EVALUATION OF LOG-BASED BLOCK AND WEAR-LEVELING TECHNIQUES

A Thesis in
Computer Science and Engineering
by
John Joseph Niedzwiecki II

© 2008 John Joseph Niedzwiecki II

Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science

August 2008
The thesis of John Joseph Niedzwiecki II was reviewed and approved* by the following:

Wang-Chien Lee  
Associate Professor of Computer Science and Engineering  
Thesis Adviser

Dongwon Lee  
Assistant Professor of Information Sciences and Technology

Raj Acharya  
Professor of Computer Science and Engineering  
Head of Department of Computer Science and Engineering

*Signatures are on file in the Graduate School
ABSTRACT

Flash memory, a nonvolatile memory for external data storage, is growing in popularity of use, largely in portable devices. Because of the properties of this memory, it must be treated and interacted with differently than magnetic disks. The largest two properties causing issues are that portion of memory must be erased before it can be written to again, and that these portions can only operate up to a limited number of erasures, after which the memory becomes unreliable. Special algorithms are needed for dealing with flash memories. These policies deal primarily with leveling the wear (or number of erasures) of the memory and efficient reclamation of blocks so they can be written to again. This thesis begins by examining many of these algorithms and policies for handling flash memory. Next, a simulator is developed and used to evaluate several of these algorithms. With this simulator, higher level programs can be tested for their performances on various flash memory techniques and trends. Based on the findings from the evaluation, these higher level algorithms can then be optimized to perform better on flash memory devices.
# TABLE OF CONTENTS

List of Figures .................................................................................................................... v
List of Abbreviations ........................................................................................................ vi
Acknowledgements ......................................................................................................... vii

Chapter 1: INTRODUCTION .......................................................................................... 1

Chapter 2: ALGORITHMS AND POLICIES ................................................................. 4
  Background ..................................................................................................................... 4
  Wear-Centric Reclamation Policies .............................................................................. 9
  Efficient Reclamation Policies ...................................................................................... 17
  Log-Structured File Systems ......................................................................................... 20

Chapter 3: RESEARCH GOALS AND METHODOLOGY .......................................... 26

Chapter 4: SIMULATOR ............................................................................................... 30
  Environment .................................................................................................................. 30
  Implementation ............................................................................................................. 30
  Algorithms Simulated .................................................................................................... 30
  Simulator Operation ...................................................................................................... 32

Chapter 5: RESULTS .................................................................................................... 36

Chapter 6: CONCLUSIONS ......................................................................................... 44

Bibliography .................................................................................................................... 45

Appendix .......................................................................................................................... 47
LIST OF FIGURES

Figure 1 - Example of Direct Map...................................................................................................... 5
Figure 2 - Simple Erase Counter Wear-Leveling Relocation........................................................... 10
Figure 3 - Hot Data Identification with Hashing and Column Major Structure............................ 14
Figure 4 - Example of Group and Averages..................................................................................... 15
Figure 5 - Clustering of Read-only and Hot Data........................................................................... 19
Figure 6 - Sequential and Random Write Blocks.......................................................................... 24
Figure 7 - Design of Simulator........................................................................................................ 32
Figure 8 - Simulator Step 1............................................................................................................. 33
Figure 9 - Simulator Step 2............................................................................................................. 34
Figure 10 - Simulator Step 3.......................................................................................................... 34
Figure 11 - Simulator Step 4........................................................................................................... 35
Figure 12 - Simulator Step 5........................................................................................................... 35
Figure 13 - Effect of Threshold on Average Erase Count and Standard Deviation for Simple Erase Counter.................................................................................................................. 38
Figure 14 - Average Erase Count of MP3 Trace............................................................................. 39
Figure 15 - Standard Deviation of MP3 Trace ............................................................................. 39
Figure 16 - Simple Erase Counter Example .................................................................................... 40
Figure 17 - FAST Example .............................................................................................................. 40
Figure 18 - Average Erase Count of MOV Trace.......................................................................... 41
Figure 19 - Standard Deviation of MOV Trace ............................................................................ 41
Figure 20 - Average Erase Count of PIC Trace.............................................................................. 42
Figure 21 - Standard Deviation of PIC Trace.................................................................................. 42
Figure 22 - Average Erase Count for PMP Trace.......................................................................... 43
Figure 23 - Standard Deviation for PMP Trace............................................................................. 43
LIST OF ABBREVIATIONS

