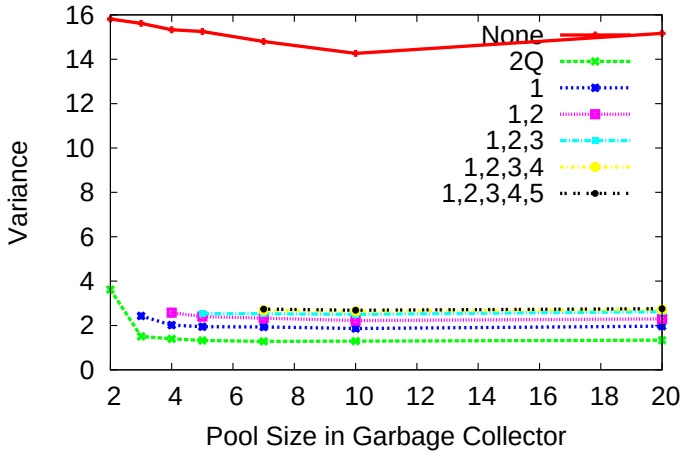
In general, for the algorithms using copyback blocks, as the size of the pool increases from 2, the wear variance falls until it doesn't change very much with pool size. For trace workloads and for lower over-provisioning factors, the pool size at which it stop decreasing is larger than for Zipf workloads and higher over-provisioning factors. Also interesting to note is that as pool size increases, the lower over-provisioning does increase write amplification but it does not affect the wear varaince significantly.

Thus, our pool size of 10 is adequate for most applications since most do not seem to decrease wear varaince for pool sizes greater than 10. For the trace workload in low over-provisiniong factor as 0.5 in Figure 5.42c, we would want to use a larger pool size since the wear variance is still decreasing with pool size.

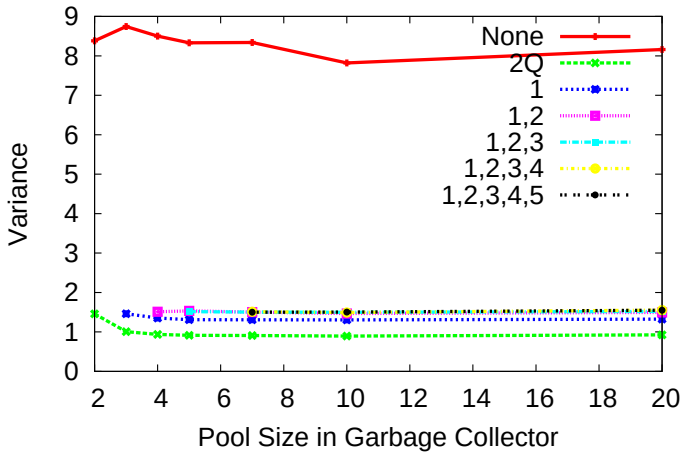
Also, note that the algorithms using larger number of copyback blocks have worse wear variances than the algorithm using just one writeblock and one copyback block.



(a) Over-Provisioning Factor 0.9

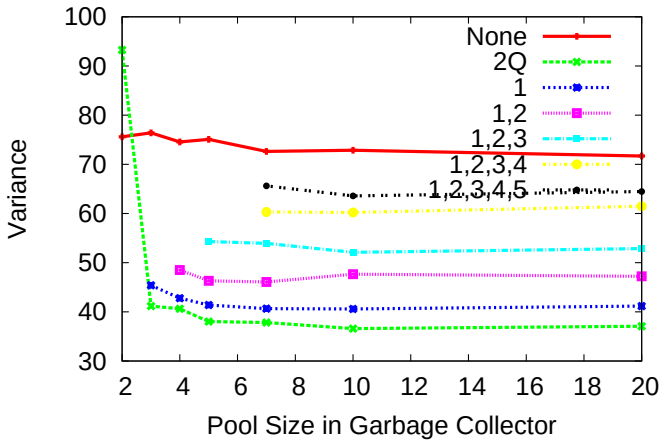


(b) Over-Provisioning Factor 0.75

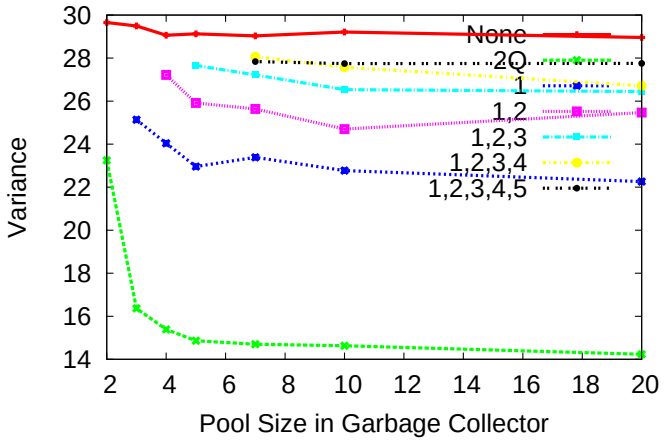


(c) Over-Provisioning Factor 0.5

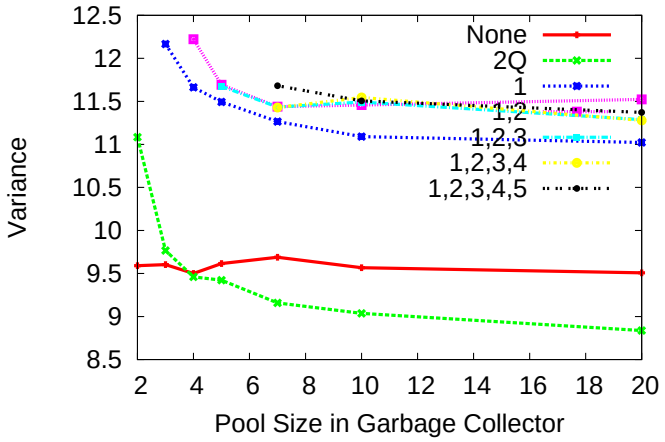
Figure 5.41: Wear Variance Using the Sorted Pool Garbage Collector For Various Pool Sizes and Algorithms for Zipf Workloads in Different Over-Provisioning Factors



(a) Over-Provisioning Factor 0.9



(b) Over-Provisioning Factor 0.75



(c) Over-Provisioning Factor 0.5

Figure 5.42: Wear Variance Using the Sorted Pool Garbage Collector For Various Pool Sizes and Algorithms for OLTP Trace Workloads

CHAPTER 6

SSD Simulator

For our experimental analysis, we developed a simplified event-based simulator called *SSDSim*. This is a heavily simplified simulator and is designed so that we can focus on the endurance and write amplification aspects of flash memory. The major simplification that we have used is that each event in the simulator is an arrival of an IO operation and there is no inherent sense of timings of the operations other than its order in the simulator. We can make this simplification because block wear and write amplification only depends on the order of the operations and not their timings.

A clock-accurate, IO timing based simulator would be more complete and accurate representation of a real SSD, but would be greatly more complicated. The analysis, insights and algorithms for block wear and write amplification that we develop for this simplified simulator can be easily modified to the IO timing based simulator or an actual SSD product and so in our study, the extra complexity is not needed. Thus, this model of the SSD that we use in our simulator serves our purposes very well for studying endurance and write amplification, and the ease of use from the model simplifications helps us focus in the properties that we are interested in.

6.1 Other Flash Memory and SSD Simulators

There have been SSD simulators written for flash memory and SSDs, some of them extensions to storage system simulators using magnetic hard disks and some written solely for flash memory. We chose not to extend a current publicly available and open source SSD simulator for our work because we are interested in a simplified simulator for looking at endurance and write amplification and extra complexity of the full featured simulator

would not give clearer results.

A flash memory SSD simulator as an extension to DiskSim [10] was developed to test different flash memory storage architectures. The simulator focused on the storage interface concurrency and the internal architecture and algorithms related to bus and interface operations among multiple flash dies. The variables in the simulator included number of flash storage dies, independent buses with different widths and speeds and different strategies for parallel data storage. All these variables were studied across different IO policies for scheduling, bus access and data bursts.

In [1], a simulator was developed for exploring data placement and workload management as well as parallelism and write ordering. Different flash memory die data organizations like ganging and interleaving were studied. The data placement was studied for wear leveling and an algorithm was given. The workload involved full workloads with timings and compared against popular benchmarking tools.

Another simulator [56] was developed with added parallelism capabilities and validated against actual SSD systems for accuracy. An object oriented open source SSD simulator is available in [82]. In [73, 74], a simulator called NANDFlashSim is presented that is a highly detailed simulator focusing on the timing models of NAND memory and NAND flash command architecture.

We wrote our own simulator because our goal was not to study parallelism and we wanted to employ our assumptions to simplify the architecture and simulation. While parallelism is a very important aspect of flash memory, it is in large regards orthogonal to our goal of studying endurance and write amplification. While a real SSD would deal with parallelism and endurance at the same time, we want to study a simplified model first that can then be adapted to a more complicated system rather than studying them together when they are independent in many aspects.

In the future, we want would like to incorporate our results to a simulator with parallelism capabilities and study how it affects designs and strategies for parallelism. Since

SSD systems must deal with issues of endurance, wear leveling and write amplification, higher up algorithms are affected by design choices below. Layout management and garbage collection will need to be redesigned with multiple dies present but the basic characteristics still stay the same and our results can be extended upwards easier than designing parallelism algorithms and then considering endurance and write amplification properties later on.

6.2 Components of the SSD Simulator

The simulator is an event driven simulator that is composed of various modules of the FTL to work together to emulate an SSD. The various modules of the simulator are

1. *IO controller*: translates IO requests to flash memory operations
2. *mapping table module*: handles the logical to physical maps
3. *layout manager*: determines where to place each new piece of data during write or copyback and invokes the garbage collector module when the free pages are needed
4. *state manager*: keeps track of the state of the data in the flash memory
5. *garbage collector*: determines which block to erase to reclaim free pages and performs copyback of valid data

Workloads are either simulated or from a trace but we do apply the workload structural reductions as described in Section 6.3. Thus, using these emulated modules that make up the simulated FTL, we perform an analysis by switching to different modules to see their effectiveness.

The simulator works via the following chain of events. At each time interval, a workload event is processed and it follows the following chain.

1. If the operation is delete, the sector is marked invalid in the mapping table.

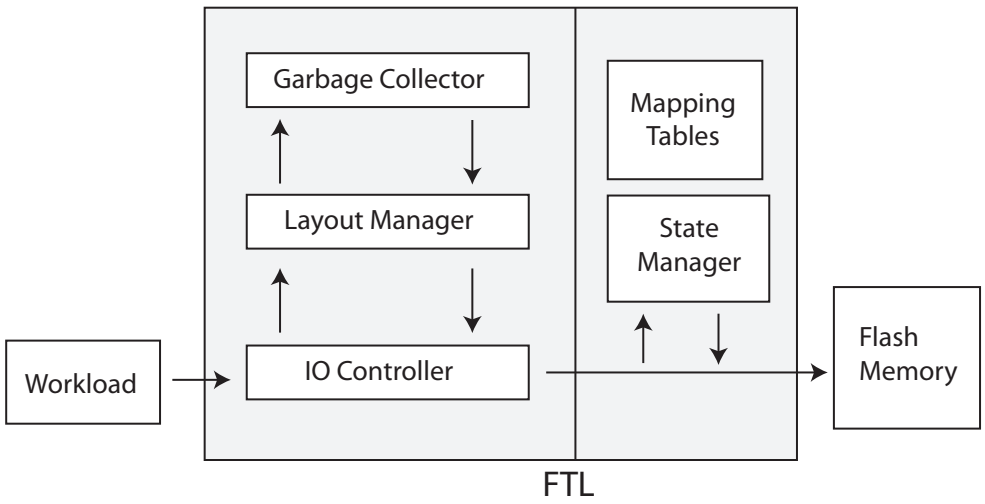


Figure 6.1: SSD Simulator Modules

2. If the operation is write, then the old version of the sector is marked invalid if it resides on the flash memory.

3. The layout manager is asked for the page location to write the new data. Here, the layout manager uses one of our layout management algorithms to determine on which page to write the data.

4. When the flash memory runs out of clean pages, the layout manager request the garbage collector to erase blocks to acquire clean pages to write to.

5. When the garbage collector is copying back pages, it also requests the layout manager where to locate the valid pages in the block to be copied-back. Here, our copyback layout management algorithms are used.

The modules and its relationships in the simulator are illustrated in Figure 6.1. In this way, the layout manager plays a central role in our simulator and we can effectively test our algorithms against write amplification from the simulation results by plugging in different layout manager algorithms.

We will look at the details of the various modules of the simulator next.

6.2.1 Layout Manager

The goal of the layout manager is decide where to place each piece of data. When new data comes in, the write layout manager decides where to place the new data. When the garbage collector is copying back old data, the copyback layout manager decides where to place the copied back data. In this way, the layout manager has full control of where every piece of data resides on flash memory and every data program to flash memory is preceded by a request to the layout manager to determine the location. We call each of the algorithm that decides the location for the data a layout management algorithm.

There is no location-based latency on flash memory and thus data can be stored anywhere without suffering any performance loss. What we gain from managing where data is stored is wear leveling and write amplification reduction. By putting data that is likely to be invalidated together, we can reduce write amplification and by matching the wear on the block to the type of data to be stored on it, we can perform wear leveling.

In our layout management algorithms, the way we determine the location to write or copyback is by maintaining a small array of write and copyback blocks. Thus, when a new write or copyback request comes in, it is written or copied back to one of the write and copyback blocks. By determining the type of data we are writing, it is sent to the appropriate writeblock among the array of writeblocks. The variations in our layout management algorithms are the number of write and copyback blocks and the algorithm we use to determine to which block the particular data goes.

6.2.2 State Manager

The state manager keeps track of metadata of the state of the flash memory. It keeps track of the following primary data:

- The mapping tables between the sectors and pages
- The number of valid pages in each block

- The erase count of each block

Since our simulator and algorithms are based on statistical properties, the state manager also keeps tracks of many other internal data. For example, it maintains the time since the block was erased or written to (rest time) for aiding the garbage collector or has internal data structures and functions for efficiency like quickly finding the block with the lowest number of valid pages or a block with the smallest score in some other metric.

6.2.3 Garbage Collector

The garbage collectors primary function is to find block to erase so that invalid pages can be reclaimed. The garbage collector maintains a pool of clean blocks where all the pages of the block are empty and is triggered when the blocks are used up from the pool. The size and ordering of the pool is configurable by the user.

The garbage collector finds the block to erase and maintains other parameters like the free pool size by various swappable garbage collector algorithms in the simulator. When it has determined which block to erase, it needs to copyback each valid page in the block and these data are written to other blocks determined by the layout manager.

6.3 Workloads

The workload is an integral part of the SSD simulator. There is a large variability in system endurance and performance depending on different workloads. The system will give behave drastically different between a random workload and a sequential one. Thus, in the simulator, we have a synthetic workload generator and a trace workload processor built in.

However, the workload used for our simulator is vastly different from a standard workload. First, there is no timings and the SSDSim workload is just a sequence of operations assuming that each operation occurs in one time unit. Second, there is no read operations

because read operations do not affect system endurance and write amplification.

There are only two types of operations, write and delete on a sector. The write operation is to write a sector data to a block page and the delete operation is to mark the sector as invalid. In trace data acquired from magnetic hard disk system, there is no delete operations because it unneeded for magnetic hard disk systems since old data is overwritten and there is no need to tell the disk when a data has been invalidated. Thus, in the simplest form, the workload can just be a sequence of sectors where we assume that each operation is a write operation.

Since flash memory can have gigabytes of storage, the workload can be millions of operations long and the trace files can be multi-gigabytes in size. Thus, synthetic workloads are generated and processed online so that there is no need to generate large synthetic workload trace files and the simulation runs faster.

6.3.1 Synthetic Workloads

A synthetic workload has an operation and a sector number following the operation. The operation can be either write or delete on the sector number following it. Thus, to create a synthetic workload we use the following parameters:

1. l, length of the workload
2. n, number of sectors
3. r, ratio of writes and deletes
4. distribution of the workload

The length of the workload is how many operations are in the workload. Since the length of the workload can be billions of operations long, a convenient way to denoting the length of the workload is the average number of times a sector is updated. Therefore,

the length of the workload is then just the number of sectors multiplied by the average update. This gives a institutive view of the length of the workload.

However, the length of the workload should be such that the property that we are interested converges to a stable value at the end of the workload. When we use our simulator in Chapter 5 for write amplification, we find the length of the workload where the write amplification reaches a stable value.

The number of sectors determines how many updateable sectors are present for that the workload to issues operations. This is usually different from the number of pages present in the flash memory to provide over-provisioning. If the workload is for a smaller number of sectors than the number of available pages in the flash memory, then the extra pages can be used for improve the flash memory performance.

The factor r is the ratio of writes and deletes in the workload, e.g., for $r = 0.7$, 70% of the operations are writes and 30% deletes. For each *write* and *delete* operation, a sector is chosen from n possible sectors and the selection is done following one of the pre-defined distribution called the *workload distribution* which we describe below. This leads to the workload being a Markov chain and r behaving as another form of over-provisioning. If simulations where we have over-provisioning by making the number of workload sectors smaller than the number of available pages, r can be set to 1 meaning no delete operations. If we want to also model delete operations, we can set the number of sectors to the number of available pages.

6.3.1.1 Workload as a Markov Process

We allow the delete operation to be performed for a sector that is not present in the disk or is invalid, in which case, the operation is ignored. The purpose of this addition is for modeling simplification; if the delete were forced occur only on valid blocks, each delete operation would be dependent on the series of previous write and delete operations. However, this model of the workload is still a valid representation of the workload as this

workload is equivalent to a workload with all the deletes to invalid sectors removed.

From our definition, r is the fraction of operations that are writes and thus, $1 - r$ fraction are delete operations. Let us consider a two state Markov process where a block is either valid or invalid. After an erase operation to the block, it is invalid and after a write operation, the block is valid, regardless of whether it was valid or invalid before. This gives us the following transition probability matrix P for our Markov chain

$$P = \begin{bmatrix} r & 1-r \\ r & 1-r \end{bmatrix}$$

whose steady state probability is

$$\begin{bmatrix} r & 1-r \end{bmatrix}$$

Thus, after the flash memory has been in operation for some time and reaches a steady state, we can expect r fraction of the blocks to be valid and $1 - r$ fraction of the blocks to be invalid. In other words, after reaching the steady state, the number of writes equals the number of actual deletes.

Here, r can be interpreted as the fraction of the disk capacity that is advertised as being available for writing while $1 - r$ is the fraction serving as the over-provisioning of the disk.

6.3.1.2 Workload Distributions

We assume that each operation is independent and identically distributed and hence, when an operation is generated it does not look at or is affected by any of the previous generated operations in the workload. This obviously means no spatial or temporal locality as well as no sequentiality.

We also assume that the distribution is a decreasing distribution where the first the sector has the highest probability and each higher sector afterwards has equal or less

probability. This is a fair assumption because the sector numbers can be sorted and rearranged to have the above property since all the sectors are mapped internally and the sector numbers can be reassigned to any label.

We use the following random variables to model the workload

1. O_t: the operation at time t which is either a *write* or *delete*.
2. C_t: the sector which the operation O_t will be performed on.

Here, by our assumption, denoting the probability of writes and deletes, for $1 \leq t \leq l$,

$$P(O_t = write) = r$$

and

$$P(O_t = delete) = 1 - r$$

Also, to illustrate that there are n sectors, all the operations must be in one of the n sectors and so,

$$P(1 \leq C_t \leq n) = 1$$

Figure 6.2 shows the workloads that we describe below. Note, that for drawing the figure we use $n = 1000$ though we use much larger values of n for simulations. We use this value for n in the figure to show the rough shape of the distributions. Additionally, we have removed the first few highest probability sectors from the graph so that the curves can be seen clearly as the low sectors have magnitudes higher probability than the ones following it.