BAST: Block Associative Sector Translation

BET: Block Erasing Table

CAT: Cost Age Time

FAST: Fully Associative Sector Translation

FTL: Flash Translation Layer

JFFS: Journaling Flash Filing System

lba: Logical Block Address

lbn: Logical Block Number

lsn: Logical Sector Number

MLC: Multiple Level Cell

NFTL: NAND Translation Layer

pbn: Physical Block Number

RAM: Random Access Memory

SLC: Single Level Cell

YAFFS: Yet Another Flash Filing System
ACKNOWLEDGEMENTS

I would like to acknowledge Dr. Dawoon Jung, from the Korea Advanced Institute of Science and Technology (KAIST). I would like to thank Dr. Jung for allowing the use of the traces of portable devices in my research. These traces are captured directly from portable devices using flash memory and are a valuable asset for testing purposes.
Chapter 1: INTRODUCTION

Flash memory has gained popularity in the past several years. This is largely the result of its properties. Flash memory is a nonvolatile memory, meaning it can retain its contents without the need for power. It also is small in size, low power consumption and has a low cost. These factors, among others, make flash memory convenient and ideal for portable devices. The popularity of these devices, such as mp3 players, portable media players, digital cameras and cell phone, has fueled the momentum of flash memory.

Flash memory is not without problems. First, that data cannot be modified in place. If you wish to rewrite or save a file again over its current location, the memory units would need to be erased first before they can be written to again. Second, flash memory can only be written for a limited number of times. After a certain point, the memory units become unreliable and can no longer be used to store data. Newer flash memory, while lower in cost, also can withstand less writes before becoming unstable (Chang, Hsieh and Kuo).

Due to these properties, new algorithms are needed to interact with the flash memory. Traditional algorithms used on magnetic disk memory are not appropriate. The new algorithms have to support writes that cannot be done in place. They also look to level the wear (number of erasures) and efficiently reclaim pages of invalid data to write to.

This thesis evaluates several different flash memory techniques, each with their own strengths in memory management. Through this evaluation, the research can help flash devices. It is hard to know how a flash memory will react to various programs that could run on these devices. This is not as simple to evaluate as with a traditional
magnetic disk hard drive. Seeing how the program affects the wear of the memory would be a very useful ability. This thesis develops a simulator with the ability to simulate different implementations of the memory and provide feedback of the status of the memory. The trends of various workloads can be analyzed against simulation results. These statistics can then be used to optimize the program for flash memory, instead of developing completely new implementations for the memory. Most papers researched in the area of flash memory have been of new techniques. These techniques are tested against others on various datasets and traces. However that is where their work stops. With the simulator proposed in this thesis, programs have the ability to test themselves on these implementations, so they can be optimized, instead of the implementations.

This thesis contributes several things:

- Develops a simulator to test flash memory implementations and memory types.
- Examines traces of several common portable devices that utilize flash memory, analyzing for workload and memory access trends.
- Analyzes the result of simulation of these traces on different memories, providing valuable information on the result of the trends effects on the memories.
- Provide a future testing environment for the optimization of programs running on flash memory.

The rest of the thesis is arranged as follows. In Chapter 2, the algorithms and policies for interacting are examined. A brief background to the nature of flash memory is examined followed by the three main classes of algorithms: wear-centric, efficient
reclamation and log-based. Individual policies are examined and compared. The research goals and methodologies are presented in Chapter 3. Next, in Chapter 4, a simulator developed to test the techniques is examined. Its design environment, implementation, algorithms implemented and the operation of the simulator are all detailed. Next, the results are examined in Chapter 5. The results of the simulator are also contrasted with those from other papers and tests. Finally, conclusions of the work are presented in Chapter 5.
Chapter 2: ALGORITHMS AND POLICIES

2.1. Background

Flash memory is becoming increasingly more popular and deployed in many devices due to its properties. Flash memory has small size, low power consumption, is shock resistant and is a non-volatile memory, making it ideal for mobile devices (Lee, Park and Chung). Though flash memory does not have the mechanical delay of hard disks (i.e. seek time and rotational latency), it does have a bottleneck due to writes. Writes operate a much slower speed than reads.

A common way to map the data on flash memory is through blocks. The main idea is to map the host block number, or logical block number, to a physical flash address, or the sector (Gal and Toledo). In flash memory, new data cannot overwrite the sector where the block is currently stored. Blocks must be erased before it can be written again. Instead, the data is written to another sector, and the map used for translating logical blocks to sectors is updated to the new sector (Gal and Toledo). The mapping can be done in several ways. The first of which are direct maps. In direct maps, arrays store in the $i$th location of the array the index of the sector currently containing block $i$ (Gal and Toledo). These are efficient at mapping blocks to sectors. They are also at least partially stored in RAM, making them able to support fast lookups. A partial example of this can be seen in Figure 1. The able on the right represents the direct map. In the table, we want to map lbn 2 to its physical location. This is done by examining its location in the table. In that location a value of 7 is found giving the sector number. This sector corresponds linearly to the second sector in the second block, or sector 1 in block 1. The presented example of a direct map shows an arrow diagramming this mapping. This directly maps
the logical block to the physical block. Another way of mapping is through inverse maps.
These maps store in the $i$th location of the array or table the identity of the block stored in the $i$th sector (Gal and Toledo). These efficiently map sectors to blocks, thus making them useful at reconstructing the direct map during the initialization of the flash device.
These are stored on the flash device, keeping the identity in the same erase unit. Another way to look at the mapping is the level the table operates on. The mapping of logical to physical address can be managed at several levels: sector-level mapping, block-level mapping or a hybrid scheme. Sector-level mapping maps the logical sector to any physical sector. This design is very flexible; however the size of the mapping table would be too large to be viable in practice. For example, in a 1 GB memory, with 512B sectors, there would be 2 million sectors to map (Lee, Park and Chung). The other option is block-level mapping, where the logical sector address consists of a logical block number and an offset. This mapping table only contains a map between the logical and physical blocks, and then uses the same offset (Lee, Park and Chung). This makes the table very small in comparison. However, this can lead to collisions in the physical block sector. When the same sector is needed for another write, the other sectors within the block (also known as an erase unit) need to be copied first to a new physical block, and the original
erased before the write can be performed. This is known as the erase-before-write problem, causing a performance bottleneck (Lee, Park and Chung). This can be addressed by a hybrid scheme. In addition to having the block-level mapping table, it contains a sector-level mapping table for a limited number of blocks. This keeps the size small, while helping to mitigate the erase-before-write problem (Lee, Park and Chung).

Flash memory comes in two main types: NOR and NAND. NOR memory is well suited for code storage and devices that use execute-in-place (XIP) applications (Park, Lim and Kwon). This makes NOR memory ideal for uses such as the operating system of a cell phone or PDA. NAND flash memory uses more sequential access with a longer latency, making it less applicable to XIP applications. These characteristics do however make it useful for data storage, along with its high density organization of cells (Park, Lim and Kwon). This makes NAND memory more popular and suitable for storage on portable devices and for use in memory cards.

Within flash memory, reliability is an important issue. The reliability is directly related to the endurance of the memory itself. This is becoming more challenging, as low-cost flash-memory designs are gaining momentum in the market. The lower cost MLC_{x2} (Multiple Level Cell) flash is only guaranteed up to 10,000 erase counts. These hold n-bit information per cell, in MLC_{x2}, n being 2 (Chang, Hsieh and Kuo). This is done through multiple voltage thresholds. If the memory cell can be charged reliably to multiple voltages, then more than 1 bit can be represented (L.-P. Chang). In traditional SLC (Single Level Cell) flash memory, the memory handles 100,000 erase counts, but represents data with a binary value, therefore only 1 bit of data (L.-P. Chang).
Flash memory is made up of many blocks (or erase units) each of which contains a fixed number of sectors (or pages). Blocks are the smallest units that can be erased. Individual sectors can only be read and written. For example, in NAND flash memory chips, SLCs can be of a smaller or larger size. The small size has 512B sectors for the data with 32 sectors per block. On the large size, the SLCs can contain 2KB sectors for data with 64 sectors per block. MLCs are the same as large SLCs. However, they contain 128 sectors per block (Chang, Hsieh and Kuo).