We choose workload from the following three distributions:

1. **Uniformly Random:** The probability that a sector is chosen is identical for all sectors. We define *uniformly random workload* as $\forall i, j \in \{1, 2, \ldots, n\}$

$$P(C_t = i) = P(C_t = j) = \frac{1}{n}$$

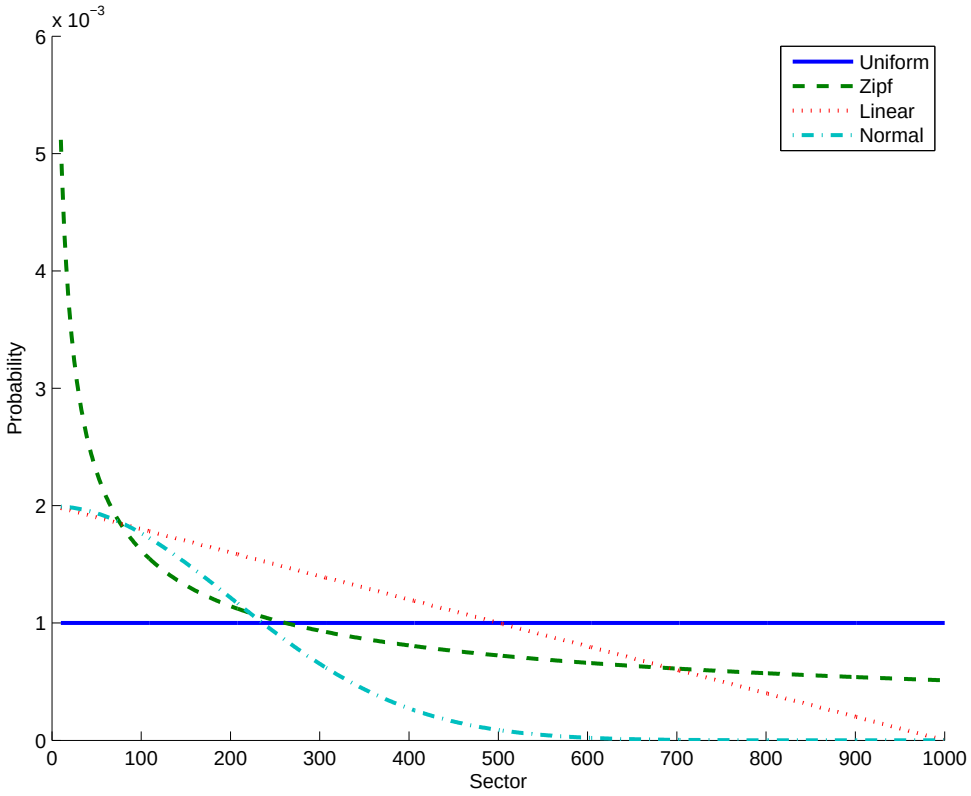


Figure 6.2: Workload Distributions for $n = 1000$

2. **Zipf:** Zipf distribution is a power law probability distribution where some sectors are likely to be accessed very frequently while others are almost static. This gives us a good model of workloads that produce dynamic and static data. Thus, the *Zipf workload distribution* is defined as follows with $\alpha = 1$:

$$P(C_t = i) = \frac{\frac{1}{i^\alpha}}{\sum_{k=1}^{m} \frac{1}{k^\alpha}}$$

The larger the value of α, the more concentrated to a fewer sectors the probabilities are.

3. **Linear:** The *linear workload distribution* serves as an intermediate distribution from the heavy tailed Zipf workload distribution and the constant uniform workload distribution and is defined as

$$P(C_t = i) = \frac{i}{\sum_{k=1}^{m} k}$$

The Zipf workload distribution has a large tail that is uniform-like, i.e., the tail probabilities are all very close to 0 that has a similar behavior to the uniform workload distribution. The linear workload distribution has no uniform-like regions and thus also serves to highlight characteristics that the uniform and Zipf workload distribution do not exhibit.

4. **Other:** We can take any probability distribution and sort it by probability to turn it into a workload distribution. For example, the normal probability distribution can be sorted from the highest to lowest probability (or just right descending half of the probability used) as in Figure 6.2. In most cases, we are interested in the disparity of probabilities between highly updated hot sectors and infrequently updated cold sectors and thus, the uniformly random and the Zipf workload distributions provide for that.

6.3.1.3 Limitations of the Synthetic Workload Models

The first limitation of our synthetic workload is that it does not model spatial or temporal locality. When a sector is updated, there is a higher probability that another sector close by will be updated (spatial locality) and also a higher probability that the updated sector will be updated again in the near future (temporal locality). The reason we have not modeled synthetic workloads with locality properties in our simulator is that there are no current models of quantification of locality in workloads and a useful synthetic workload model would require an in-depth survey of workloads and the locality they present and an acceptable model of it.

The second limitation is the lack of sequentiality modeling. In many cases, flash memory is used to store large files and when the file is deleted, data is inserted or file is changed, it triggers a large chain of updates that occur in a sequence. This can dramatically change the behavior of layout management and garbage collection algorithms and there are many designs and algorithms for sequentiality detection in workloads.

The trace workload serves the purpose of filling the gap in the synthetic workloads that we have mentioned. It does have spatial and temporal locality as well as sequential updates. Thus, we can use synthetic workloads for analysis of distribution-based variations and the trace workloads for the variations based on locality and sequentiality properties.

6.3.2 Trace Workload

The traces that have been recorded and made available from workloads recorded in storage systems have been all from systems using magnetic hard disks. As such, all writes are sector sized, there are no delete operations notifying the disk that data has been made invalid and that the disk sector size is much smaller than the flash memory page size.

To make the available trace workload suitable for use with our simulator, we remove all the read operations, remove the timing information and end up with essentially a long list

of sector updates. The number of sectors was assumed to be the largest sector accessed in the workload.

Of the many candidate trace workloads we looked at, most of the workloads were read heavy and after removing all the reads and extra information from the multi-gigabyte trace workloads, the number of remaining writes were too few to be useful for our simulation. The trace workload we use for our simulation is an OLTP trace workload that has a large number of write operations. The OLTP trace workload still uses a fraction of sectors and the average number of times each sector is written is still a fraction of what we use for synthetic workloads.

For workload traces from real systems, we use the financial OLTP traces from Storage Performance Council [117]. For the OLTP trace, $l = 30,517,401$.

6.3.3 Workload Visualization

Another way to visualize the workload distribution is to look at the sectors accessed against time as in Figures 6.3, 6.4 and 6.5 for the Zipf, uniformly random and trace workloads. Since the workloads are tens of millions operations long, we can only look at a small segment of the workload.

For the Zipf workload, we can see that the low sectors are heavily accessed and the higher sectors are accessed infrequently. For the random workloads, all sectors are possible to be updated with equal frequency and thus, the access patterns are random in the graph. The trace workload shows spatial and temporal locality as well sequentiality that is not present in the other workloads. While the synthetic workloads stay relatively the same throughout the length of the workload, the trace workload writes change slightly over time as the writes focus on different sectors over time. Another snapshot of the trace workload is given in Figure 6.6 starting at a different time and looks similar to Figure 6.5 but the locality is in different place and the sequences start at different points.

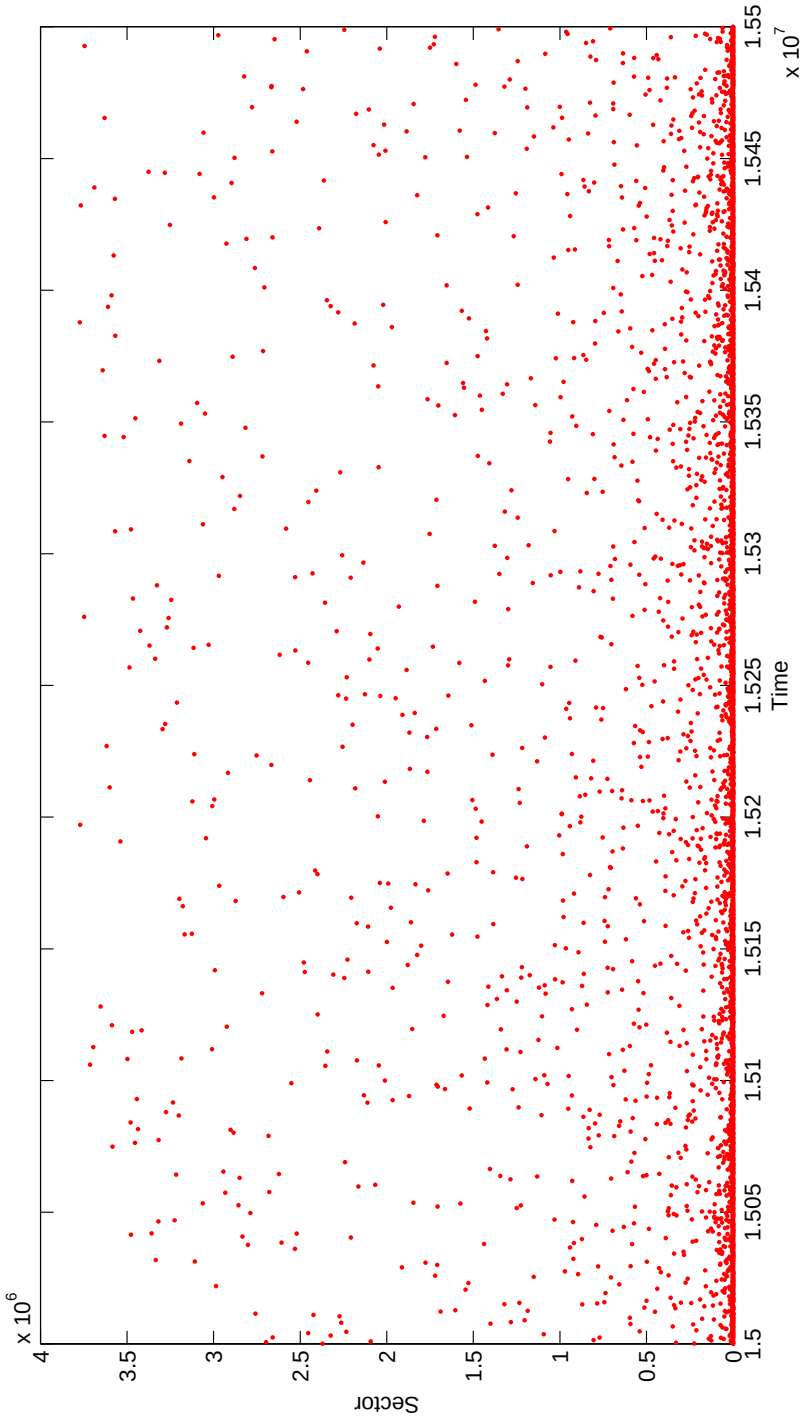


Figure 6.3: Zipf Workload

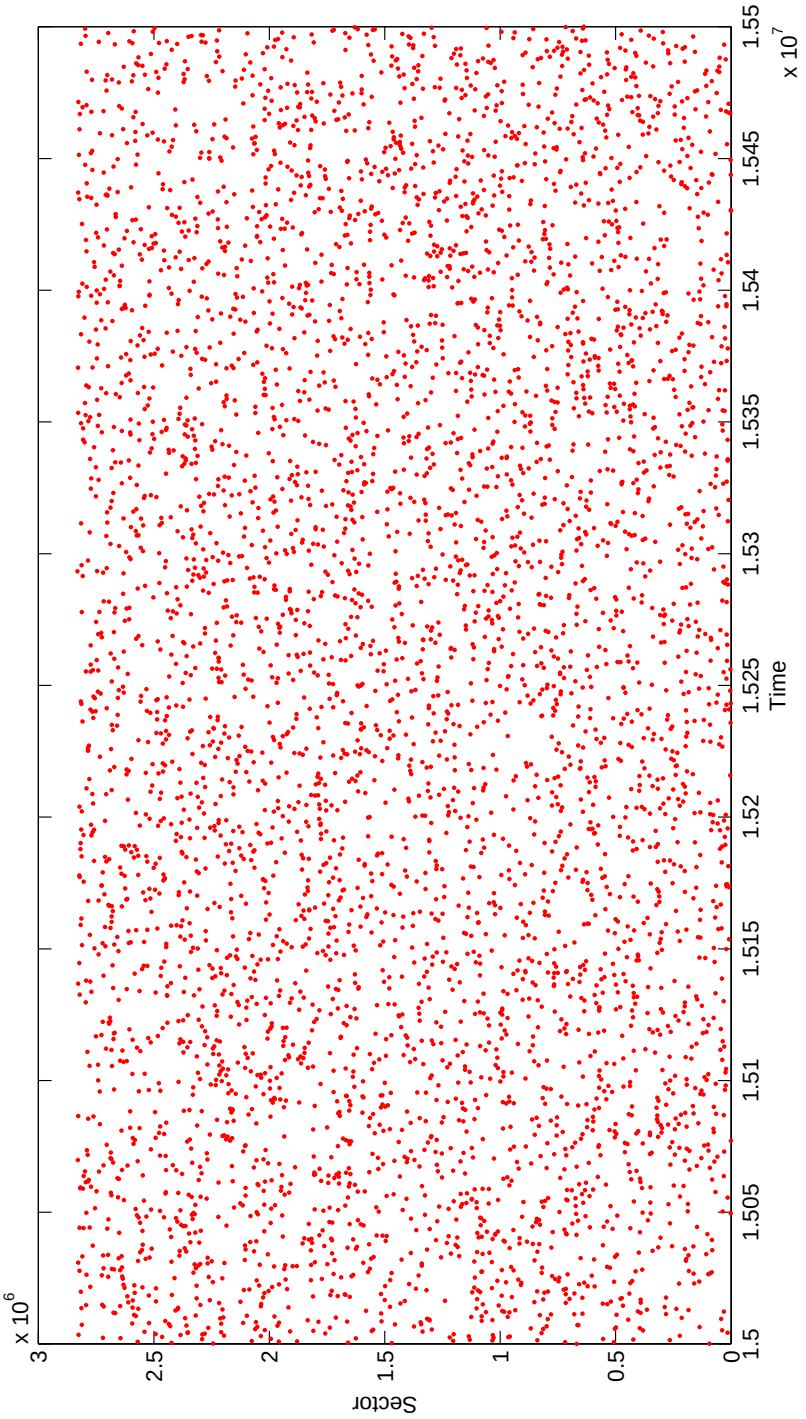


Figure 6.4: Uniformly Random Workload

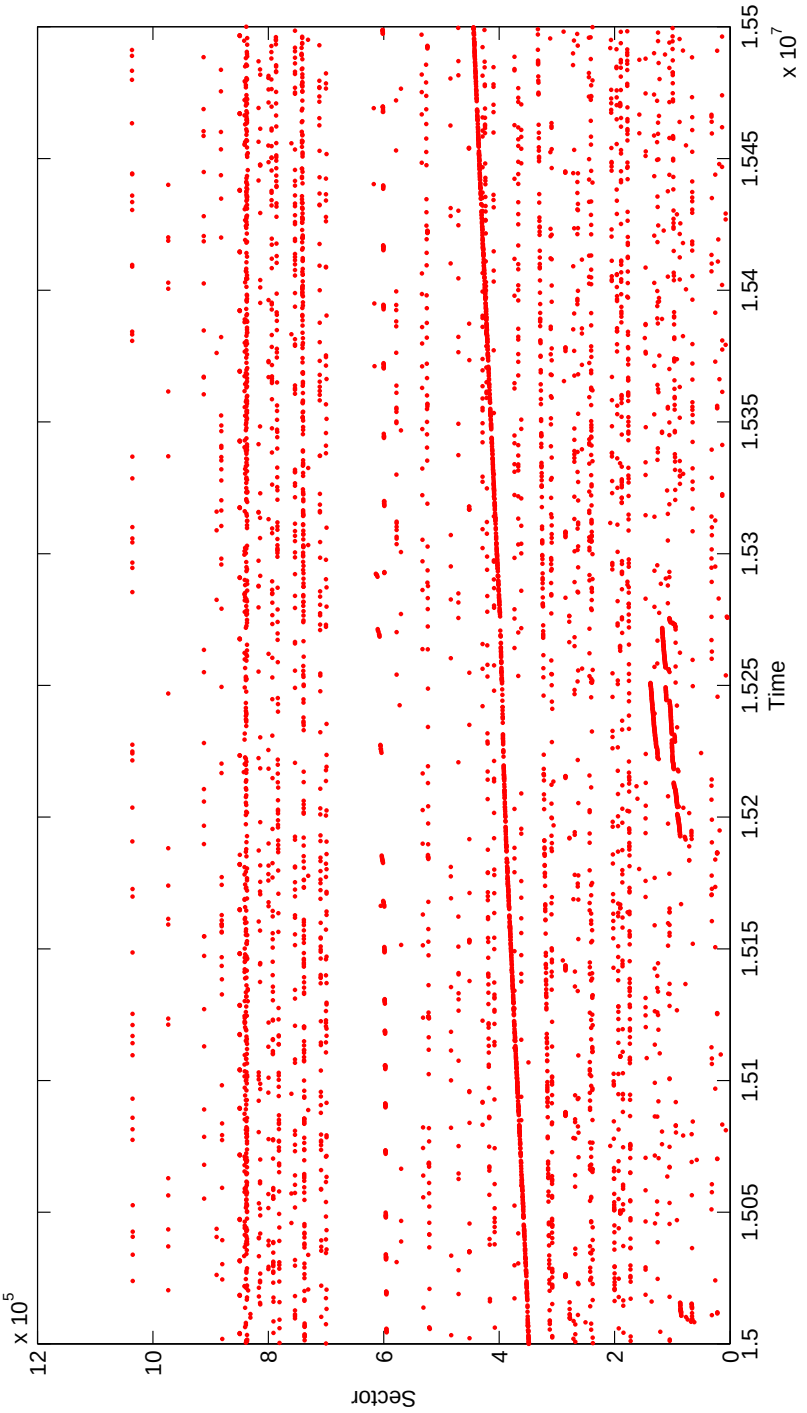


Figure 6.5: OLTP Trace Workload

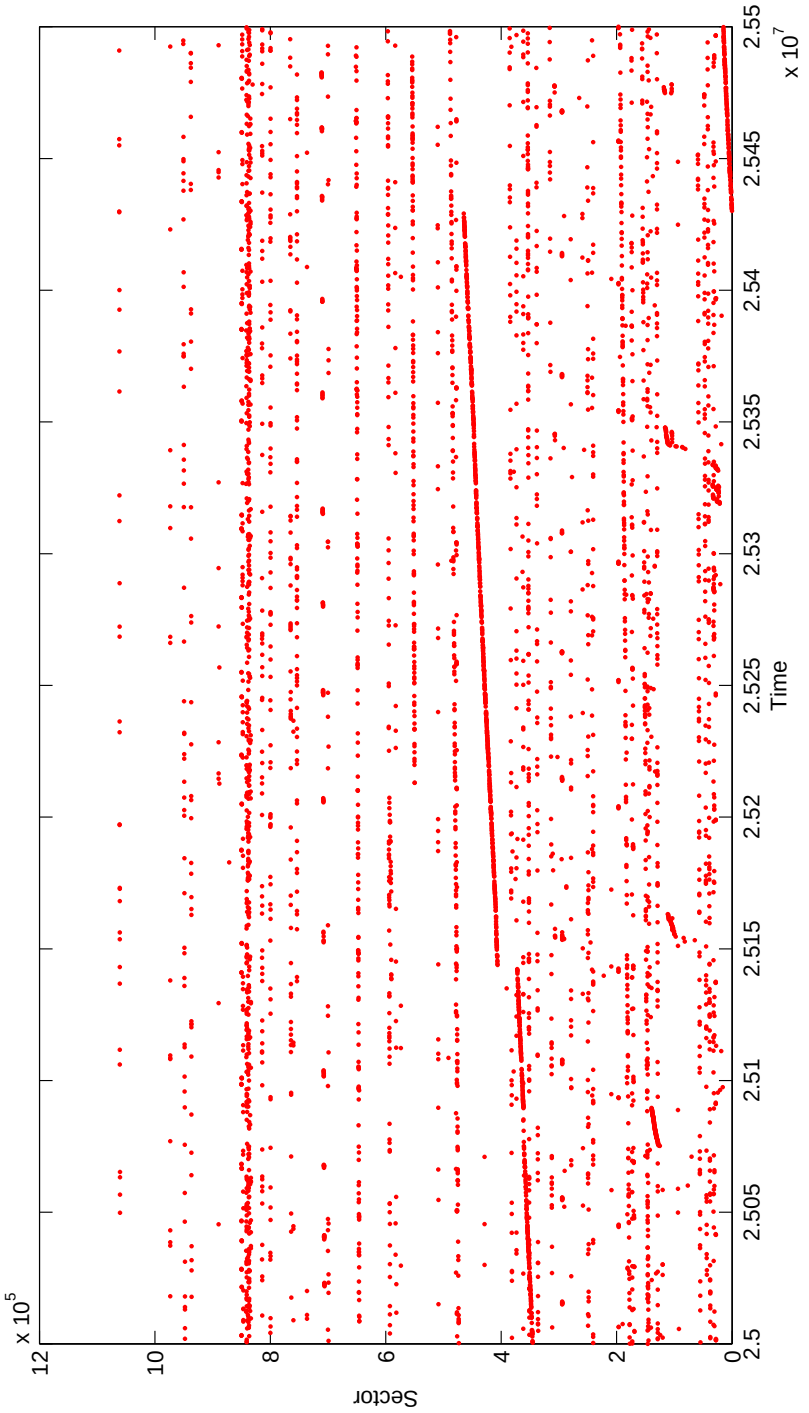


Figure 6.6: OLTP Trace Workload Starting At a Different Point In Time

CHAPTER 7

Wear Level Modeling

A unique characteristic of flash memory is that it is only possible to program a flash memory cell a limited number of times before it cannot hold data reliably and because of that, wear leveling has to be performed by the SSD. A flash memory cell can be erased only 10,000 - 1,000,000 times before it wears out [29]. Since some areas of the disks are written to more often than others, it could result in uneven wear or worn out areas thus, leading to a reduced functional lifespan. This is combated with wear leveling where data is written out of place, i.e. when data is rewritten, it is not written to the location where the data was previously stored but is written in a new location determined by a wear leveling strategy. In this way, SSDs make the limited write endurance deficiency of flash memory invisible to the storage system and manage it internally through wear leveling algorithms.

This limited write endurance of flash memory is a major concern for system integrators and considerable resources are used by the SSD to perform wear leveling. Without wear leveling, an application repeatedly updating data on a sector can easily wear out a block that would require replacement of the SSD in a short span of time. To perform wear leveling, most SSDs are engineered around the design of maintaining logical to physical sector maps to provide out of place updates such that each update to the sector is written to a different location in the flash memory. Each IO operation has to be translated through an in-memory map and each update requires an algorithm to select where the update will be placed. Thus, wear leveling plays a central role in SSDs and finding methods to improve wear leveling and make it more efficient is an important topic.