While flash memory often operates as blocks, the system often views the flash memory as a hard-disk-like block device. The system instead translates the commands into low-level operations (read, write, erase) using the physical sector addresses (Lee, Park and Chung). This translation is most commonly done utilizing the Flash Translation Layer (FTL) originally designed by Ban (A. Ban, Flash File System). The FTL uses a technique to store some of the direct map within the flash device. The block number is mapped to the logical block number, containing the logical erase unit number and sector index within the erase unit. This block-to-logical-block map is stored in RAM and flash. A two-level hierarchical structure is used. First the system looks up the sector containing the map in the RAM map. Then, it retrieves the map from flash (not stored contiguously in flash) (Gal and Toledo). Logical erase units are also mapped to physical erase units using a small direct map in the RAM.

A similar layer is used in NAND flash memory, the NAND Translation Layer (NFTL), again designed by Ban (A. Ban). The NFTL can operate in two manners: with or without space for spare data in each sector. Without the spare data space, the virtual block number is broken into a logical erase-unit number and the sector number within the
erase unit. It maps each logical erase unit to a chain of physical units. To find the proper data, the appropriate chain (of length 1 or more) is searched (Gal and Toledo). When spare data is available, chains are always of length 1 or 2. The first unit is the primary unit, while the second unit in the chain is the replacement unit. The replacement unit can contain many copies of the same virtual block; however only one is valid (Gal and Toledo).

One of the key operations in flash memories is block (erase-unit) reclamation. This is done to reclaim obsolete sectors or in the erase-before-write scenarios. The reclamation is done in the unit of blocks, i.e. the full erase-unit, spanning multiple sectors. This is also known as garbage collection. The reclamation has several stages. First the block is selected for reclamation. Then, the valid sectors of this block are copied to newly allocated free space (another block). Then, the mapping information is updated to the new block and sectors. Then, the reclaimed block is erased and the block is added to a free-block list (or free-sector reserve) (Gal and Toledo). The system keeps at least one or two free blocks at all times for valid data during reclamation. The reclamation process is ruled by two policies, 1) which blocks to reclaim and 2) where to relocate valid sectors to, which is similar to sector allocation during block updates. These two policies, with the sector allocation policy, affect the effectiveness (number of obsolete sectors in reclaimed blocks), wear leveling and mapping data structures (Gal and Toledo). This brings up some key issues in flash memories, e.g., wear leveling and the tradeoff between efficient reclamation. Wear leveling key concept is to extend the life of the flash memory by evenly distributing the wear, measured in erase counts, across all the blocks. Efficient reclamation however concentrates on the effectiveness of the reclamation of blocks. This
makes the two goals contradictory. Take for example static data, efficient reclamation says it should not be reclaimed because it will not free up any new storage. However, reclaiming blocks with static data can reduce the wear on other units, therefore leveling the wear (Gal and Toledo).

These goals lead to different designs in handling flash memory. Most techniques implement each goal to some extent; however, a compromise must be made on one to achieve the other to a larger extent. This leads to two main types of policies, wear-centric policies and efficient reclamation policies. There is as well a third type, log-structured systems, that operate slightly different, which will be discussed later. Many of these policies are not necessarily full panacea solutions, but are however particular solutions to individual ideas that can be combined into different base systems.