Wear leveling algorithms are decades old [43] but as modern SSDs are designed with

higher capacities, larger buffers and with it more controller overhead, there are numerous design decisions that need to be made for the SSD's wear leveling algorithms. The need for theoretical models have been highlighted in [44] as open problems and the authors have presented their models in [5]. But so far the effectiveness of wear leveling algorithms is simply evaluated through simulations or very rough empirical measurements and there is a lack of systemic and quantitative mathematical analysis. Furthermore, simulation has shown that the effectiveness of wear leveling is determined by multiple factors, such as the configuration of the SSD, the wear leveling strategy and the workload distribution [1]. Thus, when designing an optimal wear leveling system for an SSD, there is a need to model, characterize and quantify the wear leveling process for various workloads and how they affect wear leveling algorithms and SSD configurations.

In this chapter, we present a rigorous mathematical model for wear leveling for whole flash blocks (i.e., a block has only one write unit). We limit our model to whole blocks because it enables the analysis of wear leveling and how it is influenced by simple workload distributions and elementary wear leveling strategies without the analysis being affected by partial block flash memory characteristics like write amplification [53] and cleaning efficiency [79, 1]. Our model can easily be extended to partial block wear leveling using modeling techniques we briefly outline in this chapter but leave the details for future work. On the other hand, it is also of immediate practical importance for large SSDs which use whole blocks as a write unit for parts of the flash memory to keep controller overhead down and improve performance [8], a technique which is especially useful for sequential portions of the workload [84]. Thus, even though we limit our model here to whole block wear leveling while flash memory is capable of partial block write units, it still has significant theoretical and practical importance and serves as a precursor to the more complicated partial block wear leveling.

As far as we know this is the first work at constructing a general mathematical framework for quantitatively studying wear leveling. Defining *erase count* as the number of

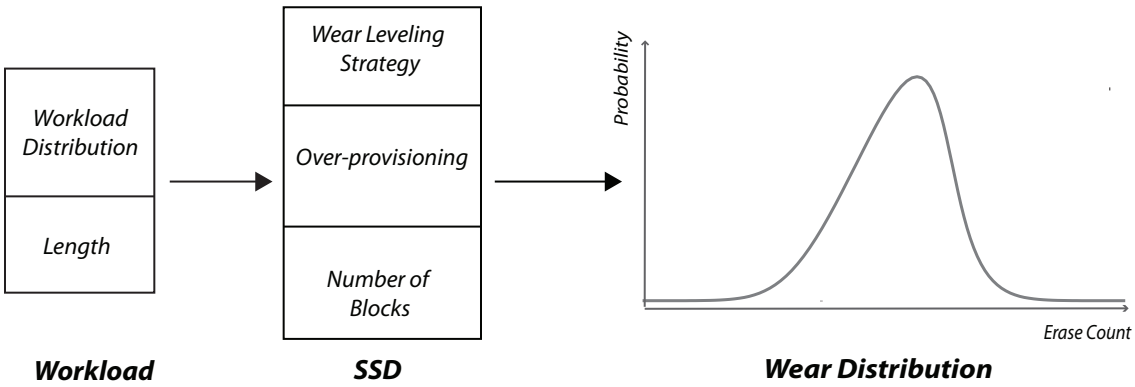


Figure 7.1: Main idea: How does wear distribution vary with parameters?

times a block has been erased and the *wear distribution* as the probability density function of erase counts for a block at a given time, our approach to build a framework for studying how the wear distribution varies with changes in SSD, flash memory and workload parameters (Figure 7.1). In our model, we take the erase count of a block as a random variable and study the wear distribution in a probabilistic framework and how it is influenced by other random variables like the workload, the SSD framework and the wear leveling algorithms. This gives us equations relating wear leveling effectiveness (the variance of the wear distribution) to the various elements and architecture of the SSD system. We use variance of the wear distribution as the measure to quantify the effectiveness of the wear leveling system for the given parameters. Thus, the framework provides a powerful tool to study wear leveling algorithms numerically and to calculate optimal SSD designs, and is also useful to explain flash memory observations and characteristics that have been reported, like the effects of over-provisioning [5] and the effects of number of blocks in the flash memory [107].

In section 7.1, we give a brief overview of the related work on SSD modeling, theoretical issues and studies on practical algorithms of wear leveling. In section 7.2, we describe the components of our model: the flash memory, workloads and wear leveling strategies. In section 7.3, we present a modeling framework and from it derive recursive equations

to generate the wear distributions. In section 7.4, we analyze wear distribution and the effectiveness of wear leveling under various SSD configurations and parameters, and remark on interesting implications and observations of our analysis that are especially useful for SSD system designers.

7.1 Related Work

An approach to solving the whole block wear leveling problem [44] was presented in [5], where it was found that using a randomized algorithm to select the next block to erase was optimal, but was based on an overly simplified model of flash memory. Given n blocks with a wear limit of H (i.e., a block can be erased H times before becoming unusable), the authors presented an optimal but offline algorithm that allows for greater or equal than $n(H-1)$ writes before a block in the flash memory reaches its wear limit. Furthermore, an online randomized algorithm, where given a write to sector s which is stored in block i, either with probability p randomly selects another block j to write s onto and writes the contents of j to i, or with probability $1-p$ writes sector s back to block i, was shown to get close to optimal for certain large H and small p. The limitation of this approach is the lack of the ability to mark blocks as invalid and do garbage collection since writing a sector to the block it was stored in is slow, requiring an erase and write to the block to complete a write request. Here, we use a more detailed model of flash memory SSDs with over-provisioning, block mapping and more complex workloads.

Orthogonal to wear leveling, there have been many works on the nature of flash cell wear, practical wear leveling algorithms proposed (mostly for partial block wear leveling) and analytical models of flash memory for exploring flash memory based SSD phenomenon like write amplification. Models of flash memory have been proposed for performance evaluation through estimating and calculating write amplification [53, 12] and its limits [52] and the impact of garbage collection on it [58]. Research in practical wear leveling al-

gorithms include reverse engineering algorithms in commercial SSDs [8], reducing wear leveling operational cost by summarizing block information [72] and designs for moving rarely updated static data proactively [21], through dual [18] or multiple [62] queues. Studies on flash endurance have put forward the notion that current estimates are inaccurate [29] and that current models of wear endurance are too simplistic [123] where recovery periods could increase endurance [96]. Additionally, there have been studies on the nature of endurance failure through bit error rates, program latency and program/read disturbs for patterns of block writes [47]. Studies on workloads and how it affects the SSD system include metrics [9], experimental results on real life workloads [103], real time garbage collection [19] and performance [23]. Thus, our work on wear leveling covers an important segment in the vibrant and rich field of flash memory modeling and algorithms.

7.2 Wear Leveling Model Components

7.2.1 Flash Memory and SSDs

Flash memory bits are stored in memory cells which are then organized into pages and blocks. Flash cells store bits by trapping electrons in a floating gate transistor called programming which can then be read by sensing the miniscule electric field that the trapped electrons generate. To rewrite data, the trapped electrons must be freed and is called erasing but erasing is done in a large area of flash memory called a block. To allow for as many memory cells as possible in the flash memory die while minimizing the other supporting circuitry, 30-60 thousand memory cells are organized as pages such that reads and writes occur at the granularity of a page. Furthermore, 32-64 pages are organized as a block, the smallest erase unit. In this way, cells, pages and block make up the fundamental building blocks of flash memory storage technology.

We model flash memory to reflect the fact that the SSD interface exports a linear array of sectors while internally, flash memory is an array of memory cells grouped together by

erase units. In our model, flash memory is an array of blocks where a block is the smallest erase unit and the SSD interface is an array of sectors where a sector the smallest read and write unit used by the file system. A block hold one or more sectors, with each sector stored in a page in a block. Here, we assume that the size of the sector equals the size of the block; i.e., *whole block wear leveling* where the write and erase units are the same size. This is a simpler modeling problem than *partial block wear leveling* problem where when a block can hold several sectors [44]. Let m be the number of blocks in the flash memory and n be the number of sectors that the SSD advertises as its capacity. Here $n \leq m$. For our purposes, the size of the block or sector does not play a part in our results and thus, we do not define it.

We assume that the SSD supports the following three sector operations: read, write and delete (TRIM [111]) which is then translated to the following four block operations in the flash memory: read, program, erase and delete. Although the delete or TRIM operation is not needed for the correct functioning of a storage system, it has been shown to greatly improve SSD performance [27] and is supported by most modern operating systems. An SSD read operation is translated to a read in the flash memory, an SSD write operation is translated to an erase and then program in the flash memory, and a delete or TRIM operation is used by the controller to mark the relevant block as invalid so it can be erased later. As the erase operation is an order of magnitude slower than the program operation, translating each write operation to the erase and program would take a large performance hit. Most SSDs implement a garbage collector that maintains a pool of erased blocks such that writes can be translated to programs and erases are done in the background. However, translating write to erase and program does not affect wear leveling and is simpler to model and thus, we choose this approach.

Blocks can be in one of the three states: valid, invalid or clean. After a block is erased and before it is programmed, it is in the clean state. When data is first programmed into a block, it is valid and after the programming but before the block is erased, it can either

be valid or invalid depending on if the programmed data is current or not. When a delete or write operation for the sector that the block stores the data of is performed, the data then becomes invalid. An invalid block can be safely erased without the fear of losing data. For each block, if an accurate count of the number of times it has been erased is maintained, it is called the *erase count* of the block.

7.2.2 Workload

A workload is a sequence of read, write and delete operations to the disk. While modeling workloads, we make the following simplifying assumptions as they do not affect whole block wear leveling:

1. As read operations have no effect on wear, we simplify workload as a sequence of writes and deletes.

2. We do not model spatial locality or sequentiality.

3. We do not model the effect of the buffer and assume that every write and delete is directly translated to the corresponding flash memory operations.

Let l be the length of the workload that is composed of *write* and *delete* operations, each operation specifying a target sector. Let r be the ratio of writes to the sum of writes and deletes., e.g., for $r = 0.7$, 70% of the operations are writes and 30% deletes. For each *write* and *delete*, a sector is chosen from 1 to n, following one of the pre-defined distribution called the *workload distribution* which we describe below. For a workload distribution, we assume that an operation for a sector is independent from previous operations and follows a discrete probability distribution and thus, if some sectors are written to more often than others then they will be represented as having higher probability in the workload distribution.

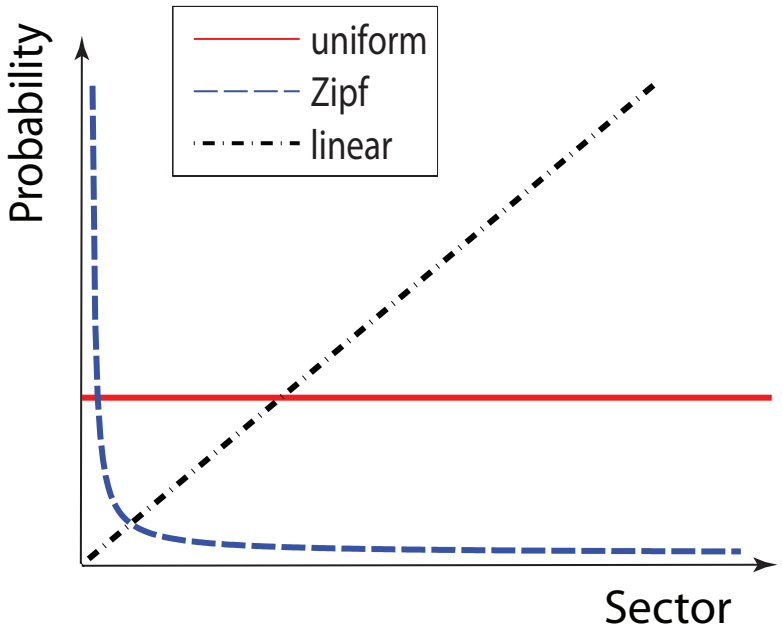


Figure 7.2: Workload Distributions

7.2.2.1 Workload Distributions

Suppose we have n sectors and a workload of length l. The following random variables model the workload:

1. O_t: the operation at time t, which is either a *write* or *delete*.
2. C_t: the sector which the operation O_t will be performed on.

Here, by our assumption $P(O_t = write) = r$ and $P(O_t = delete) = 1 - r$

We choose workload from the following three distributions (see Figure 7.2):

1. **Uniformly Random:** The probability that a sector is chosen is identical for all sectors. We define *uniformly random workload* as $\forall i, j \in \{1, 2, \ldots, n\}$

$$P(C_t = i) = P(C_t = j) = \frac{1}{n}$$

The uniform distribution is the simplest workload and has been used in various flash memory studies [80] and serves as a reference to compare to with other distributions.

2. **Zipf:** Workloads in numerous applications follow the Zipf distribution, e.g. enterprise media server workloads [25], ethernet traffic [87], internet server workloads [3] and so on. Zipf distribution is a power law probability distribution where some sectors are likely to be accessed very frequently while others are almost static. This gives us a good model of workloads that produce dynamic and static data. Thus, the *Zipf workload distribution* is defined as follows with $\alpha = 1$:

$$P(C_t = i) = \frac{\frac{1}{i^\alpha}}{\sum_{k=1}^{m} \frac{1}{k^\alpha}}$$

3. **Linear:** The *linear workload distribution* serves as an intermediate distribution from the heavy tailed Zipf workload distribution and the constant uniform workload distribution and is defined as

$$P(C_t = i) = \frac{i}{\sum_{k=1}^{m} k}$$

The Zipf workload distribution has a large tail that is uniform-like, i.e., the tail probabilities are all very close to 0 that has a similar behavior to the uniform workload distribution. The linear workload distribution has no uniform-like regions and thus also serves to highlight characteristics that the uniform and Zipf workload distribution do not exhibit.

7.2.3 Wear Leveling Strategies

When a write operation for a sector s arrives, an invalid block is chosen and erased, the clean block is then programmed with the data for sector s. A *wear leveling strategy* is defined as a function that chooses an invalid block to be erased and written to, deciding

from the current state of the flash memory. We make the following simplifying assumption about our wear leveling strategies: for each user write, there is only one corresponding write to flash memory. The strategies do not move data around internally. Thus, at time t, the mean erase count is $\frac{rt}{m}$ where again r is the ratio of writes and deletes, m is the number of sectors and t is the length of the workload.

We consider the following wear leveling strategies:

7.2.3.1 Null Strategy

In the Null strategy, sector i is mapped to block i, i.e., each write to a sector goes to the block with the same number. In effect, there is no wear leveling occurring in this strategy.

Null strategy is our baseline strategy, i.e., a strategy where no wear leveling is employed and that the statistics from this strategy will serve as a reference measure to which more sophisticated strategies can be compared against. The Null strategy thus enables us to measure how much is gained by wear leveling.

The Null strategy can also be a viable strategy in instances when very low overhead is required, requiring no data structures to hold information about the blocks.

7.2.3.2 Strategy for Mapped Blocks

Flash transition layer (FTL), a PCMCIA standard [28], was designed to provide a sector interface array while also performing wear leveling. Using the SSD's powerful micro-controller and RAM, an out-of-place write algorithm is used to schedule the operations for the flash memory using a technique called block mapping [44]. Using maps or lookup tables, a sector can be mapped to any physical block in the flash memory. These maps can be updated and a sector can use a different block at each update. This enables wear leveling as writes to a sector can be spread out over a number of blocks.

The following strategies *Random Erase Count Invalid First* (RECIF) and *Lowest Erase Count Invalid First* (LECIF) assume a block-mapping in the SSD.

1. **RECIF:** This serves as a baseline strategy for a block-mapped SSD. When a block has to be chosen to be erased, it is chosen at random from the set of blocks that are currently invalid.

2. **LECIF:** As an enhancement to block mapping, accurately keeping or estimating the erase count of each of the blocks allows for more sophisticated block mapping techniques. To facilitate this, the erase count can be stored in the block headers and the incremented count is written back to the header after each write, or that the erase count can be estimated by the erase latencies [49].

   Thus, provided some mechanism to get an erase count for a block, the LECIF strategy chooses the invalid block with the lowest erase count as the next block to be erased and written to. LECIF provides a baseline measure for SSDs with erase counts.

7.3 Quantitative Analysis of Wear Leveling Strategies

Let X_t denote the random variable for the erase count of a block at time t. Our goal is to calculate $P(X_t = k)$ for $k = 0, \ldots M$ (M is an upper bound for the erase count) for various strategies, workloads and configurations. This gives us the probability distribution of the erase counts at time t, *the wear distribution*, which we use to analyze the wear pattern characteristics and measure the effectiveness of wear leveling for the given SSD configuration. This section details the mathematical machinery to accomplish this.

We use the following random variables:

1. X_t: erase count of a block at time t

2. R_t: state of the block at time t; either *valid* or *invalid*

3. S_t: sector currently stored in the block

We use the following short-hand notation for clarity

1. $P(Xi_t = k) = P(X_t = k, R_t = invalid)$
2. $P(Xv_t^{(s)} = k) = P(X_t = k, R_t = valid, S_t = s)$

Since we use block mapping the block can have any sector stored in it, but for a valid block at time t, we only keep track of the current sector that is stored in the block.

The variables have the following properties:

- **Sum to 1:** The following holds true for each t,

$$\sum_{i=0}^{\infty} \left[P\left(Xi_t = i\right) + \sum_{s=1}^{m} P\left(Xv_t^{(s)} = i\right) \right] = 1$$

  as a block can either be invalid or valid with one of the m sectors.

- **Initial Values:** Initially all the blocks are invalid with erase count 0. Thus, we have the following initial conditions:

$$P\left(Xi_0 = 0\right) = 1$$

  while all the other probabilities are 0.

- **Independence:** The jointly distributed random variables X_t, R_t and S_t are independent with each other at time t for non-Null strategies.

7.3.1 Wear Model

For each erase count k, we have $m + 1$ states; one state for an invalid block of erase count k and m states for a valid block of erase count k with sector s stored in the block for $s = 1, \ldots, m$. We have states for erase counts $0, 1, \ldots, M$ where M is such that

$P(X_l = i) = 0$ for $\forall i > M$, i.e., at the end of processing the workload, the probability that a block has erase count greater than M is 0. Since l is the length of the workload, it serves as an upper bound for M, i.e., $M \leq l$, but we expect M to be significantly lower since we are using wear leveling.

A block can move from a valid state to an invalid state of the same erase count, and from an invalid state to a valid states with one higher erase count. For each time t, each state has a probability that a block is in that state: $P(Xi_t = k)$ gives the probability that the block is invalid and has erase count k and $P(Xv_t^{(s)} = k)$ gives the probability that it is valid with sector s stored in it and has erase count k. We then calculate

$$P(X_t = k) = P(Xi_t = k) + \sum_{s=1}^{m} P(Xv_t^{(s)} = k)$$

for $k = 0, 1, \ldots, M$ which gives the probability that a block has erase count k. Thus, from this we get a probability distribution function of the erase counts, the *wear distribution*.

Since we are interested in wear leveling, we need a quantification of the effectiveness of wear leveling for which we use a measure of dispersion of the wear levels among the blocks. We use the variance as the dispersion measure where a lower variance indicates a more effective wear leveling; variance zero indicating all blocks have the same erase count and thus the best possible wear leveling.

7.3.2 Null Strategy Recursive Equations

For the Null strategy, the erase count of a block k only increases when there is a write to block k and delete operations can be ignored. Thus, we do not have to keep track of invalid blocks and we only need to keep track of the variable X_t, the erase count of a block. Since the wear distribution of each block could be different, we use X_t^s to denote the erase count of block s.

Let $q_t^{(s)}$ denote the probability that block s will be written to next. Then,

$$q_t^{(s)} = P(O_t = w)P(C_t = s) = rP(C_t = s)$$

The probability that a block has erase count k at time t is that it has erase count k at time $t-1$ and the block wasn't chosen to be written to at time t. The other case is that block s had erase count $k-1$ and a write to block s incremented the erase count to k. Thus, the probability that block s has erase count k at time t is

$$P(X_t^s = k) = P(X_{t-1}^s = k)\left(1 - q_t^{(s)}\right) + P(X_{t-1}^s = k-1)\, q_t^{(s)}$$

Example Let the workload distribution be uniformly random. Then, we have

$$q_t^{(s)} = \frac{r}{m}$$

Thus, our recursive formula for calculating the distribution at time t becomes

$$P(X_t^s = k) = P(X_{t-1}^s = k)\left(1 - \frac{r}{m}\right) + P(X_{t-1}^s = k-1)\,\frac{r}{m}$$

Solving for X_t^s, we see that it is the binomially distributed as proved in B.0.1. As r and m are fixed and as t varies, X_t^s depends only on X_{t-1}^s. We have the initial values X_0^s and using the above recursive equations, we calculate X_t^s.

7.3.3 Block Mapped Strategies

Modeling of block mapped strategies follows from the following observations:

1. **Same wear distribution for all blocks:** With block mapping, every block in the flash memory has the same wear distribution. The next block to be written to is

chosen independent of the block number since the strategies only have information about the erase count and the valid/invalid state of the blocks. The physical block number and the mapping tables are never available to the strategy. In other words, $X_t^s = X_t^r$ for all $r, s \in \{1, \ldots, m\}$ and we only use X_t as the variable.

2. **$q_t^{(s)}(k)$ defines the strategy:** Let $q_t^{(s)}(k)$ be the probability that at time t, a block with erase count k will be chosen by the strategy for writing and will contain sector s after writing. Note that after writing the block will have erase count $k+1$.

   For each wear leveling strategy, we will use a particular $q_t^{(s)}(k)$ for it which we give below and $q_t^{(s)}(k)$ wholly determines the strategy in our model. We call $q_t^{(s)}(k)$ for a strategy the *block selection probability*.

We next define a set of recursive equations that defines block mapping.

7.3.3.1 Valid Blocks of Erase Count k

For valid blocks of erase count k at time t which contains sector s,

- it can gain a block when the strategy chooses a block of erase count $k-1$ to write to, making it a valid block of erase count k with sector s on it with probability (see Figure 7.3)

$$r\ q_t^{(s)}(k-1)$$

- it can lose a block to the set of invalid blocks of erase count k by a delete or a write on a block of erase count k with sector s with probability (see Figure 7.4)

$$P(Xv_{t-1}^{(s)} = k)\ P(C_t = s)$$

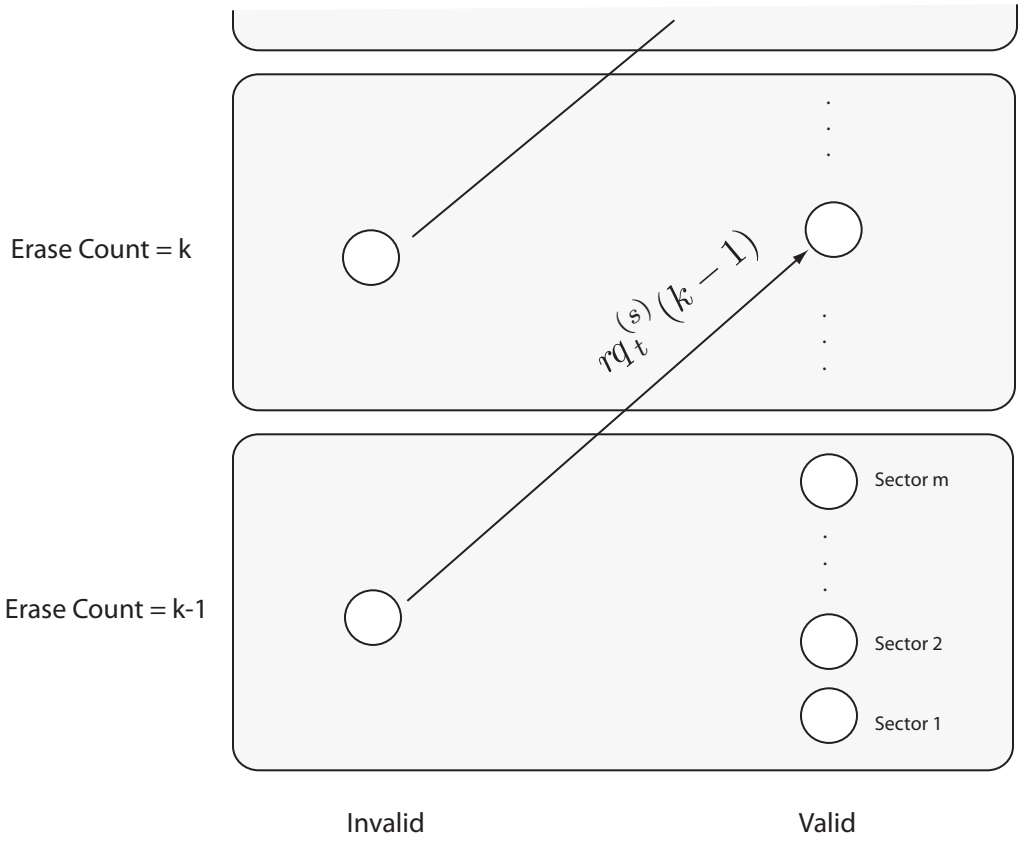


Figure 7.3: Writes increasing valid blocks

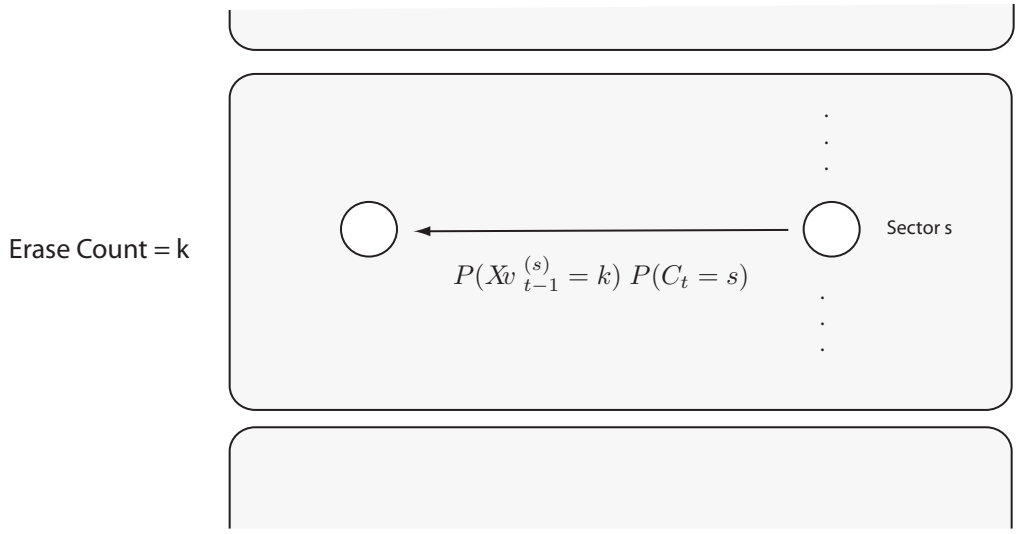


Figure 7.4: Delete or write decreasing valid blocks

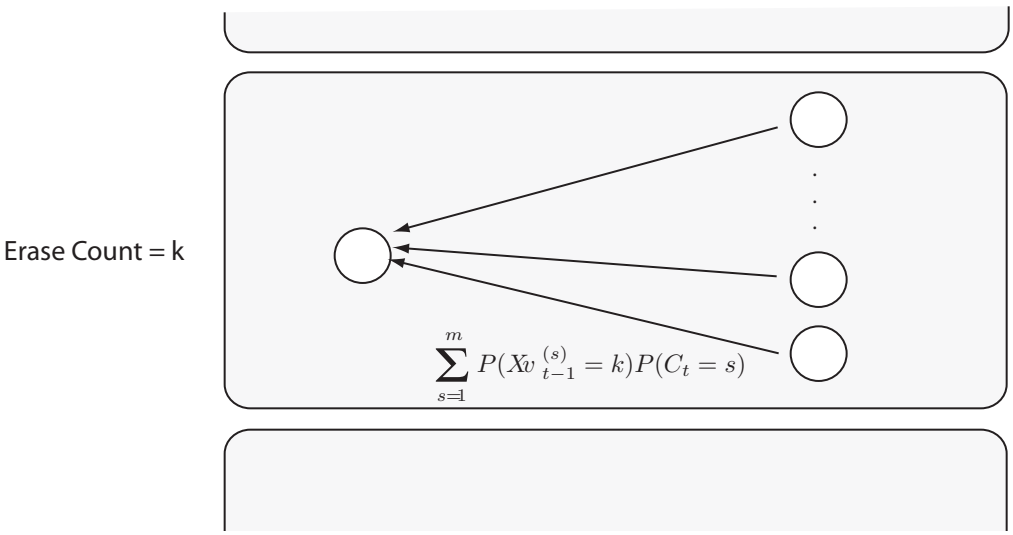


Figure 7.5: Delete or write increasing invalid blocks

Thus, we have

$$P(Xv_t^{(s)} = k) = P(Xv_t^{(s)} = k) - P(Xv_{t-1}^{(s)} = k)\ P(C_t = s) + r\ q_t^{(s)}(k-1)$$
$$= P(Xv_t^{(s)} = k)\,[1 - P(C_t = s)]\ +\ r\ q_t^{(s)}(k-1)$$

7.3.3.2 Invalid Blocks of erase count k

For the set of invalid blocks of erase count k at time t,

- it gains a block when a valid block becomes invalid by a delete or write to the sector on the block with probability (see Figure 7.5)

$$\sum_{s=1}^{m} P(Xv_{t-1}^{(s)} = k)P(C_t = s)$$

- it loses a block when the strategy chooses a block of erase count k to write to with probability (see Figure 7.6)

$$r \sum_{s=1}^{m} q_t^{(s)}(k)$$

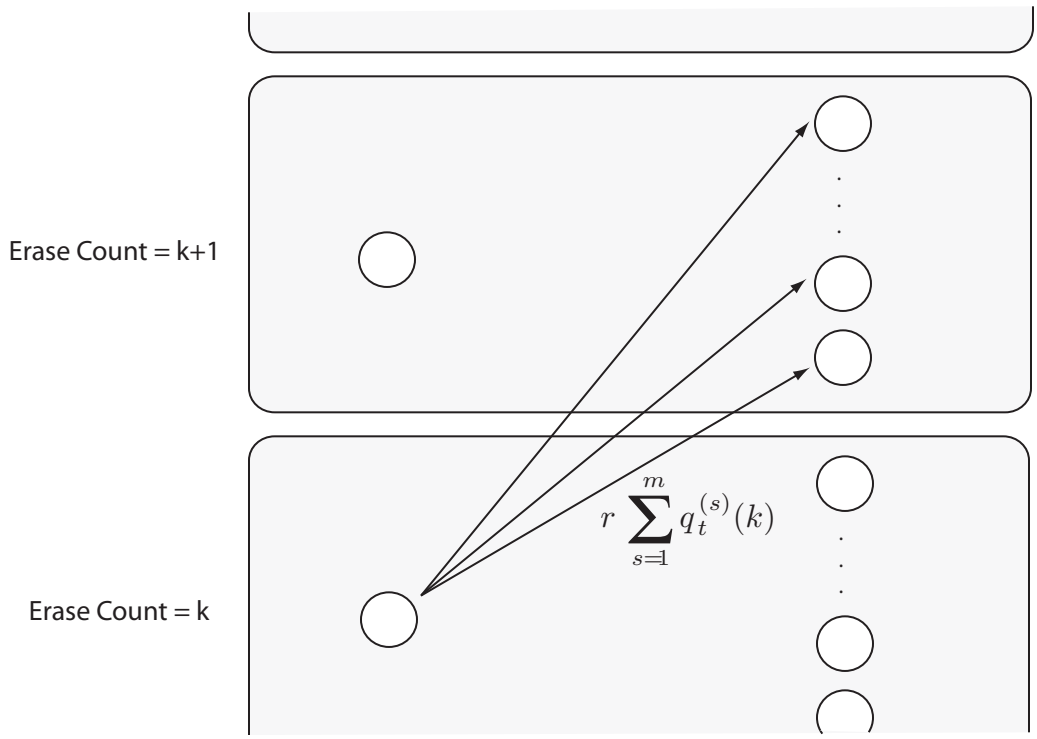


Figure 7.6: Write decreasing invalid blocks

Thus, we have

$$P(Xi_t = k) = P(Xi_{t-1} = k) + \sum_{s=1}^{m} \left[P(Xv_{t-1}^{(s)} = k) P(C_t = s) - r\ q_t^{(s)}(k) \right]$$

7.3.3.3 Using the recursive equations

We thus have the following recursive equations for modeling block mapped wear leveling

$$P(Xi_t = k) = P(Xi_{t-1} = k) + \sum_{s=1}^{m} \left[P(Xv_{t-1}^{(s)} = k) P(C_t = s) - r\ q_t^{(s)}(k) \right]$$

$$P(Xv_t^{(s)} = k) = P(Xv_{t-1}^{(s)} = k)\left[1 - P(C_t = s)\right] + \ r\ q_t^{(s)}(k-1)$$

Here, r is fixed and $P(C_t = s)$, the workload distribution, is also fixed. Thus, at each step we need to calculate $q_t^{(s)}(k)$, $P(Xi_t = k)$ and $P(Xv_t^{(s)} = k)$ for $s \in \{1, \ldots, m\}$, which are only dependent on $P(Xi_{t-1} = k)$ and $P(Xv_{t-1}^{(s)} = k)$ for $s \in \{1, \ldots, m\}$, the probabilities in the previous step. Thus, we can recursively calculate the above values,

and then calculate $P(X_t = k)$ from $P(Xi_t = k)$ and $P(Xv_t^{(s)} = k)$.

7.3.3.4 Properties of $q_t^{(s)}(k)$

$q_t^{(s)}(k)$ has the following properties:

1. Suppose we have a strategy that chooses the next block to be written independently of the sector that will be written to the block. Then, we can define

$$q_t(k)$$

as the probability that a block with erase count k will be chosen by the strategy at time t independent of the sector stored in the block. We can simplify as

$$q_t^{(s)}(k) = q_t(k)\ P(C_t = s) \text{ and } q_t(k) = \sum_{s=1}^{m} q_t^{(s)}(k)$$

which we call *sector independent*.

2. Given n blocks, we have

$$\sum_{k=0}^{\infty}\sum_{s=1}^{m} q_t^{(s)}(k) = \frac{1}{n} \tag{7.1}$$

and this gives us the following for sector independent $q_t(k)$

$$\sum_{k=0}^{\infty} q_t(k) = \frac{1}{n}$$

The above serves as a necessary condition to check that the formulation of a strategy's $q_t^{(s)}(k)$ values is correct. For all the strategies that we define in the following section, we verify that the above necessary condition holds. We prove the above property in B.0.3 and B.0.4.

3. We have the following equations which can be derived from the assumption that the

mean is $\frac{rt}{m}$

$$P(X_t = k) - P(X_{t-1} = k) = r \sum_{s=1}^{m} \left[q_t^{(s)}(k-1) - q_t^{(s)}(k) \right]$$

which simplifies to the following when $q_t(k)$ is sector independent.

$$P(X_t = k) - P(X_{t-1} = k) = r \left[q_t(k-1) - q_t(k) \right]$$

Thus, the change in the erase count at each time t can be expressed in terms of $q_t^{(s)}(k)$ and is proved in B.0.2.

4. The variance of the wear distribution can be calculated directly from $q_t(k)$ and is given by

$$\mathrm{Var}(X_t) = \mathrm{Var}(X_{t-1}) + r \sum_{k=0}^{\infty} k^2 Q_t(k) - (2t-1)\frac{r^2}{m^2}$$

where

$$Q_t(k) = \sum_{s=1}^{k} \left[q_t^{(s)}(k-1) - q_t^{(s)}(k) \right]$$

Telescoping the above and using the fact that $\mathrm{Var}(X_0) = 0$, we have the following:

$$\mathrm{Var}(X_t) = r \sum_{\tau=1}^{t} \sum_{k=0}^{\infty} k^2 Q_\tau(k) - \left[\frac{(rt)}{m} \right]^2 \tag{7.2}$$

Here, r, t and m are determined beforehand and $\frac{rt}{m} = E[X]$. The above equation is proved in B.0.6.

7.3.4 $q_t^{(s)}(k)$ for strategies

The next step is to define $q_t^{(s)}(k)$ for the various strategies which will complete all the parts required to compute the wear distributions. We assume that the block to erase and write is chosen before the block containing sector s is marked invalid and in this case $q_t^{(s)}$

is independent of s and we only need to define $q_t(k)$ for the following strategies.

7.3.4.1 RECIF

An invalid block is chosen at random to fulfill the next write. Let

$$p_{t-1} = \sum_{i=0}^{\infty} P(Xi_{t-1} = i)$$

or the probability that a block is invalid at time $t-1$. Let I_t be the random variable that gives us the number of invalid blocks in $\mathcal{F}$ at time t. Then,

$$E[I_{t-1}] = np_{t-1}$$

We have,

$$q_t(k) = \frac{P(Xi_{t-1} = k)}{E[I_{t-1}]}$$

Note that for large t, $p_t = p_{t-1}$ and the following holds

$$\sum_{k=0}^{\infty} q_t(k) = \frac{\sum_{k=0}^{\infty} P(Xi_{t-1} = k)}{np_{t-1}} = \frac{p_t}{np_{t-1}} \to \frac{1}{n}$$

which is the probability of choosing an invalid block of erase count k choosing from all the invalid blocks.

7.3.4.2 LECIF

In this strategy, we choose from the set of blocks that were invalid at time $t-1$ and have the lowest erase count to write to.

7.3.4.2.1 Defining $\alpha_t(k)$, $\beta_t(k)$ and $\Omega(k)$ To define $q_t(k)$, we first define $\alpha_t(k)$ and $\beta_t(k)$ as follows:

$$\alpha_t(k) = P(Xi_{t-1} = k)$$
$$\beta_t(k) = P(Xv_{t-1}) + P(Xi_{t-1} > k)$$

Here, $P(Xv_{t-1})$ is the probability that a block is valid regardless of the erase count and the sector stored on it. Thus,

$$P(Xv_{t-1}) = \sum_{k=1}^{\infty} \sum_{s=1}^{m} P(Xv_{t-1}^{(s)} = k)$$

Here, we have

- $\alpha_t(k)$ gives the probability that an invalid block is of erase count k
- $\beta_t(k)$ gives the probability that a block is either valid or invalid with erase count greater than k

We define

$$\begin{aligned}\Omega_t(k) &= \sum_{i=0}^{n-1} \frac{1}{i+1} \binom{n-1}{i} [\alpha_t(k)]^i [\beta_t(k)]^{n-1-i} \\ &= \frac{[\alpha_t(k) + \beta_t(k)]^n - \beta_t(k)^n}{\alpha_t(k)\ n}\end{aligned}$$

The above quantifies how LECIF chooses a block; the unique lowest erase count invalid block is the next block to write to or if there are more than one invalid block with the lowest erase count, then one of the blocks with the lowest erase count is chosen at random.

7.3.4.2.2 Defining $q_t(k)$ For LECIF, we use

$$q_t(k) = P(Xi_{t-1} = k)\ \Omega_t(k)$$

This gives the probability that an invalid block with erase count k was chosen: either the block was invalid and had the unique lowest erase count k at time t or among all the invalid blocks that had the lowest erase count k, the block was chosen at random. Property 2 given by equation (7.1) is shown in B.0.5.

7.3.4.3 Round-Off Errors in Numerical Recursion

One of the dangers in calculating $q_t^{(s)}(k)$ is the division by a small number that can magnify errors. This can be seen in

$$\Omega_t(k) = \frac{[\alpha_t(k) + \beta_t(k)]^m - \beta_t(k)^m}{\alpha_t(k)\ m}$$

However, this can be easily avoided by setting

$$\Omega_t(k) = \sum_{i=0}^{m-1} \frac{1}{i+1} \binom{m-1}{i} [\alpha_t(k)]^i [\beta_t(k)]^{m-1-i}$$

which avoids division by small floating point numbers.

Thus, the only division needed is for r and n which are large numbers that will not result in large errors. The rest of the calculations are done via multiplications and additions of double precision floating point numbers. Though floating point errors still accumulate over time from addition and multiplication, it is not catastrophic as division by a small number. Thus, we would expect our errors to be small and our results accurate.

7.4 Analysis

In the previous section, we presented recursive equations to compute wear distributions. In this section, we use the equations to generate wear distributions for analysis of wear leveling while varying wear leveling strategies, workload characteristics, amount of over-provisioning through r, size of the flash memory through m and the length of the workload

through l. In other words, for each strategy, we generate wear level distributions for uniform, Zipf and linear workload distributions while varying r, m and l.

Increasing amount of information of the state of the SSD is used by the strategies going from Null to RECIF, and then to LECIF. For the Null strategy, there is no information maintained about the blocks. For RECIF, we keep track of which blocks are valid and which blocks are invalid, and for LECIF, we additionally keep track of the erase count of each block. We expect strategies that utilize more information on the state of the SSD to give us better wear leveling which is reflected through a lower variance of the wear distribution. However, the tradeoff for better wear leveling is the complexity imposed by the wear leveling strategy itself. Our goal in this section is to understand how wear leveling is affected by the wear leveling algorithm, workload, r, m and l. This will aid SSD designs which has to optimize between SSD system complexity and the disk endurance parameters.

7.4.1 Null Strategy

Analysis of the Null strategy is greatly simplified because $P(X_t = k)$ can be explicitly stated as a binomial distribution $B(\hat{n}, \hat{p})$ where $\hat{n}$ is the number of writes and $\hat{p} = \frac{1}{m}$ (input uniformly distributed). Then, by the property of the binomial distribution, the mean is $\hat{n}\hat{p}$ and the variance is $\hat{n}\hat{p}(1-\hat{p})$. For large n, the binomial distribution can be approximated closely by a normal distribution $\mathcal{N}(\hat{n}\hat{p}, \hat{n}\hat{p}(1-\hat{p}))$.

For the Null strategy, varying r does not influence the wear distribution (provided that the number of writes is constant). If we vary the number of writes, then the mean and variance change linearly (as can be seen from the equations for the mean and the variance)

$$\hat{n}\hat{p} = \frac{l}{m} \qquad \hat{n}\hat{p}(1-\hat{p}) = \frac{l(m-1)}{m^2}$$

As we vary the number of writes $\hat{n}$, the mean and variance vary linearly with it.