2.2. Wear-Centric Reclamation Policies

Wear leveling can be very important to flash memories. With limited lifespan of the flash cells, based upon the erase count, leveling the wear across the whole memory is important. The nature of memory activity shows why some form of wear-leveling is needed. Traces of memory show that 2/3 of the writes are of no more than 8 sectors. And while they are over 10% of the written data, they span only 1% of the address space (Hsieh and Kuo). To put it in simple terms, only a small fraction of the address space for a disk is frequently referenced. Also, writes of a small amount of the data often go with frequently accessed data. There is a clear motivation to level the wear among the memory cells.
A simple erase counter method was patented by Lofgren et al (Lofgren, D. and Thelin). In this method, the head of each erase unit includes a counter. Also, one erase unit is set aside as a spare. When a unit is reclaimed, the counter is compared to the least worn-out unit. If this difference exceeds a threshold (i.e. 500) wear-leveling relocation is used, as demonstrated by Figure 2. The first set of blocks, represents a starting set, Block 3, the block to be reclaimed, Block 45 the extra block and Block 12 the least worn-out block. If the threshold is exceeded, the least worn-out unit is copied to the spare, as demonstrated by the second set of blocks. Then, the most worn out unit is copied to the just erased least worn-out unit. The most worn out unit, which was just reclaimed becomes the new spare, demonstrated by the last set of blocks. This simple method attempts to identify worn-out sectors and static blocks, relocating the static blocks to the worn out sectors. This method is a fairly simple as well as accurate method of leveling.
well. With a small threshold, the standard deviation, a strong method for identifying wear leveling, can be kept small. However the smaller the threshold, the more erases overall will be performed. Due to the copying and moving of data in the wear-leveling reclamation, two erases are performed instead of one for normal reclamation. This method also runs a risk of losing the counter value.

A method such as this does run the risk of losing the counter during power loss. There are several methods proposed to counteract this risk. One example of this is to store the erase counter of the unit $i$ on another unit $j$ where $j \neq i$ (Marshall and Manning). This method, proposed as part of a file system, copies the counter to the specially-marked area in the arbitrary unit $j$. If the power is lost, the counter can be recovered from $j$. While being more complicated for storing and updating counters, it would be a safer method of storing it. However updating the counter causes an update in another unit. Another proposal to solve this problem is a bounded unary 8-bit erase counter, patented by Assar et al (Assar, Nemazie and Estakhri). In this method, the counter is stored in another erase unit, and its starts as all ones. Every time the block is erased, a bit is cleared. This method provides a counter that can be updated in place, which therefore does not need to be erased every time. However, erases beyond 8 cause a loss in the accuracy of this counter.

Another simple method looks to use a spare unit with randomness to level the wear. This method patented by Ban states that for every certain number of reclamations, an erase unit is selected at random. The contents are relocated to a spare unit, and the current unit is marked as a spare. This can be deterministic or have a random trigger. The aim is for every unit to undergo a fairly large number of random swaps during the lifetime of the device (A. Ban). This would diminish the chances of a unit storing static
data for much of the lifetime. It also would spread any overhead evenly over the lifetime of the device. Over a large lifetime of the device, or a short trigger between random reclamations, this would cause most units to be relocated. By this manner, static data would be moved around. However, the number of swaps static data will incur equals to the number of swaps hot data will incur due to the random reclamations. This method, while providing some wear-leveling, would seem to be less effective by nature.

Jou and Jeppensen propose an erase-before-write method, which maintains an upper bound on wear by the number of erasures (Jou and Jeppesen III). The bound is proven always correct however not necessarily tight. When an erase unit is to be reclaimed, the valid contents are copied to another unit, which are not immediately erased. Instead, they are marked as an erasure candidate and added to a queue of candidates. The queue is sorted by wear, with the least worn out first. This levels the wear some by delaying the reuses of worn-out units, depending on the amount of surplus units. By having enough extra units, worn-out units will be at the end of the queue. When a unit with less erases is placed into the queue, it will supersede the worn-out unit, thus delaying its erase and help leveling the wear. This method however does not do anything to ensure low wear units are reclaimed. This method only helps delay high wear units from erases.