For the Zipf and linear workload distributions, the wear distribution varies by each block. Each block has its own normally distributed wear distribution $\mathcal{N}(\hat{n}, \hat{p}\ P(C_t = s))$. The final distribution of the entire flash memory is the sum of the individual distributions $\sum_{s=1}^{n} \mathcal{N}(\hat{n}, \hat{p}\ P(C_t = s))$. Figure 7.12c shows the output distribution of the Null strategy with the the various workload distribution for $r = 0.9$, $m = 50$ and $l = 1,000$.

7.4.2 Block Mapped Strategies

As shown in Section 7.3.3, each block in the flash memory using a block mapped wear leveling strategy has the same wear level distribution and has mean $\frac{rt}{m}$ after t operations. Using the variance of the wear distribution as the measure of the effectiveness of wear leveling, we make the following observations:

7.4.2.1 Shape of the Wear Distribution

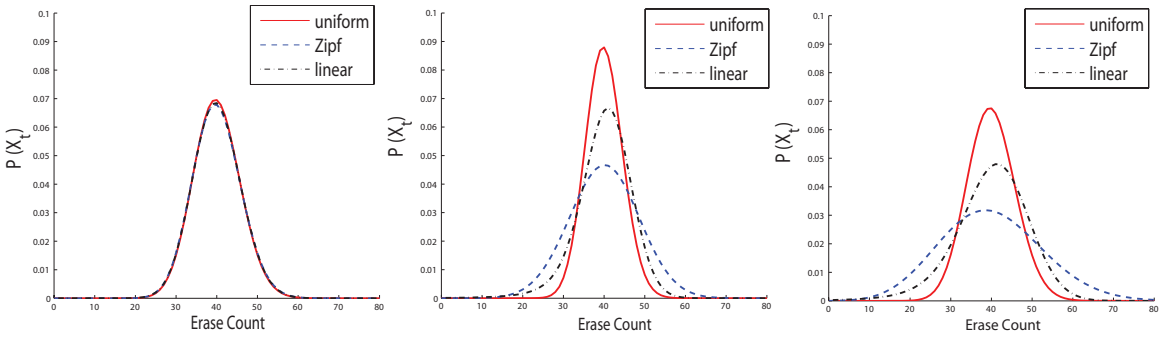
Given $P(X_t = i)$ for $i = 0, 1, \ldots, M$ (the probability that at index t the block has erase count k), we calculate the mean $\bar{X}_t$ as

$$\bar{X}_t = \sum_{i=1}^{M} iP(X_t = i)$$

and variance σ_t^2 as follows

$$\sigma_t^2 = \sum_{i=1}^{M} \left[i^2 P(X_t = i)\right] - \bar{X}_t^2 = \sum_{i=1}^{M} \left[(i - \bar{X}_t)^2 P(X_t = i)\right]$$

7.4.2.1.1 RECIF For small values of r, as shown in Figure 7.7a, ($r < 0.1$), the wear distribution is close to normal for all workload distributions. For larger r (or for lower over-provisioning), as seen in Figure 7.7b and 7.7c, the variance of the wear distribution generated by the Zipf or linear workloads is higher than for the uniform workloads. For

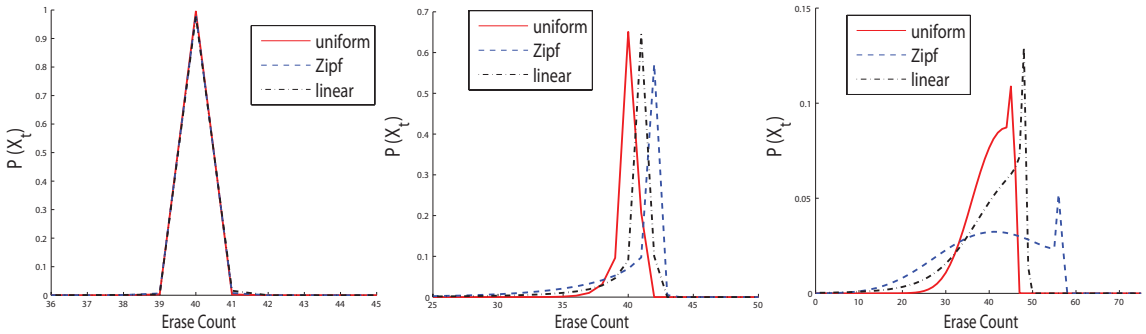


(a) $r = 0.1$ (b) $r = 0.6$ (c) $r = 0.95$

Figure 7.7: The shape of the wear distribution for various r using the RECIF strategy for $m = 250$ and $5,000$ write operations.

the linear workload, for high r, there is a negative skew which results in a longer tail to the left of the peak of the distribution. The skew is present because non-uniform workloadsa have static blocks which stay valid for a long time whereas invalid blocks all have equal probability of being selected for erasing by the RECIF strategy.

7.4.2.1.2 LECIF For small values of r, e.g. $r = 0.1$ as shown in Figure 7.8a, the probability is almost one at the mean. In this case, most erase counts of the blocks in the flash memory would be one or two erase counts away from the mean, thus giving a very low wear variance. For slightly larger values of r, e.g. $r = 0.6$ as in Figure 7.8b, there is a presence of a long tail on the left side while the right side of the distribution is steep. However, the majority of the distribution is still concentrated at the mean. For $r = 0.95$ as in Figure 7.8c, we see a smooth left distribution which ends on the right at a spike. The right side of the distribution is steep because as LECIF selects a block with the lowest erase count, a block with a high erase count has very low probability of being selected. A block with the highest erase count can only be selected if every other block also has the same erase count.



(a) $r = 0.1$ (b) $r = 0.6$ (c) $r = 0.95$

Figure 7.8: The shape of the wear distribution for various r using the LECIF strategy for $m = 250$ and $10,000$ write operations.

7.4.2.2 Varying l, m and r

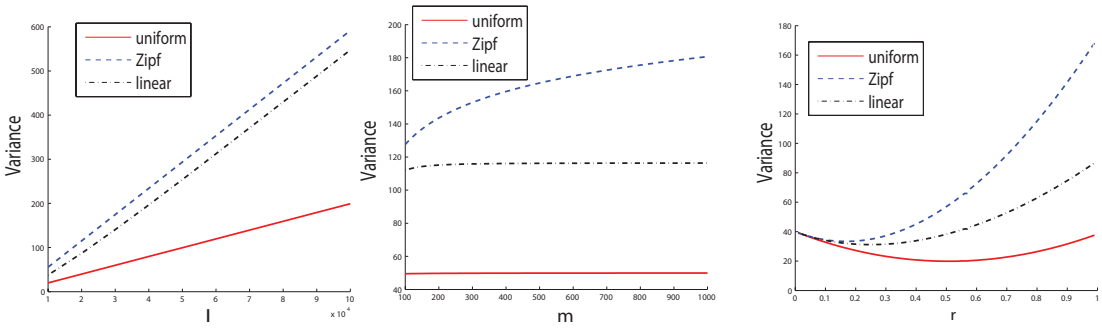
We analyze the behavior of the variance as we vary l, m and r for the various workloads and wear leveling strategies. Unlike the Null strategy, we have not found closed form expressions for the variance in terms of m, l and r and our analysis will be based on the analysis of the numerically generated wear distributions and its graphs.

7.4.2.2.1 Data Normalization The wear distribution is influenced by the number of writes per block; as we vary m and r, the number of writes per block also changes. For each of these cases, we normalize the number of writes so that the mean write per block remains the same.

When we increase m and keep l constant, the number of writes per block decreases, and thus the mean and the variance change along with it. While varying m, we keep the mean writes per block constant by increasing l, i.e., let C be a constant and we set $l = mC$ and mean writes per block is $\frac{rmC}{m} = rC$ which is independent of m.

When varying r, we also keep the mean writes per block constant by varying l, i.e., let w be a constant and let $l = \frac{w}{r}$, then the mean writes per block is

$$\frac{r\frac{w}{r}}{m} = \frac{w}{m}$$



(a) Varying l for $m = 250$ and $r = 0.5$ (b) Varying m for $r = 0.95$ (c) Varying r for $w = 5,000$ and $m = 250$

Figure 7.9: RECIF: variances as l m and r vary

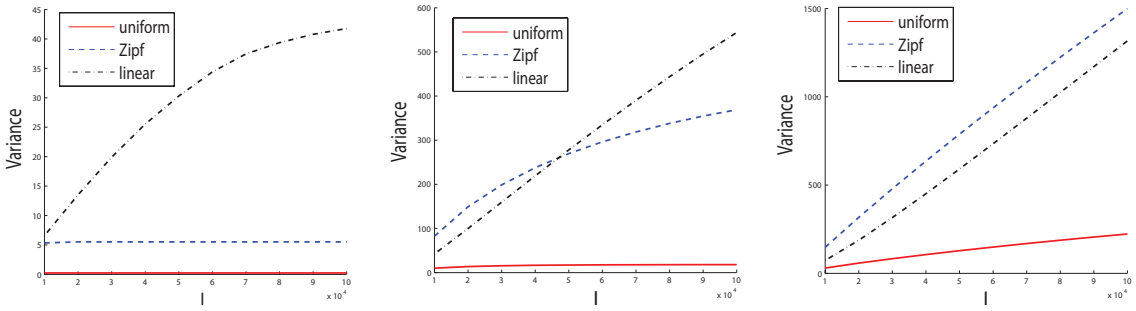
independent of r.

7.4.2.2.2 RECIF When l is increased while r and m are kept constant, the mean increases linearly with l and is identical for all the three workload distributions. The variance also increases *linearly* with l; however the rate of increase varies by the workload distribution. Figure 7.9a shows how the variance varies with l while keeping m and r constant.

Varying m, as can be seen from Figure 7.9b, when the mean write per block is constant, the variance quickly reaches a constant asymptote. However, for the Zipf distribution, it takes longer to get to the asymptote. Thus, for large enough m and for all workloads we can assume that the variance does not depend on m.

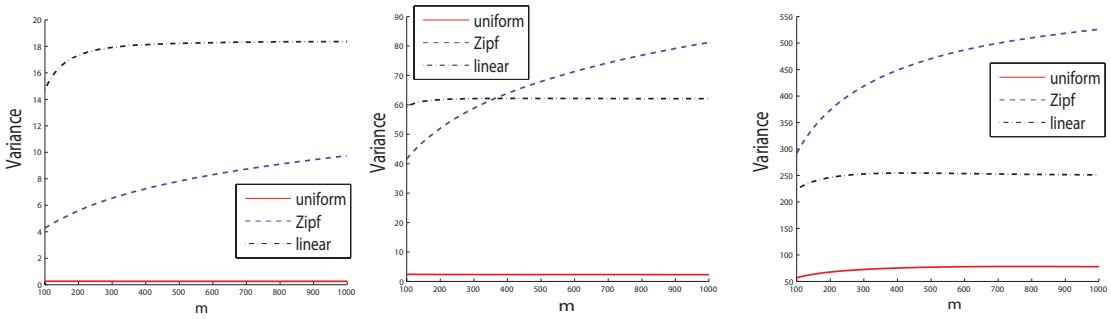
Figure 7.9c shows how the variance changes with r for fixed m and w. The variance curve is quadratic and surprisingly the minimum variance is not for the smallest value of r; for uniform workload it is around 0.5; for Zipf workload around 0.2 and for linear workload around 0.25.

7.4.2.2.3 LECIF Varying l, for $r = 0.49$, the variance for the wear distribution for the uniform workload is almost 0 as shown in Figure 7.10a. The Zipf workload also produces a constant variance for $r = 0.49$ but the variance is not a constant for the



(a) $r = 0.49$ (b) $r = 0.89$ (c) $r = 0.99$

Figure 7.10: LECIF: variance as l increases with r and m constant.



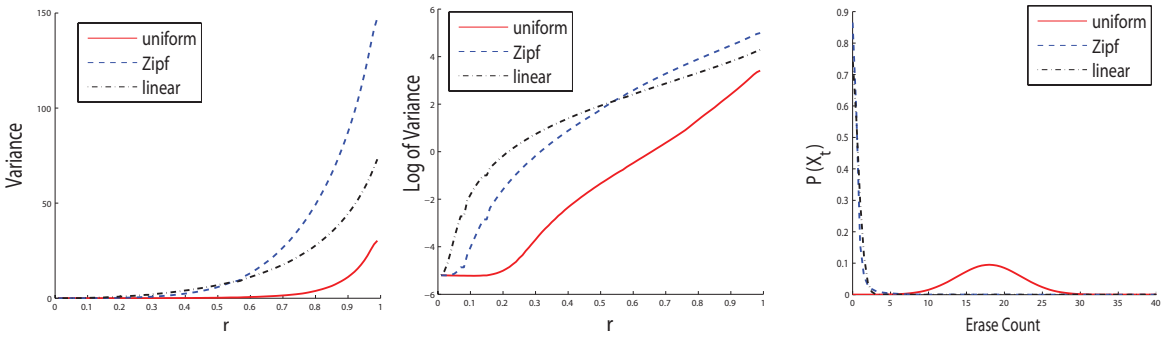
(a) $r = 0.50$ (b) $r = 0.75$ (c) $r = 0.99$

Figure 7.11: LECIF: variance as m increases with r and mean wear is constant.

linear workload and increases towards an asymptote. For $r = 0.89$, the variance for Zipf workload increases to an asymptote (Figure 7.10b) and for $r = 0.99$, all three workloads produce almost linear variance. Note that the linear workloads have a higher variance for the lower r while for $r = 0.99$, Zipf has a higher variance.

While varying m, for $r = 0.5$, all three workloads exhibit an increase towards a limit in Figure 7.11. However, it is the slowest for the Zipf workload whereas the uniform and linear workloads reach a limit quite quickly.

Varying r, for small r as can be seen in Figure 7.12a, the variance is close to 0 as almost all the blocks have erase counts very close to the mean. As Figure 7.12a shows, the variance increases exponentially as r increases. Fig 7.12b shows the log of the graph. For $r > 0.2$, the logarithm of the variance is very close to linear and so we can estimate



(a) Variance (b) Log Variance (c) Wear distributions for the Null strategy

Figure 7.12: LECIF: Variance and Log Variance as r varies with $w = 5,000$ and $m = 250$ and Wear Distribution for Null Strategy

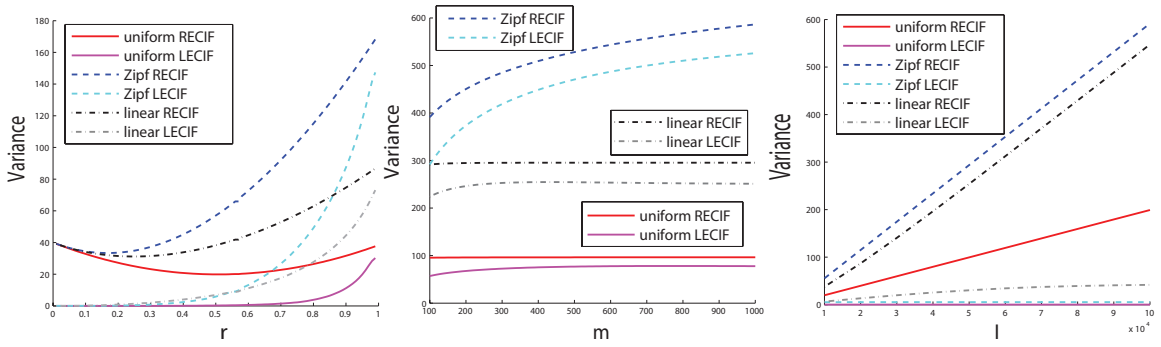
that variance as exponentially related to r.

7.4.3 Comparing the Strategies : RECIF vs LECIF

For small values of r, the variance of wear distribution is close to 0 and hence, the LECIF strategy performs very well for small r. However, as r approaches 1, the variance for the RECIF and LECIF get much closer as seen in Figure 7.13a. This indicates that for SSDs with low over-provisioning (high r), the added complexity of maintaining erase counts only yields a small improvement in the wear leveling.

Varying m while keeping the mean wear per block constant, the variance is constant over the different values of m or moving towards an asymptote (Figure 7.13b), suggesting that the total number of blocks does not play a part in wear leveling.

For increasing l and for large r, the variances for both strategies increase linearly with increasing l. However, the slopes for the RECIF curves are higher than the LECIF curves, suggesting that for large values of l, RECIF's variance goes further away from LECIF's variance. For small r, variances for LECIF reach an asymptote while the variances for RECIF increase linearly as seen in Figure 7.13c. Most SSDs perform a maintenance cycle where static data in blocks with low erase count is copied to blocks with high erase count



(a) Varying r (b) Varying m for $r = 0.99$ (c) Varying l for $r = 0.49$

Figure 7.13: Comparing the RECIF and LECIF strategies

so that blocks that previously stored static data can be used. The maintenance cycle is triggered when the variance becomes higher than a given threshold. This suggests that for small r (high over-provisioning), LECIF does not require the maintenance cycles as its variance reaches an asymptote and does not increase with l. However, for large r (low over-provisioning) and for RECIF, maintenance is required since the variance will keep on growing as l increases.

Since the variance for both the RECIF and LECIF strategies get very close together when r is large, LECIF based methods may not be the best strategy to use for systems with low over-provisioning. For high performance systems with high over-provisioning, LECIF based strategy would be the strategy of choice as it can get close to zero variance of the wear distribution. The number of blocks available in the flash memory would not play a part in the deciding the strategy as the variance remains constant. The other advantage of high over-provisioned LECIF system is that the variance over l reaches an asymptote and thus, maintenance cycles of moving static data is not required whereas RECIF systems would require it.

7.4.4 Optimal Strategy

One of the methods to define the optimal strategy is to let M be some value where if any block reaches M erase count, the entire flash memory has to be replaced. Then, the

optimal strategy can be described as the maximizing the total number of writes before a block reaches M erase count. However, this definition of optimality is not suitable where the flash memory is over-provisioned. Let N be the number of block over-provisioned and we can extend optimality to maximize the number of writes before $N+1$ blocks have erase count M. However, the flash memory could still operate under reduced storage capacity. Furthermore, the erase count dispersion is ignored and focused on a fraction of the blocks' erase counts. The endurance of a flash block and its usability at the advertised endurance level have also been investigated [29] and often exceeded with the blocks still being useful and thus, setting an upper limit on the flash block erase count might not make sense.

In our model, the optimal strategy for a given r, t and m is the strategy where the variance is minimized and from the variance equation (7.2), the expression

$$\left(r \sum_{\tau=1}^{t} \sum_{k=0}^{\infty} k^2 Q_\tau(k) - \frac{(rt)^2}{m^2} \right)$$

shows that it depends on $Q_\tau(k)$ for $\tau = 1 \ldots t$. If all the blocks have the same erase count at time t, then variance is 0.

7.4.5 Observations from Analysis

The following observations are interesting results from the analysis that are not obvious and a bit surprising.

1. When r is very high (there is very little over-provisioning), all the wear leveling strategies (Null, RECIF and LECIF) have very similar variances. Thus, for SSDs that are always almost full, there isn't any need for block mapped wear leveling algorithms, and wear leveling can be ignored.

2. For low over-provisioning, the extra complexity of LECIF might not pay off over RECIF in terms of wear leveling.

3. The number of blocks m, does not seem to influence wear leveling in our case and what holds for small systems would hold for larger systems. This can be used to obtain significant savings in the complexity of the mapping tables because a large flash memory can be mapped as a series of smaller flash memory without affecting wear leveling.

4. The best wear leveling system is high over-provisioned LECIF systems since it has very low variance of the wear distributions and does not require or requires very few static wear leveling cycles. However, to achieve the high quality wear leveling, it requires the most flash memory and the most complex block management system but the wear leveling performance can get close to optimal.

5. The RECIF strategy is less effective when r is very small than when r is between 0.3 and 0.5. The wear variance follows a quadratic shape with the minimum not close to 0.

Here we have presented a mathematical framework for analyzing whole block flash memory wear leveling where using components like $P(C_t = i)$ that models the workload and $q_t^{(s)}$ that models the strategy are used to calculate the wear distribution. This gives an SSD designer a method to investigate wear leveling under various configurations for analysis and optimal design, and also serves as a starting point for a larger model of flash memory which in the future we plan to add more flash memory features such as partial block wear leveling, complex garbage collection policies and advanced garbage collection policies for specialized workloads.

CHAPTER 8

Conclusions and Future Work

8.1 Conclusions

Flash memory promises massive performance gains, immense decrease in power usage and physical space requirements. Hence, it promises to be the storage technology of the future and provide new possibilities in IO and storage systems. However, every flash memory system must deal with its fundamental characteristics and limitations like that of wear leveling, write amplification and block cleaning before all of its positive aspects can be fully utilized.

Here, we have presented many techniques, algorithms, and models to attack the problem of endurance in flash memory by managing wear leveling, write amplification and block cleaning by using data placement, layout management and coding. From our algorithms, there are parameters where floating codes are optimal, write amplification is zero and wear variance is almost zero. Thus from this, designing and engineering around the limitations of flash memory to fully utilize the power of flash memory does seem wholly possible but possibly requires a careful management of parameters and many good algorithms to do so.

First, we presented update codes, a subset of floating codes, and gave a poset view of floating and update codes. We explored the properties of the cell, update and variable set posets in terms of their covers and roots. Next, we gave an algorithm for constructing an update code that is t_1-optimal (optimal for single cell increments) for $l = 2$ and arbitrary q, k, $n \in \{k, k+1\}$ or arbitrary n, q and $k = 2$. We proved that the algorithm indeed produces a update code that maps $(n-k+1)(q-1)$ update cell increments to a single

cell increment. An update code is equivalent to a floating code and thus, the above t_1-optimal update code also generates a t_1-optimal floating code. Next, we showed that the binary floating codes from update codes are isomorphic to l-ary floating codes. Thus, if we only need to find isomorphic binary floating codes because any higher l-ary codes are isomorphic to the binary floating codes.

Next, we explored the limits of reduction of write amplification. We did this by looking at the offline workloads and layout management. First, we showed that zero write amplification for any workload and flash memory configuration is not possible and even the worst case over-provisioning for zero write amplification is the trivial one. We can now pose the problem of finding the limits of write amplification reduction as finding the minimum write amplification given a workload and an over-provisioning amount. We estimate this minimum by using an algorithm based on the method of decomposition of the workloads. From decomposing workloads and using a simulator, we see that write amplification can be pushed to zero for moderate amounts of over-provisioning. Thus, there is a possibility that with enough information and moderate over-provisioning, write amplification can be pushed close to zero in flash memory systems.

Next, we have given simple and efficient algorithms that are effective at reducing write amplification and performing wear leveling. When data is being copied back blindly, a phenomenon called *sedimentation* occurs. Low volatility data settle at the low pages of a block evenly throughout the SSD and every copyback operation ends up copying back the low volatility data repeatedly. We have presented our technique of using separate and multiple copyback blocks that eliminates sedimentation and decreases write amplification. By sorting the garbage collection queue and using blocks of differing erase counts for write blocks and copyback blocks, we are also able to provide wear leveling in addition to write amplification reduction.

Finally. we have presented a mathematical framework for analyzing whole block flash memory wear leveling where using components like $P(C_t = i)$ that models the work-

load and $q_t^{(s)}$ that models the strategy are used to calculate the wear distribution. The components form a difference equation which we can use to model wear distributions.

8.2 Future Work

8.2.1 Update Codes

In the future, we want to extend it to arbitrary n and k, and also extend the t_1-optimal construction to a fully optimal construction by exploring multiple cell increments for variable updates in floating codes.

It is not known if t_1-optimal floating codes even exists for arbitrary k and n. It could also be the case that t_1-optimal update codes might not exist while t_1-optimal floating codes might. Due to the underlying structure of the codewords in floating codes where each cell is dependent to the other cells, many of the techniques of coding theory do not apply. Thus, new techniques may have to be developed to analyze floating codes.

The biggest unsolved problem in floating codes and update codes is if optimal codes can be found for all parameters, or a method can be found to show that optimal codes cannot exist for certain parameters. If it turns out that the currents bounds on optimality cannot be reached, then a new tight optimality criteria has to be found and codes constructed to reach it.

While floating codes provide encoding data as increments, it does not provide error-correcting capabilities. The basic poset structure of floating codes would possibly have to be revamped if we were to introduce error correcting and codeword distances.

8.2.2 Minimizing Write Amplification

While our method of estimating the minimum write amplification is practical, feasible and easy to implement, there are many theoretical questions unanswered. The very nature

of the decomposition and its properties needs to be explored. Techniques to find ways around the limitations discussed in Section 4.8.4.1 also have to be found.

On practical matters, while we have given algorithms to find the limits of write amplification reduction, we have not suggested any online algorithms based on our estimation methods. It would be of great practical value to have algorithms for online workloads. While our analysis suggests that number of copybacks can be pushed to zero, it is still unknown if we can we design a flash memory system that can actually achieve that for the workloads it receives using an online layout management algorithm. We would also like to expand our analysis to more workloads, both traces and synthetic. Especially useful would be real traces geared towards flash memory rather than magnetic hard disks. For synthetic workloads, we would also like to model other properties like sequentiality and different forms of locality.

While reducing write amplification is a major issue in flash memory, it has to be done with various complementary considerations like performance, wear leveling and limited resources for the flash memory controller. A method to analyze these different facets of flash memory alongside write amplification reduction is also needed.

8.2.3 Practical Algorithms

While we have presented a effective algorithms to reduce write amplification and perform wear leveling, this is still quite far from the maximum reduction that is possible that we calculated in Chapter 4. Our goal is to develop a practical algorithm that is able to achieve close to the theoretical limits. In order to do so, we have to explore many aspects like volatility prediction, effects of the errors in prediction, the relationship between workload predictability and write amplification, and to find practical algorithms that are simple yet effective under these considerations.

Our offline analysis had full information about the workload and thus to emulate that in practical algorithms, we would need to use a combination of application workload

knowledge and workload prediction. Even partial accuracy can be used to reduce write amplification but we still have to investigate the properties under errors in prediction or incorrect workload knowledge. After characterizing these error properties, then we can begin to understand how we can utilize partial or probabilistic workload knowledge for practical algorithms from our analysis of offline workloads.

8.2.4 Wear Level Modeling

We would like to extend the model to partial block wear leveling by finding the equivalent components for the strategy and block wear in blocks with multiple pages. The main challenge is that we have to deal with partial block data validity as a block can be in many different states of valid, invalid and clean page combinations. Additionally, there will be many different and new strategies that have to be analyzed for partial blocks as well as deal with write amplification.

After we have developed the model, it will enable us to examine mathematically write amplification and its relation to wear leveling and ways to see how we can minimize both write amplification and maintain wear leveling. Additionally, we want to examine other theoretical questions regarding strategies, workloads and their relation to wear leveling and write amplification and also use these quantitative models to create practical algorithms or fine tune existing algorithms by analyzing their mathematical characteristics.

APPENDIX A: ADDITIONAL UPDATE CODE POSETS

We present here additional poset diagrams that further illustrate the posets used in the construction of update and floating codes.

Figures A.1 and A.2 show cell posets. Figure A.1 shows a cell poset with two cells that has four levels. Figure A.2 shows cell state vectors with four cells that have three levels. This cell poset is the basis for the update code in Figure 3.7. Figures A.3 and A.4 shows variable posets for two variables with ternary variables with the empty start and the full start.

Figures A.5 and A.6 shows an update code for $n = 3, q = 3$ (3 cells of 3 levels) and $k = 3, l = 2$ (3 binary variables). This is a t-1 optimal floating code because it can do

$$(n - k + 1)(q - 1) = (q - 1) = 2$$

updates. Figures A.7 and A.8 shows an update codes for $n = 4, q = 3$ (4 cells with 3 levels) and $k = 4, l = 2$ (4 binary variables). Similarly, this is also a t_1-optimal code as it allows $q - 1 = 2$ updates. After these two updates, it will require more than a single increment to represent all variable updates and hence, we have left it grey. Note that the cell poset $n = 4, q = 3$ is the same used for the update code example in Figure 3.7.

Figure A.9 shows the update code in figure 3.11 in a cell poset. This is the same cell poset $n = 4, q = 3$ used in our previous example. As per our assertion, the $(n = 4, q = 3, k = 2, l = 3)$ is isomorphic to $(n = 4, q = 3, k = 4, l = 2)$ from Theorem 3.5.1 and Figures A.9 and A.8 look identical in structure.

The figures also illustrate that a large portions of the generations are not used (shown as light gray). We have only given t-1 optimal constructions when $n = k$ or $n = k - 1$. When n is large, we want k to be small so that a large array of cell with a few levels can

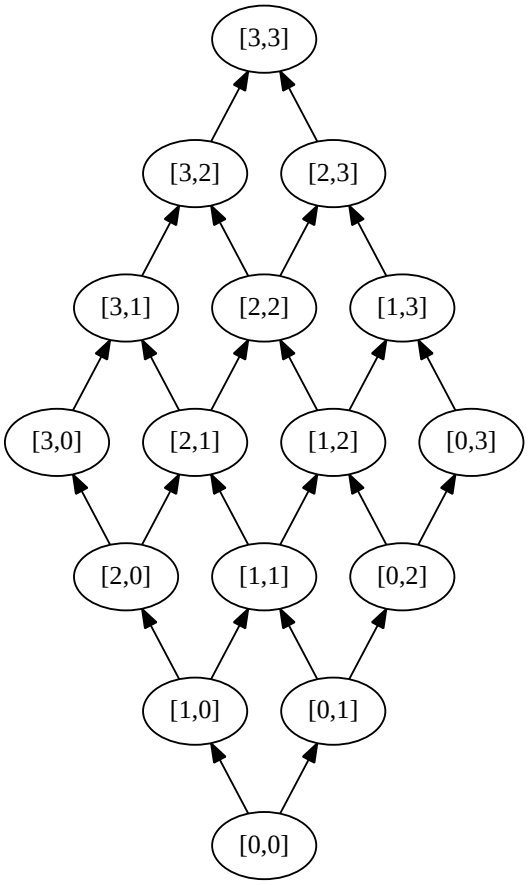


Figure A.1: Cell Poset for $n = 2$ and $q = 4$

simulate a small number of update cells with a large number of levels for useful update codes. Thus, finding optimal or near-optimal good update codes for large n and small k is important.

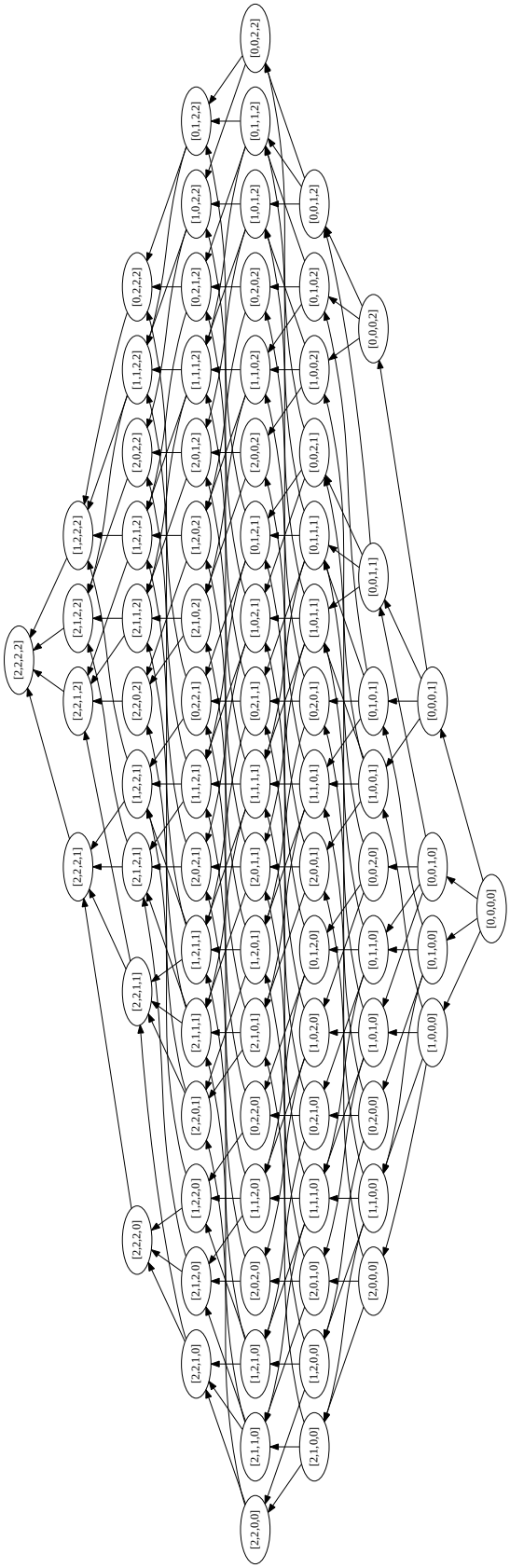


Figure A.2: Cell Poset for $n = 4$ and $q = 3$

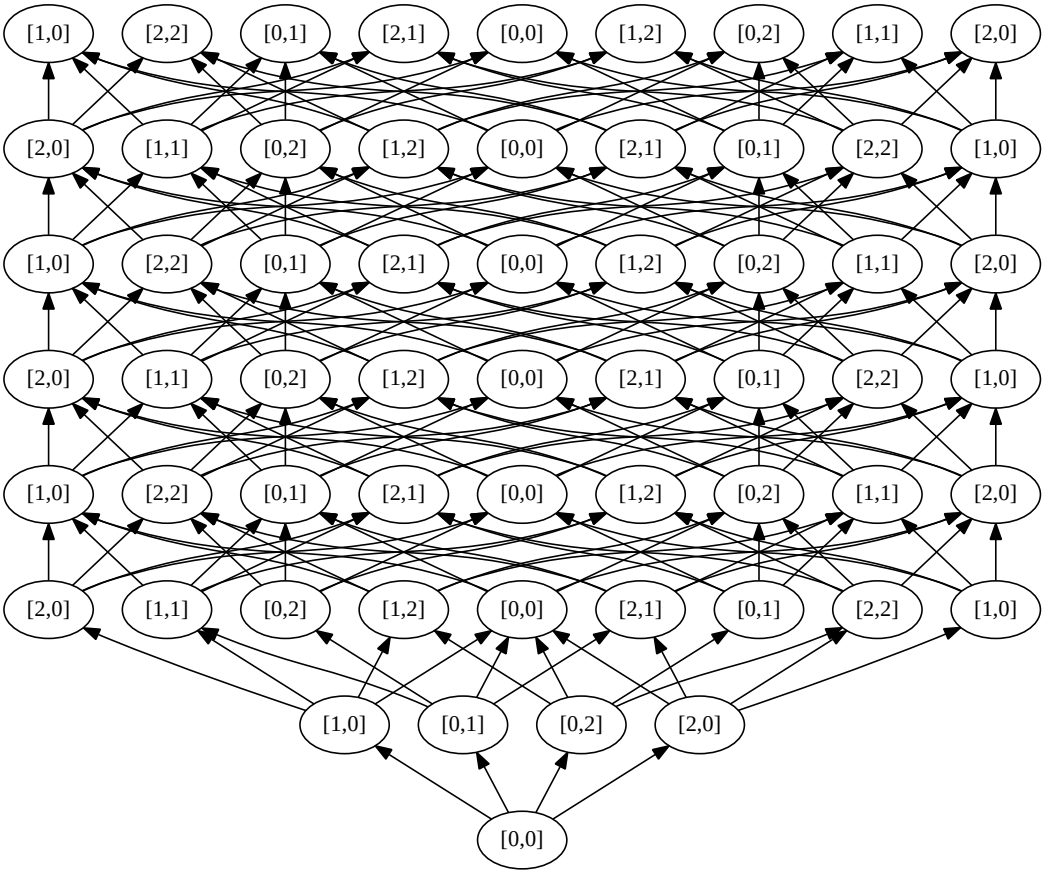


Figure A.3: Zero Start Variable Poset for $k = 2$ and $l = 3$

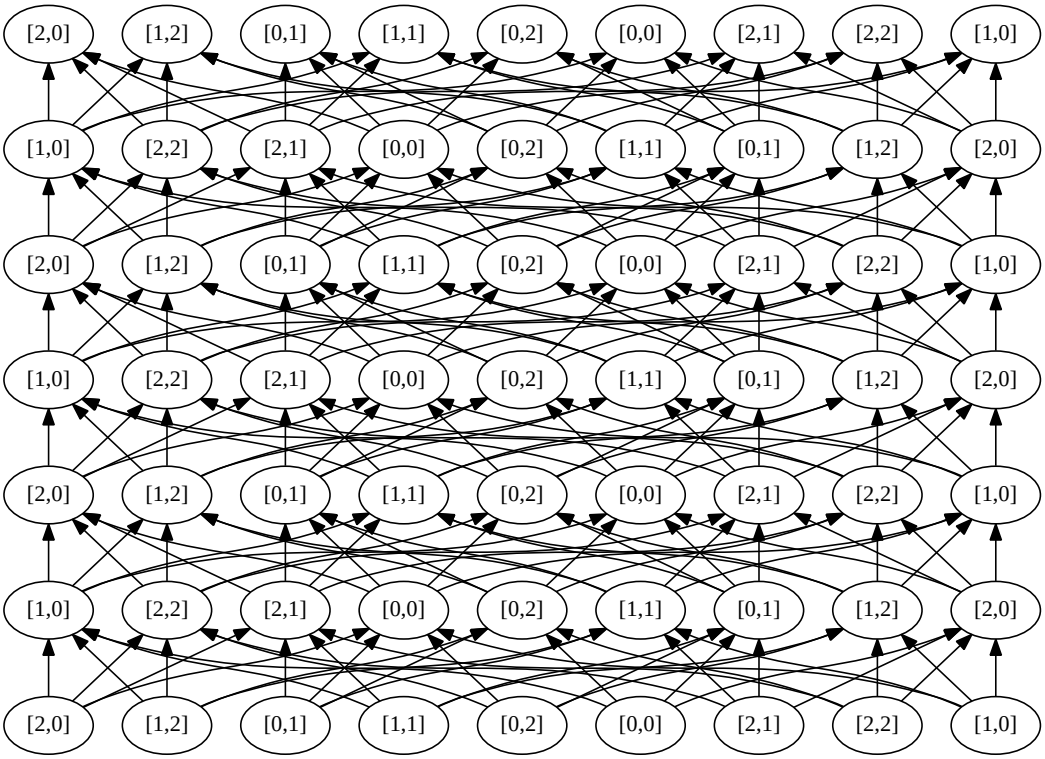


Figure A.4: Full Start Variable Poset for $k = 2$ and $l = 3$

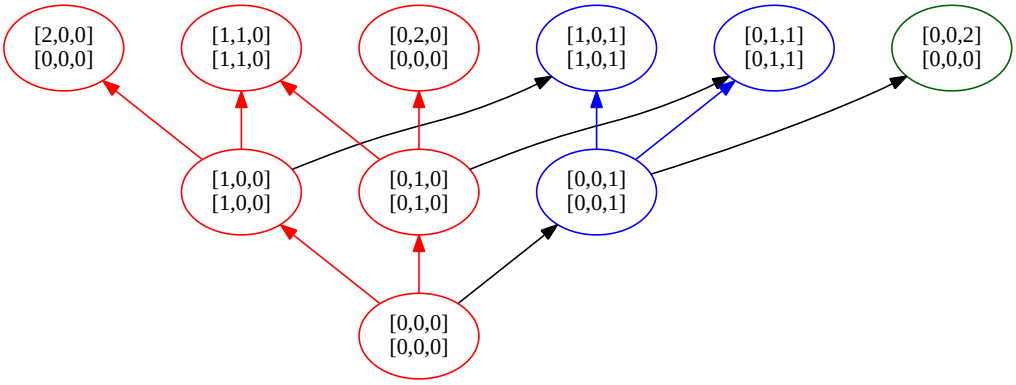


Figure A.5: t_1-optimal update code for $n = 3, q = 3, k = 3, l = 2$

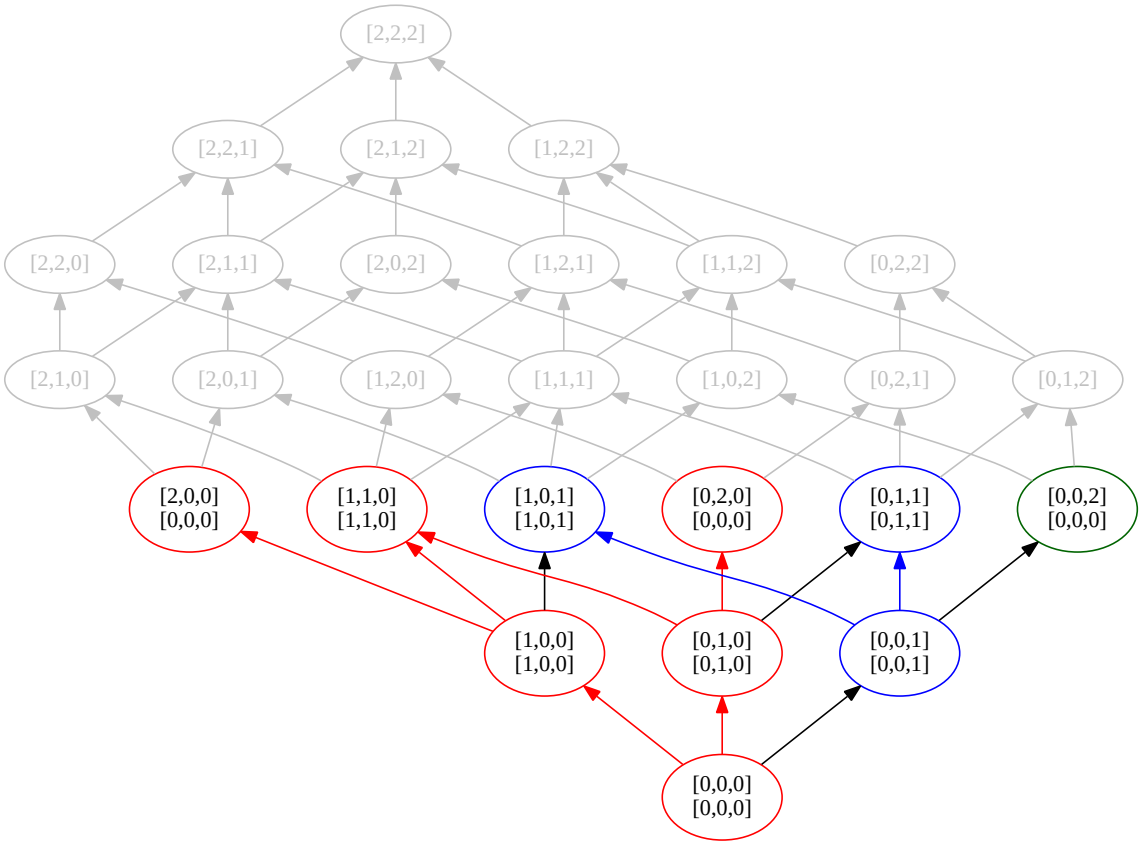


Figure A.6: t_1-optimal update code for $n = 3, q = 3, k = 3, l = 2$ in cell poset