A method proposed by Han attempts to estimate the wear using the erase latencies (Han). The idea is that on some devices, erase latency increase with the more wear on the block. It compares the erase times in order to rank the blocks. This avoids the need to store counters for each block. This estimation can then be used in a wear leveling relocation policy. This method can only estimate after the block as been erased in the
session, therefore would not be useful in a device when the sessions will be short. This method is also only applicable to such devices where the latency will increase with wear, making it very limited in use.

Another method is proposed by Chang et al. They propose an efficient static wear leveling design (Chang, Hsieh and Kuo). The design is motivated by preventing cold data from staying in any block for a long period of time, which would in turn minimize the maximum erase-count difference of any two blocks. They propose a modular design to be integrated into existing implementations (i.e. NTL, NFTL). It uses a Static Wear (SW) Leveler associated with a Block Erasing Table (BET). The BET remembers which blocks have been erased in a selected period of time. It is built as a bit array, with each bit corresponding to $2^k$ contiguous blocks. When a block is erased, the bit is set to 1. This is used to locate blocks of cold data. The SW Leveler has 2 procedures, one invoked to update the BET, and the other SWL-Procedure when static wear-leveling is needed. This is invoked with the ratio of the number of 1s in the BET and the total number of block erases since the last reset is greater or equal to a determined threshold. The main memory space overhead depends on the size of the memory storage system and the size of k (i.e. 1GB, k=2, 256B or k=3, 128B). With the right value of k, the table is small in relative size. It also achieves its results with little change to current implementations of FTLs and NFTLs.

Hsieh and Kuo proposed a new method for identifying hot data (Hsieh and Kuo). This method was proposed to be a highly efficient method through hash functions. Many other methods carry either large memory or computation overhead in order to track hot (frequently accessed) data. This method uses a new multihash-function framework. In
this framework, it adopts K independent hash functions to hash a given logical block address (lba) into multiple entries of an m-entry hash table which tracks the write number of each LBA. Each entry is associated with a counter of C bits. When a write is issued, the corresponding lba is hashed simultaneously by K hash functions, and each counter is incremented by one. This is demonstrated in the left side of Figure 3, created from figures and descriptions in (Hsieh and Kuo). As well as incrementing, for every given number of sectors being written (a decay period), the values of all counters are divided by two, done by shifting right. This acts as an aging mechanism to exponentially decay the values of all the write numbers over time. To verify if the data is hot in a given lba, the lba is hashed and it considered hot if the H most significant bits of every counter based on the K hash functions contain a nonzero bit value. The counters are also safe from overflow by either freezing a counter when it reaches its maximum positive value or right shifting all counters whenever a counter would be overflowed. The hash functions can be a various number of functions such as the division method, the multiplication method or a newer locality-based hash function discussed in (Hsieh and Kuo). Finally, this storage of
counters in this method is implemented in a column major hash table, as shown in the right side of Figure 3. Each gray or white area represents a different counter. The general idea of this design is to lessen the cost of the exponential decaying of counters, which could be proportional to the table size. In a column major structure, the counters run “up and down” spanning multiple rows, but just one bit wide. Therefore, in a 32-bit array, 32 would fit across instead of just 8, but also make for an efficient implementation of the decay, by making zeroing a row and shifting simple, done across many counters (32 in the example) at once (Hsieh and Kuo). In Figure 3, counters are 5 bits long, and 4 counters fit across. The counter from the left-most top row would be 00001. While this method does not keep as accurate a count on the level of wear for each individual block, it provides a quick and fairly small (memory wise) method. By using computationally simple enough hash functions and the column major structure, calculating the hashes and decaying the values can be fast. The tradeoff is with the less accurate count which can lead to false identifications.