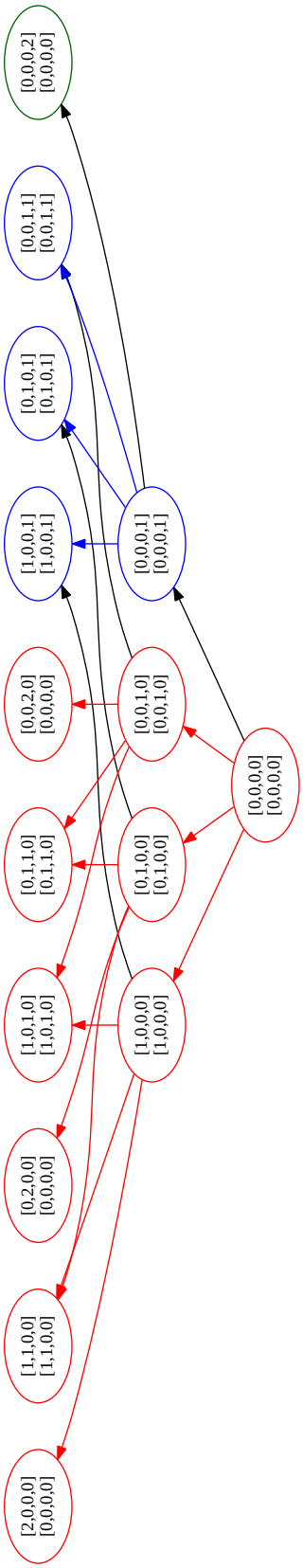


Figure A.7: t_1-optimal update code for $n = 4, q = 3, k = 4, l = 2$

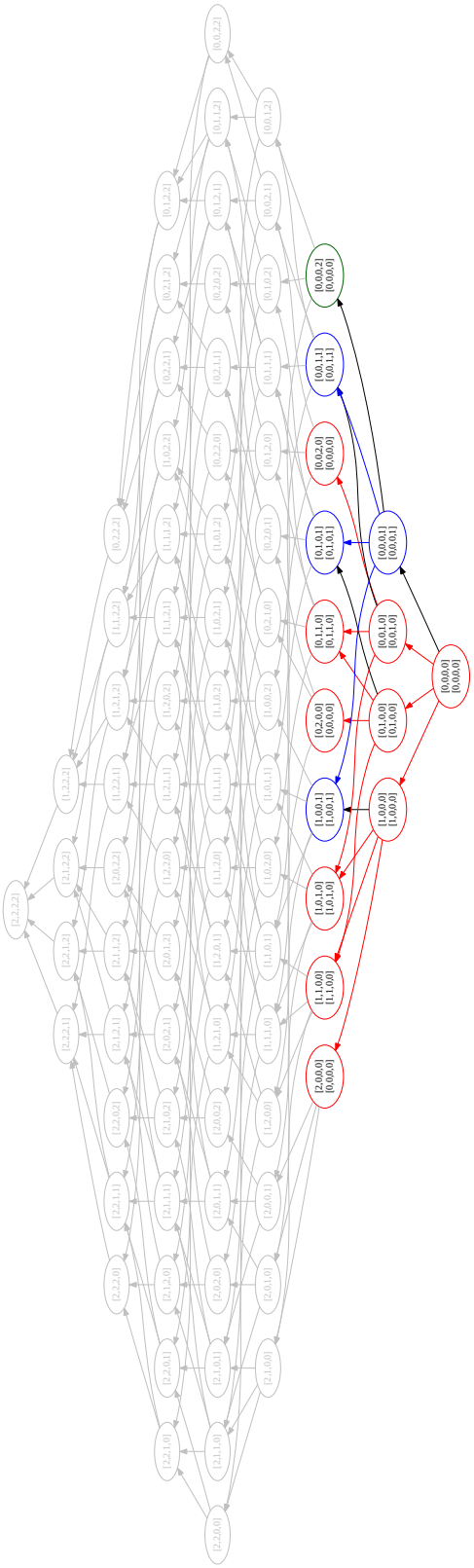


Figure A.8: t_1-optimal update code for $n = 4, q = 3, k = 4, l = 2$ in cell poset

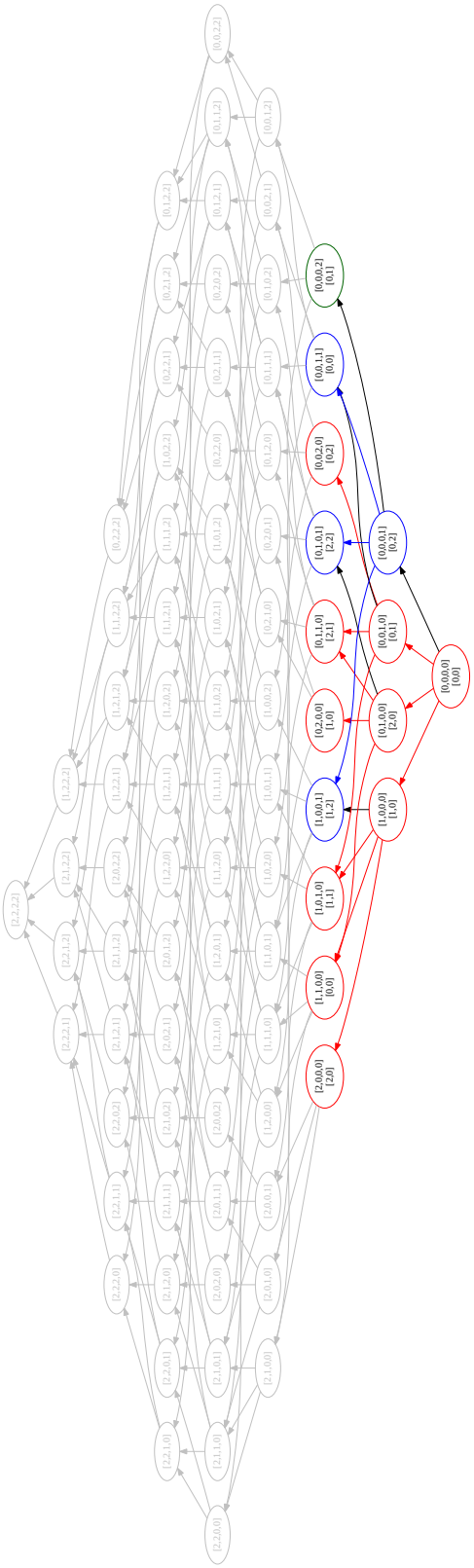


Figure A.9: t_1-optimal update code for $n = 4, q = 3, k = 2, l = 3$ in cell poset

APPENDIX B: PROOFS FOR WEAR LEVEL MODELING

Lemma B.0.1. *The wear distribution for the Null strategy for the uniform workload is binomial.*

Proof. For the uniform workload, we have the following recursive equation for the Null strategy

$$P(X_t = i) = P(X_{t-1} = i)\left(1 - \frac{r}{m}\right) + P(X_{t-1} = i-1)\frac{r}{m}$$

Now, the above equation has the terms $P(X_{t-1} = i)$ and $P(X_{t-1} = i-1)$, which we can determine from the above as

$$P(X_{t-1} = i) = P(X_{t-2} = i)\left(1 - \frac{r}{m}\right) + P(X_{t-2} = i-1)\frac{r}{m}$$
$$P(X_{t-1} = i-1) = P(X_{t-2} = i-1)\left(1 - \frac{r}{m}\right) + P(X_{t-2} = i-2)\frac{r}{m}$$

Combining the above together, we have

$$\begin{aligned} P(X_t = i) &= P(X_{t-2} = i)\left(1 - \frac{r}{m}\right)^2 \\ &\quad + 2P(X_{t-2} = i-1)\left(1 - \frac{r}{m}\right)\frac{r}{m} P(X_{t-2} = i-2)\left(\frac{r}{m}\right)^2 \\ P(X_t = i) &= P(X_{t-3} = i)\left(1 - \frac{r}{m}\right)^3 + 3P(X_{t-3} = i-1)\left(1 - \frac{r}{m}\right)^2 \frac{r}{m} \\ &\quad + 3P(X_{t-3} = i-2)\left(1 - \frac{r}{m}\right)^1 \left(\frac{r}{m}\right)^2 + P(X_{t-3} = i-3)\left(\frac{r}{m}\right)^3 \end{aligned}$$

Proceeding further, we can generalize the above,

$$P(X_t = i) = \sum_{j=0}^{k} \binom{k}{j} P(X_{t-k} = i-j)\left(1 - \frac{r}{m}\right)^{k-j}\left(\frac{r}{m}\right)^j$$

Set $k = t$ so we have the equation from time 0. $P(X_0 = 0) = 1$ and $P(X_0 = i) = 0$

for all $i \neq 0$, we have

$$P(X_t = i) = \binom{t}{i} P(X_0 = 0) \left(1 - \frac{r}{m}\right)^{t-i} \left(\frac{r}{m}\right)^i$$
$$= \binom{t}{i} \left(1 - \frac{r}{m}\right)^{t-i} \left(\frac{r}{m}\right)^i$$

The above statement says this is the binomial distribution $B(t, \frac{r}{m})$ where in t tries we have i successes with t tries with probability $\frac{r}{m}$. □

Lemma B.0.2. *For a block mapped wear leveling strategy*

$$P(X_t = k) - P(X_{t-1} = k) = r \sum_{s=1}^{m} \left[q_t^{(s)}(k-1) - q_t^{(s)}(k) \right]$$

Proof. For the block mapped strategies, we have the following set of equations:

$$P(Xi_t = k) = P(Xi_{t-1} = k) + \sum_{s=1}^{m} \left[P(Xv_{t-1}^{(s)} = k) P(C_t = s) - r\ q_t^{(s)}(k) \right]$$
$$P(Xv_t^{(s)} = k) = P(Xv_{t-1}^{(s)} = k) \left[1 - P(C_t = s)\right] \ + \ r\ q_t^{(s)}(k-1)$$

We can combine the two equations together to form the following equation:

$$P(Xi_t = k) + \sum_{s=1}^{m} P(Xv_t^{(s)} = k) =$$
$$P(Xi_{t-1} = k) + \sum_{s=1}^{m} P(Xv_{t-1}^{(s)} = k) + r \sum_{s=1}^{m} \left[q_t^{(s)}(k-1) - q_t^{(s)}(k) \right]$$

which can be rewritten as

$$P(X_t = k) = P(X_{t-1} = k) + r \sum_{s=1}^{m} \left[q_t^{(s)}(k-1) - q_t^{(s)}(k) \right]$$

□

Lemma B.0.3. *For a block mapped strategy*

$$\sum_{k=0}^{\infty}\sum_{s=1}^{m} q_t^{(s)}(k) = \frac{1}{n}$$

Proof. We have from lemma B.0.2

$$P(X_t = k) = P(X_{t-1} = k) + r\sum_{s=1}^{m}\left[q_t^{(s)}(k-1) - q_t^{(s)}(k)\right]$$

and since the expected value at time t is always $\frac{rt}{n}$, we have that

$$\sum_{k=0}^{\infty} kP(X_t = k) = \frac{rt}{n}$$

and from the above, we have that

$$\sum_{k=0}^{m} kP(X_{t-1} = k) + r\sum_{k=0}^{\infty} k\left[\sum_{s=1}^{m}\left[q_t^{(s)}(k-1) - q_t^{(s)}(k)\right]\right] = \frac{rt}{n}$$

Since

$$\sum_{k=0}^{\infty} kP(X_{t-1} = k) = \frac{r(t-1)}{n}$$

we have that

$$\sum_{k=0}^{\infty} k\left[\sum_{s=1}^{m}\left[q_t^{(s)}(k-1) - q_t^{(s)}(k)\right]\right] = \frac{1}{n}$$

or

$$\sum_{k=0}^{\infty} k\sum_{s=1}^{m} q_t^{(s)}(k-1) - \sum_{k=0}^{\infty} k\sum_{s=1}^{m} q_t^{(s)}(k) = \frac{1}{n}$$

which we can rewrite as

$$\sum_{k=1}^{\infty}(k-1)\sum_{s=1}^{m} q_t^{(s)}(k-1) + \sum_{k=1}^{\infty}\sum_{s=1}^{m} q_t^{(s)}(k-1) - \sum_{k=0}^{\infty} k\sum_{s=1}^{m} q_t^{(s)}(k) = \frac{1}{n}$$

which can be simplified to

$$\sum_{k=0}^{\infty} k \sum_{s=1}^{m} q_t^{(s)}(k) + \sum_{k=0}^{\infty} \sum_{s=1}^{m} q_t^{(s)}(k) - \sum_{k=0}^{\infty} k \sum_{s=1}^{m} q_t^{(s)}(k) = \frac{1}{n}$$

which gives us

$$\sum_{k=0}^{\infty} \sum_{s=1}^{m} q_t^{(s)}(k) = \frac{1}{n}$$

□

Lemma B.0.4. *For a block mapped strategy with sector independent block selection strategy*

$$\sum_{k=0}^{\infty} q_t(k) = \frac{1}{n}$$

Proof. We have $q_t^{(s)}(k) = q_t(k)\ P(C_t = s)$. In the property, we can simplify as

$$\begin{aligned}
\sum_{k=0}^{\infty} \sum_{s=1}^{m} q_t^{(s)}(k) &= \sum_{k=1}^{\infty} \sum_{s=1}^{m} q_t(k) P(C_t = s) \\
&= \sum_{k=0}^{\infty} q_t(k) \sum_{s=1}^{m} P(C_t = s) \\
&= \sum_{k=0}^{\infty} q_t(k) = \frac{1}{n}
\end{aligned}$$

□

Lemma B.0.5. *For $q_t(k)$ defined for the LECIF strategy*

$$\sum_{k=0}^{\infty} q_t(k) = \frac{\sum_{k=1}^{\infty} \left[(\alpha_t(k) + \beta_t(k))^n - \beta_t(k)^n \right]}{n} = \frac{1}{n}$$

Proof. Note that,

$$\sum_{k=0}^{\infty} q_t(k) = \frac{\sum_{k=1}^{\infty} \left[(\alpha_t(k) + \beta_t(k))^n - \beta_t(k)^n \right]}{n} = \frac{1}{n}$$

when

$$\sum_{k=0}^{\infty}[(\alpha_t(k)+\beta_t(k))^n-\beta_t(k)^n]=1$$

To see that the above is true, note that

$$\begin{aligned}\alpha_t(k)+\beta_t(k)&=P(Xi_{t-1}=k)+P(Xv_{t-1})+P(Xi_{t-1}>k)\\&=P(Xv_{t-1})+P(Xi_{t-1}\geq k)\\&=P(Xv_{t-1})+P(Xi_{t-1}>k-1)\\&=\beta_t(k-1)\end{aligned}$$

Thus, from above we have that

$$\begin{aligned}\sum_{k=0}^{\infty}[(\alpha_t(k)+\beta_t(k))^n-\beta_t(k)^n]&=\sum_{k=0}^{\infty}[\beta_t(k-1)^n-\beta_t(k)^n]\\&=\beta_t(-1)^n\\&=[P(Xv_{t-1})+P(Xi_{t-1}\geq 0)]^n\\&=1^n=1\end{aligned}$$

For each k, $(\alpha_t(k)+\beta_t(k))^n-\beta_t(k)^n$ represents all the blocks whose lowest invalid erase count is k. Thus, each invalid block is either k or higher. Thus, when we sum over all the values of k, we sum all the possible patterns of wear and thus, the total probability is 1. □

Lemma B.0.6. *Given that*

$$Q_t(k)=\sum_{s=1}^{k}\left[q_t^{(s)}(k-1)-q_t^{(s)}(k)\right]$$

for some wear leveling strategy, we have

$$Var(X_t)=r\sum_{\tau=1}^{t}\sum_{k=0}^{\infty}k^2Q_t(k)-\frac{(rt)^2}{m^2}$$

Proof. We have

$$\mathrm{Var}(X_t) = \mathrm{E}[X^2] - (E[X])^2 = \sum_{k=0}^{\infty} k^2 P(X_t = k) - \left(\frac{rt}{m}\right)^2$$

Also, note that

$$\left(\frac{(t-1)r}{m}\right)^2 = (t^2 - 2t + 1)\frac{r^2}{m^2} = \left(\frac{tr}{m}\right)^2 - \frac{r^2}{m^2}(2t-1)$$

Let

$$Q_t(k) = \sum_{s=1}^{k} \left[q_t^{(s)}(k-1) - q_t^{(s)}(k)\right]$$

then, we can write

$$P(X_t = k) = P(X_{t-1} = k) + rQ_t(k)$$

and, so we can write

$$\begin{aligned}\mathrm{Var}(X_t) &= \sum_{k=0}^{\infty} k^2 P(X_{t-1} = k) + r\sum_{k=0}^{\infty} k^2 Q_t(k) + \left(\frac{rt}{m}\right)^2 \\ &= \mathrm{Var}(X_{t-1}) - (2t-1)\frac{r^2}{m^2} + r\sum_{k=0}^{\infty} k^2 Q_t(k)\end{aligned}$$

Telescoping and with the fact that $\mathrm{Var}(X_0) = 0$, we have the following

$$\begin{aligned}\mathrm{Var}(X_t) &= r\sum_{\tau=1}^{t}\sum_{k=0}^{\infty} k^2 Q_\tau(k) - \frac{r^2}{m^2}\sum_{\tau=1}^{t}(2\tau - 1) \\ &= r\sum_{\tau=1}^{t}\sum_{k=0}^{\infty} k^2 Q_\tau(k) - \frac{(rt)^2}{m^2}\end{aligned}$$

□

REFERENCES

[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 57–70, Berkeley, CA, USA, 2008. USENIX Association.

[2] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: a fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 1–14, New York, NY, USA, 2009. ACM.

[3] Martin F. Arlitt and Carey L. Williamson. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Transactions on Networking*, 5:631–645, 1997.

[4] D. Atkisson and D. Flynn. Apparatus, system, and method for managing contents of a cache, March 14 2013. WO Patent App. PCT/US2012/026,790.

[5] Avraham Ben-Aroya and Sivan Toledo. Competitive analysis of flash-memory algorithms. In *ESA'06: Proceedings of the 14th conference on Annual European Symposium*, pages 100–111, London, UK, 2006. Springer-Verlag.

[6] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti. Introduction to flash memory. *Proceedings of the IEEE*, 91(4):489–502, April 2003.

[7] Roger Bitar. Deploying hybrid storage pools with sun flash technology and the solaris zfs file system. *Technical Report SUN-820-5881-10*, December 2009.

[8] Simona Boboila and Peter Desnoyers. Write endurance in flash drives: measurements and analysis. In *Proceedings of the 8th USENIX conference on File and storage technologies*, FAST'10, pages 9–9, Berkeley, CA, USA, 2010. USENIX Association.

[9] Luc Bouganim, Bjrn r Jnsson, and Philippe Bonnet. uflip: Understanding flash io patterns. In *CIDR 2009, FOURTH BIENNIAL CONFERENCE ON INNOVATIVE DATA SYSTEMS RESEARCH*. CIDR, 2009.

[10] John S Bucy, Jiri Schindler, Steven W Schlosser, and Gregory R Ganger. The disksim simulation environment version 4 . 0 reference manual. *Environment*, 2008.

[11] W. Bux, Xiao-Yu Hu, I. Iliadis, and R. Haas. Scheduling in flash-based solid-state drives - performance modeling and optimization. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, pages 459–468, 2012.

[12] Werner Bux. Performance evaluation of the write operation in flash-based solid-state drives. *IBM Technical Report*, 2009.

[13] Werner Bux. System performance issues in flash-based solid-state drives. *IBM Technical Report RZ 3801*, 2011.

[14] Werner Bux and Ilias Iliadis. Performance of greedy garbage collection in flash-based solid-state drives. *Perform. Eval.*, 67:1172–1186, November 2010.

[15] Paulo Cappelletti and Carla Golla, editors. *Flash Memories*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[16] Y. Cassuto and E. Yaakobi. Short q-ary wom codes with hot/cold write differentiation. In *Information Theory Proceedings (ISIT), 2012 IEEE International Symposium on*, pages 1391–1395, 2012.

[17] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, page 217–228, New York, NY, USA, 2009. ACM, ACM.

[18] Li-Pin Chang and Chun-Da Du. Design and implementation of an efficient wear-leveling algorithm for solid-state-disk microcontrollers. *ACM Trans. Des. Autom. Electron. Syst.*, 15:6:1–6:36, December 2009.

[19] Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *Trans. on Embedded Computing Sys.*, 3(4):837–863, 2004.

[20] Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Trans. Embed. Comput. Syst.*, 3:837–863, November 2004.

[21] Yuan-Hao Chang, Jen-Wei Hsieh, and Tei-Wei Kuo. Improving flash wear-leveling by proactively moving static data. *IEEE Trans. Comput.*, 59:53–65, January 2010.

[22] B.W. Chen. Configurable flash memory controller and method of use, October 9 2012. US Patent 8,285,921.

[23] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, pages 181–192, New York, NY, USA, 2009. ACM.

[24] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Hystor: making the best use of solid state drives in high performance storage systems. In *Proceedings of the*

international conference on Supercomputing, ICS '11, pages 22–32, New York, NY, USA, 2011. ACM.

[25] Ludmila Cherkasova and Minaxi Gupta. Characterizing locality, evolution, and life span of accesses in enterprise media server workloads. In *NOSSDAV '02: Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, pages 33–42, New York, NY, USA, 2002. ACM.

[26] F. Chierichetti, H. Finucane, Zhenming Liu, and M. Mitzenmacher. Designing floating codes for expected performance. *Information Theory, IEEE Transactions on*, 56(3):968–978, 2010.

[27] Intel Corporation. Intel solid state drive optimizer.

[28] Intel Corporation. Understanding the flash translation layer (ftl) specification.

[29] Peter Desnoyers. Empirical evaluation of nand flash memory performance. *SIGOPS Oper. Syst. Rev.*, 44(1):50–54, 2010.

[30] Peter Desnoyers. Analytic modeling of ssd write performance. In *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR '12, pages 12:1–12:10, New York, NY, USA, 2012. ACM.

[31] Cagdas Dirik. *Performance Analysis of NAND Flash Memory Solid-State Disks*. PhD thesis, University of Maryland, College Park, MD 20742-7011, 2009. `http://drum.lib.umd.edu/handle/1903/9875`.

[32] Cagdas Dirik and Bruce Jacob. The performance of pc solid-state disks (ssds) as a function of bandwidth, concurrency, device architecture, and system organization. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 279–289, New York, NY, USA, 2009. ACM.

[33] B. Eitan, R. Kazerounian, A. Roy, G. Crisenza, P. Cappelletti, and A. Modelli. Multilevel flash cells and their trade-offs. *Electron Devices Meeting, 1996., International*, pages 169–172, 8-11 Dec 1996.

[34] H. Finucane, Zhenming Liu, and M. Mitzenmacher. Designing floating codes for expected performance. In *Communication, Control, and Computing, 2008 46th Annual Allerton Conference on*, 2008.

[35] Hilary Finucane. Worst-case and average-case floating codes for flash memory. B.S. thesis.

[36] D. Flynn, D. Atkisson, and J. Aune. Apparatus, system, and method for caching data, July 16 2013. US Patent 8,489,817.

[37] D. Flynn, B. Lagerstedt, J. Strasser, J. Thatcher, J. Walker, and M. Zappe. Apparatus, system, and method for storage space recovery in solid-state storage, March 19 2013. US Patent 8,402,201.

[38] D. Flynn, B. Lagerstedt, J. Strasser, J. Thatcher, and M. Zappe. Apparatus, system, and method for managing data using a data pipeline, September 10 2013. US Patent 8,533,569.

[39] D. Flynn, D. Nellans, J.G. Peterson, J. Strasser, and R. Wipfel. Apparatus, system, and method for auto-commit memory, October 11 2012. WO Patent App. PCT/US2011/064,728.

[40] D. Flynn, J. Thatcher, J. Aune, J. Fillingim, B. Inskeep, J. Strasser, and K. Vigor. Apparatus, system, and method for managing data storage, April 2 2013. US Patent 8,412,978.

[41] D. Flynn, J. Thatcher, and M. Zappe. Apparatus, system, and method for managing concurrent storage requests, April 2 2013. US Patent 8,412,904.

[42] E. Gal and S. Toledo. Mapping structures for flash memories: techniques and open problems. *Software - Science, Technology and Engineering, 2005. Proceedings. IEEE International Conference on*, pages 83–92, 22-23 Feb. 2005.

[43] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2):138–163, 2005.

[44] Eran Gal and Sivan Toledo. Mapping structures for flash memories: Techniques and open problems. In *SWSTE '05: Proceedings of the IEEE International Conference on Software - Science, Technology & Engineering*, pages 83–92, Washington, DC, USA, 2005. IEEE Computer Society.

[45] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[46] Mahesh Balakrishnan Gokul Soundararajan, Vijayan Prabhakaran and Ted Wobber. Extending ssd lifetimes with disk-based write caches. *FAST 2010: 8th USENIX Conference on File and Storage Technologies*, February 2010.

[47] Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H. Siegel, and Jack K. Wolf. Characterizing flash memory: anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 24–33, New York, NY, USA, 2009. ACM.

[48] S. Haberman. The distributions of "s" for numerous and extensive ties in one of a pair of rankings of length n. *Review of the International Statistical Institute*, Vol. 30, No. 3 (1962), pp. 336-351.

[49] S.-W. Han. Flash memory wear leveling system and method. *US patent 6,016,275. Filed November 4, 1998; Issued January 18, 2000; Assigned to LG Semiconductors*, 2000.

[50] Sang-goo Lee Han-joon Kim. An effective flash memory manager for reliable flash memory space management. *IEICE Trans Inf Syst (Inst Electron Inf Commun Eng)*, Volume E85-D, No. 6:Page 950–964, June 2002.

[51] M. Hicken, T. Swatosh, and M. Dell. Flash memory organization, October 8 2013. US Patent 8,555,141.

[52] X.-Y. Hu and R. Haas. The fundamental limit of flash random write performance: Understanding, analysis and performance modelling. *IBM Technical Report*, 2010.

[53] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write amplification analysis in flash-based solid state drives. In *SYSTOR '09: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–9, New York, NY, USA, 2009. ACM.

[54] Xiao-Yu Hu, R. Haas, and E. Evangelos. Container marking: Combining data placement, garbage collection and wear levelling for flash. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*, pages 237–247, 2011.

[55] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Chao Ren. Exploring and exploiting the multilevel parallelism inside ssds for improved performance and endurance. *Computers, IEEE Transactions on*, 62(6):1141–1155, 2013.

[56] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Shuping Zhang. Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 96–107, New York, NY, USA, 2011. ACM.

[57] Amber Huffman and Dale Juenemann. The nonvolatile memory transformation of client storage. *Computer*, 46(8):38–44, 2013.

[58] Ilias Iliadis. Performance of the greedy garbage-colection scheme in flash-based solid-state drives. *IBM Technical Report*, 2010.

[59] Soojun Im and Dongkun Shin. Flash-aware raid techniques for dependable and high-performance flash memory ssd. *Computers, IEEE Transactions on*, 60(1):80–92, 2011.

[60] A. Jagmohan, M. Franceschini, and L. Lastras. Write amplification reduction in nand flash through multi-write coding. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–6, 2010.

[61] Sudeep Jain and Yann-Hang Lee. Real-time support of flash memory file system for embedded applications. *Software Technologies for Future Embedded and Ubiquitous Systems, and International Workshop on Collaborative Computing, Integration, and Assurance, The IEEE Workshop on*, 0:69–74, 2006.

[62] Kee-Hoon Jang and Tae Hee Han. Efficient garbage collection policy and block management method for nand flash memory. In *Mechanical and Electronics Engineering (ICMEE), 2010 2nd International Conference on*, pages V1–327 –V1–331, 2010.

[63] Anxiao Jiang. On the generalization of error-correcting wom codes. In *Information Theory, 2007. ISIT 2007. IEEE International Symposium on*, pages 1391–1395, 2007.

[64] Anxiao Jiang, V. Bohossian, and J. Bruck. Rewriting codes for joint information storage in flash memories. *Information Theory, IEEE Transactions on*, 56(10):5300–5313, 2010.

[65] Anxiao Jiang and J. Bruck. Joint coding for flash memory storage. In *Information Theory, 2008. ISIT 2008. IEEE International Symposium on*, pages 1741–1745, 2008.

[66] Anxiao Jiang and J. Bruck. Information representation and coding for flash memories. In *Communications, Computers and Signal Processing, 2009. PacRim 2009. IEEE Pacific Rim Conference on*, pages 920–925, 2009.

[67] Anxiao Jiang, M. Langberg, M. Schwartz, and J. Bruck. Universal rewriting in constrained memories. In *Information Theory, 2009. ISIT 2009. IEEE International Symposium on*, pages 1219–1223, 2009.

[68] Anxiao Jiang, Hao Li, and J. Bruck. On the capacity and programming of flash memories. *Information Theory, IEEE Transactions on*, 58(3):1549–1564, 2012.

[69] Anxiao Jiang, R. Mateescu, M. Schwartz, and J. Bruck. Rank modulation for flash memories. *Information Theory, IEEE Transactions on*, 55(6):2659–2673, 2009.

[70] Anxiao (Andrew Jiang and Vasken Bohossian. Floating codes for joint information storage in write asymmetric memories. In *Proc. IEEE International Symposium on Information Theory (ISIT*, 2007.

[71] William K. Josephson, Lars A. Bongo, Kai Li, and David Flynn. Dfs: A file system for virtualized flash storage. *Trans. Storage*, 6:14:1–14:25, September 2010.

[72] Dawoon Jung, Yoon-Hee Chae, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A group-based wear-leveling algorithm for large-capacity flash memory storage systems. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '07, pages 160–164, New York, NY, USA, 2007. ACM.

[73] Myoungsoo Jung, Ellis Herbert, Wilson Iii, David Donofrio, and John Shalf Mahmut Taylan K. Nandflashsim: Intrinsic latency variation aware nand flash memory system modeling and simulation at microarchitecture level.

[74] Myoungsoo Jung and Mahmut Kandemir. An evaluation of different page allocation strategies on high-speed ssds. In *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[75] Dongwon Kang, Dawoon Jung, Jeong-Uk Kang, and Jin-Soo Kim. mu-tree: an ordered index structure for nand flash memory. In *EMSOFT*, pages 144–153, 2007.

[76] Woochul Kang, Sang H. Son, John A. Stankovic, and Mehdi Amirijoo. I/o-aware deadline miss ratio management in real-time embedded databases. *Real-Time Systems Symposium, IEEE International*, 0:277–287, 2007.

[77] Taeho Kgil and Trevor Mudge. Flashcache: a nand flash memory file cache for low power web servers. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 103–112, New York, NY, USA, 2006. ACM.

[78] Dong Kim, Kwanhu Bang, Seung-Hwan Ha, Sungroh Yoon, and Eui-Young Chung. Architecture exploration of high-performance pcs with a solid-state disk. *Computers, IEEE Transactions on*, 59(7):878–890, 2010.

[79] Han-joon Kim and Sang-goo Lee. A new flash memory management for flash storage system. In *23rd International Computer Software and Applications Conference*, COMPSAC '99, pages 284–, Washington, DC, USA, 1999. IEEE Computer Society.

[80] Hyojun Kim and Seongjun Ahn. Bplru: A buffer management scheme for improving random writes in flash storage. *FAST'08: Proceedings of the 6th conference on USENIX Conference on File and Storage Technologies*, pages 239–252, 2008.

[81] Jaehong Kim, Sangwon Seo, Dawoon Jung, Jin-Soo Kim, and Jaehyuk Huh. Parameter-aware i/o management for solid state disks (ssds). *Computers, IEEE Transactions on*, 61(5):636–649, 2012.

[82] Youngjae Kim, Brendan Tauras, Aayush Gupta, and Bhuvan Urgaonkar. Flashsim: A simulator for nand flash-based solid-state drives. *Advances in System Simulation, International Conference on*, 0:125–131, 2009.

[83] Ioannis Koltsidas and Stratis D. Viglas. Flashing up the storage layer. *Proc. VLDB Endow.*, 1(1):514–525, 2008.

[84] Hunki Kwon, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Variable space page mapping method and apparatus for flash memory device. *US patent 20100082886. Filed 10/01/2009; Issued 04/01/2010*, 2010.

[85] Sang-Won Lee and Bongki Moon. Design of flash-based dbms: an in-page logging approach. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 55–66, New York, NY, USA, 2007. ACM.

[86] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory ssd in enterprise database applications. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1075–1086, New York, NY, USA, 2008. ACM.

[87] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of ethernet traffic (extended version). *IEEE/ACM Trans. Netw.*, 2(1):1–15, 1994.

[88] Yan Li and Khandker N. Quader. Nand flash memory: Challenges and opportunities. *Computer*, 46(8):23–29, 2013.

[89] Yinan Li, Bingsheng He, Qiong Luo, and Ke Yi. Tree indexing on flash disks. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 1303–1306, Washington, DC, USA, 2009. IEEE Computer Society.

[90] Xiang Luojie and B.M. Kurkoski. An improved analytic expression for write amplification in nand flash. In *Computing, Networking and Communications (ICNC), 2012 International Conference on*, pages 497–501, 2012.

[91] M-Systems. Two technologies compared: Nor vs. nand. `http://www.dataio.com/pdf/NAND/MSystems/MSystems_NOR_vs_NAND.pdf`, 2003.

[92] Hessam Mahdavifar, Paul H. Siegel, Alexander Vardy, Jack K. Wolf, and Eitan Yaakobi. A nearly optimal construction of flash codes. In *Proceedings of the 2009 IEEE international conference on Symposium on Information Theory - Volume 2*, ISIT'09, pages 1239–1243, Piscataway, NJ, USA, 2009. IEEE Press.

[93] U. Maheshwari. Flash memory cache for data storage device, October 9 2012. US Patent 8,285,918.

[94] Charles Manning. Yet another flash file system. `http://www.yaffs.net/`, 2004.

[95] Micron Technology Inc. *64Gb, 128Gb, 256Gb, 512Gb Asynchronous/Synchronous NAND Features*, 2011. NAND Flash Memory.

[96] Vidyabhushan Mohan, Taniya Siddiqua, Sudhanva Gurumurthi, and Mircea R. Stan. How i learned to stop worrying and love flash endurance. In *Proceedings of the 2nd USENIX conference on Hot topics in storage and file systems*, HotStorage'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[97] M. Murin and E. Sharon. Ad hoc flash memory reference cells, November 27 2012. US Patent 8,321,623.

[98] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating server storage to ssds: analysis of tradeoffs. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 145–158, New York, NY, USA, 2009. ACM.

[99] H. Nobukata, S. Takagi, K. Hiraga, T. Ohgishi, M. Miyashita, K. Kamimura, S. Hiramatsu, K. Sakai, T. Ishida, H. Arakawa, M. Itoh, I. Naiki, and M. Noda. A 144-mb, eight-level nand flash memory with optimized pulsewidth programming. *Solid-State Circuits, IEEE Journal of*, 35(5):682–690, May 2000.

[100] A. Nozoe, H. Kotani, T. Tsujikawa, K. Yoshida, K. Furusawa, M. Kato, T. Nishimoto, H. Kume, H. Kurata, N. Miyamoto, S. Kubono, M. Kanamitsu, K. Koda, T. Nakayama, Y. Kouro, A. Hosogane, N. Ajika, and K. Koyashi. A 256-mb multilevel flash memory with 2-mb/s program rate for mass storage applications. *Solid-State Circuits, IEEE Journal of*, 34(11):1544–1550, Nov 1999.

[101] A.K. Olbrich and D.A. Prins. Flash memory controller garbage collection operations performed independently in multiple flash memory groups, September 10 2013. US Patent 8,533,384.

[102] C. Park, Euiseong Seo, Ji-Yong Shin, Seungryoul Maeng, and Joonwon Lee. Exploiting internal parallelism of flash-based ssds. *Computer Architecture Letters*, 9(1):9–12, 2010.

[103] S. Park and Kai Shen. A performance evaluation of scientific i/o workloads on flash-based ssds. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1 –5, 2009.

[104] Seung-Ho Park, Jung-Wook Park, Shin-Dug Kim, and C.C. Weems. A pattern adaptive nand flash memory storage structure. *Computers, IEEE Transactions on*, 61(1):134–138, 2012.

[105] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. Transactional flash. In *In Proc. Symposium on Operating Systems Design and Implementation (OSDI*, 2008.

[106] Abhishek Rajimwale, Vijayan Prabhakaran, and John D. Davis. Block management in solid-state devices. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 21–21, Berkeley, CA, USA, 2009. USENIX Association.

[107] Marcus Marrow Rajiv Agarwal. A closed-form expression for write amplification in nand flash. *ACTEMT Workshop on Application of Communication Theory to Emerging Memory Technologies*, 2010.

[108] Ronald L. Rivest and Adi Shamir. How to reuse a ẅrite - once m̈emory. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, STOC '82, pages 105–113, New York, NY, USA, 1982. ACM.

[109] David Roberts, Taeho Kgil, and Trevor Mudge. Integrating nand flash devices onto servers. *Commun. ACM*, 52(4):98–103, 2009.

[110] Samsung Electronics. *512M x 8Bit / 1G x 8Bit NAND Flash Memory.*, 2007. K9W8G081M/K9K4G08U0M Flash Memory Datasheet.

[111] E. Seppanen, M.T. O'Keefe, and D.J. Lilja. High performance solid state storage under linux. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1 –12, May 2010.

[112] M. Shrestha and Lihao Xu. A quantitative framework for modeling and analyzing flash memory wear leveling algorithms. In *GLOBECOM Workshops (GC Wkshps), 2010 IEEE*, pages 1836 –1840, dec. 2010.

[113] Eitan Yaakobi Alexander Vardy Paul H. Siegel and Jack K. Wolf. Multidimensional flash codes. *Proc. 46-th Annual Allerton Conference on Communication, Control and Computing*, September 2008.

[114] SNIA. Storage networking industry association: Iotta repository home : Microsoft production server traces. `http://iotta.snia.org/traces/158`, 2008.

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

[116] J. Thatcher, D. Flynn, E. Barnes, J. Strasser, R. Wood, and M. Zappe. Apparatus, system, and method for using multi-level cell solid-state storage as single level cell solid-state storage, May 14 2013. US Patent 8,443,259.

[117] UMass. Umass trace repository. `http://traces.cs.umass.edu/index.php/Storage/Storage`, 2002.

[118] David Woodhouse. Jffs: The journalling flash file system. In *Ottawa Linux Symposium*. RedHat Inc., 2001.

[119] R. Haas X.-Y. Hu. The fundamental limit of flash random write performance: Understanding, analysis and performance modelling. In *IBM Technical Paper RZ3771*, 2010.

[120] E. Yaakobi, S. Kayser, P.H. Siegel, A. Vardy, and J.K. Wolf. Efficient two-write wom-codes. In *Information Theory Workshop (ITW), 2010 IEEE*, pages 1–5, 2010.

[121] E. Yaakobi, S. Kayser, P.H. Siegel, A. Vardy, and J.K. Wolf. Codes for write-once memories. *Information Theory, IEEE Transactions on*, 58(9):5985–5999, 2012.

[122] Eitan Yaakobi, Hessam Mahdavifar, Paul H. Siegel, Alexander Vardy, and Jack K. Wolf. Rewriting codes for flash memories. *CoRR*, abs/1210.7515, 2012.

[123] Hong Yang, Hyunjae Kim, Sung il Park, Jongseob Kim, Sung-Hoon Lee, Jung-Ki Choi, Duhyun Hwang, Chulsung Kim, Mincheol Park, Keun-Ho Lee, Young-Kwan Park, Jai Kwang Shin, and Jeong-Taek Kong. Reliability issues and models of sub-90nm nand flash memory cells. In *Solid-State and Integrated Circuit Technology, 2006. ICSICT '06. 8th International Conference on*, pages 760 –762, 2006.

[124] Weimin Zheng Youyou Lu, Jiwu Shu. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, 2013.

[125] H. ZHONG, Y. Li, R. Danilak, and T. Cohen. Ldpc erasure decoding for flash memories, September 15 2011. WO Patent App. PCT/US2011/028,244.

ABSTRACT

STUDY ON ENDURANCE OF FLASH MEMORY SSDS

by

MOCHAN SHRESTHA

May 2014

Advisor: Dr. Lihao Xu

Major: Computer Science

Degree: Doctor of Philosophy

Flash memory promises to revolutionize storage systems because of its massive performance gains, ruggedness, large decrease in power usage and physical space requirements, but it is not a direct replacement for magnetic hard disks. Flash memory possesses fundamentally different characteristics and in order to fully utilize the positive aspects of flash memory, we must engineer around its unique limitations. The primary limitations are lack of in-place updates, the asymmetry between the sizes of the write and erase operations, and the limited endurance of flash memory cells. This leads to the need for efficient methods for block cleaning, combating write amplification and performing wear leveling. These are fundamental attributes of flash memory and will always need to be understood and efficiently managed to produce an efficient and high performance storage system.

Our goal in this work is to provide analysis and algorithms for efficiently managing data storage for endurance in flash memory. We present update codes, a class of floating codes, which encodes data updates as flash memory cell increments that results in reduced block erases and longer lifespan of flash memory, and provides a new algorithm for constructing optimal floating codes. We also analyze the theoretically possible limits

of write amplification reduction and minimization by using offline workloads. We give an estimation of the minimal write amplification by a workload decomposition algorithm and find that write amplification can be pushed to zero with relatively low over-provisioning. Additionally, we give simple, efficient and practical algorithms that are effective in reducing write amplification and performing wear leveling. Finally, we present a quantitative model of wear levels in flash memory by constructing a difference equation that gives erase counts of a block with workload, wear leveling strategy and SSD configuration as parameters.

AUTOBIOGRAPHICAL STATEMENT

Mochan Shrestha received his M.S. in Mathematics from Michigan State University, East Lansing, MI and his B.S. in Computer Science from Grand Valley State University, Allendale, MI. During the PhD program, he was a member of Network and Information Systems Lab (NISL) in the Department of Computer Science, Wayne State Universeity, Detroit MI.

His research interests are in the areas of flash memory, coding theory and storage systems.