Jung et al proposed a group-based wear-leveling algorithm (Jung, Chae and Jo). They look to achieve a small memory footprint by grouping several logically sequential blocks and managing only summary information for the groups. This algorithm takes a
proposed effective group summary structure. A common approach to do so would be to maintain a cleaning index for each block. This information could then be used to move hot data to less worn blocks. This would require memory space proportional to the number of blocks. The main purpose of the group-based algorithm is to reduce the memory requirement for storing wear information, which then can be used in hold-cold swapping. The group consists of a fixed number of contiguous logical blocks and group summary to represent the overall wear. The wear-leveling ability can only be as good as the accuracy of the summary information. The algorithm maintains two averages, the total average as the average erase cycles for all blocks and the partial average as the average erase cycle for only those not swapped yet. This can be seen in Figure 4. The total average, denoted as tAvg shows the average over all the blocks (124, 97, 18 and 19) is 130, while the partial average, denoted as pAvg, over the not yet swapped blocks is only 97 for blocks 18 and 19. Using just the total average is not accurate enough. To check if a swap is needed, the total average is compared with the erase cycle of the newly allocated update block, and the difference is compared to a threshold. If it is necessary, the youngest group with the lowest partial average is chosen. Then, within the group, a round-robin manner is used. This method provides roughly the same wear-leveling performance as other schemes with less than 8.75% of the memory space. The group size greatly affects the accuracy of the amounts and thus the accuracy of wear-leveling. With storing only partial information for several blocks, this method does provide efficiency in this area.
2.3. Efficient Reclamation Policies

There are several policies that look more towards efficient reclamation. These methods generally trade off some wear-leveling ability for recovering more sectors. One of such methods is a block device driver patented by Kawaguchi et al (Kawaguchi, Nishioka and Motoda). It was originally intended for log-structured UNIX file system. However, it is applicable to flash memory. It operates with two reclamation policies. The first selects the next unit for reclamation based on a weighted benefit / cost ratio. The benefit is the amount of invalid space in the unit, while the cost is the need to read valid data and write it elsewhere. This is then weighted by the age of the block, which is its time since last invalidation, and a large weight indicates relatively static data. The second policy writes the data to two units, one for sectors relocated from “hot” units and updates not during reclamation. The other is for sectors relocated during reclamation of “cold” units. This clusters the static data in some and dynamic data in others to increase efficiency, at the expense of worse wear leveling. This makes it easier to find the data that is updated often, however lets cold data sit in blocks, widening the gap is the leveling of wear.

Another policy looks to take advantage of hot and cold data. Wu and Zwaenepoel proposed a partitioned erase units method, that partitions erase units into fixed-size partitions (Wu and Zwaenepoel). The lowered numbered partitions are for “hot” blocks while the higher number partitions are for “cold” data. Each partition also has one active erase unit for updated blocks, and when it fills, the policy finds the unit with the least valid sectors in the same partition. Blocks that are not updated slide toward the beginning of erase units. If the frequency of the current partition is higher than the average
frequency, some blocks are moved to neighboring partitions. The policy does use a simple swap for some wear-leveling. When the erase-count of the most worn-out unit is higher by 100 than the least worn-out unit, the data is swapped between the two units. By separating data, it allows for easier swapping when needed. By combining cold and hot data into single blocks the count of cold sectors are brought up by the hot sectors, without directly helping the overall erase counts as when cold data is in their own blocks and swapped with hot data. Here it helps level but at the cost of overall erase counts, leading to quicker failures of blocks.

Several policies use calculated scores to determine units to be reclaimed. The first of these is the weighted combo by Wells (Wells). It is a combination of efficiency and wear-leveling. The unit with the maximal score is reclaimed next. This score for block $j$ is determined by $score(j) = 0.8 \times obsolete(j) + 0.2 \times (\max\{erasures(k)\} - erasures(j))$, where $k$ represents an arbitrary block (here with the maximum number of erases), obsolete is the amount of invalid data and erasures are the number of erasures undergone. This weights efficiency heavily and wear-leveling lightly. When the difference between the most and least worn out units is 500+, it selects units using a wear-heavy policy of $score'(j) = 0.2 \times obsolete(j) + 0.8 \times (\max\{erasures(k)\} - erasures(j))$. This ensures the wear imbalance does not get too extreme (Gal and Toledo). By changing the threshold, this method can run more efficiently or with more wear leveling, making it more dynamic. With the current level of 500, it keeps the balance in check from getting too large, however is more concerned with efficiency.

Another policy calculating a score is the CAT block clustering patented by Chiang et al (Chiang, Lee and Chang). This method was also built upon and improved
and called DAC (Chiang and Chang). It uses the temperature of the block to estimate the likelihood that it will be updated soon. The temperature is maintained by rising when updated and cooling down over time. Data is also classified as read-only, cold and hot, and is only reclassified during reclamation. The blocks are also given a score calculated by \( \text{cat-score}(j) = \frac{\text{obsolete}(j) \times \text{age}(j)}{\text{valid}(j) \times \text{erasures}(j)} \) with age being a discrete monotone function of time since last erased. This identification and score based method can lead to data clustering within blocks, as demonstrated in Figure 5, from figures and descriptions in (Chiang, Lee and Chang). With read-only (or cold) blocks staying in a block after the hot blocks are updated, the read-only (or cold) blocks are moved towards the beginning of the new block when the current block is reclaimed. When hot data is added, it will be towards the end of the block. This system prefers units with lots of obsolete data and little valid data. Blocks with those characteristics are very efficient for reclamations, because many free sectors will be available after the reclamation. However, this system does little to level the wear of the blocks. Also, maintaining the temperature is complicated by the cooling down decay effect over time. The method’s classification
of blocks also complicates the mapping of data, as well as contributes to the skew wear-
leveling.

2.4. Log-Structured File System

Log-structured file systems look for a way around the problem with modifying flash memory in place. They take the idea of a journaling file system and make the journal, usually used for metadata, the file system. The disk is like a continuous, infinite medium called a log. It consists of fixed-sized segments of contiguous areas of disk chained into a linked list. Data is always written to the end of the log, with pointers used to change data. The files are identified by logical inode index, where logical inodes are mapped to physical location in the log by a table in memory. It reclaims space by finding segments with large amounts of obsolete data, and copies the valid data to the end of the log, erase the segment and add it to the end of the log.

On such system is the Research-In-Motion File System (Parker). It is designed to store contiguous variable-lengths records of data with unique identifiers. The file system is organized as a perfectly circular log containing a sequence of records. Being circular, cleaning may not be effective, so it proposes to partition flash further into a log for hot and a log for cold data. Keeping the records contiguous also allows the system to return pointers directly into the NOR flash in read operations. By the base of this design, if hot data is kept in one log and cold in another, unless blocks are exchanged between logs, one log will have high erase counts and the other low.

The Journaling Flash Filing System (JFFS) focuses mainly on NOR devices. The log consists of a linked list of variable-length nodes, most of which contain parts of files. At mount time, it scans nodes and builds two data structures, a direct map from each
inode number to the most recent version on flash in a hash table, and a collection of structures that represent each valid node on the flash. Each structure is in two linked-lists, one according to the physical address while the other in order. An updated version, JFFS2 has a simple wear-leveling technique. Every 100\textsuperscript{th} reclamation, it selects a unit with only valid data in attempt to move static data around (Gal and Toledo). The structures of mapping the data provide easy access to the data. As can be seen however, this implementation does little to level the wear across its blocks.

A system similar to JFFS, but designed for the NAND flash devices is Yet Another Flash Filing System (YAFFS). The files are stored in fixed sized chunks (512B, 1KB or 2KB). It relies on being able to associate a header with each chunk. The mapping uses a tree structure with 32 byte nodes, where internal nodes contain 8 pointers to other nodes and leave nodes contain 16 2-byte pointers to physical addresses. It uses approximate pointers, where each pointer value represents a contiguous range of chunks, which work because the data they point to are self describing. Invalidity is signaled by at least 4 or more zero bits in the byte. The tree structure uses another way of looking at log designs. Wear-leveling is attempted by infrequent random selection of an erase unit to reclaim. They argue that wear-leveling is less important for NAND devices because they are often shipped with bad pages to improve yield, therefore can cope with errors from wear imbalance (Gal and Toledo). While these devices may be better at coping with errors, bad wear imbalance can lead to many larger failures in the device, making it unusable due to loss of data.

Similar to YAFFS is the Trimble File System, which was designed as a NOR implementation for navigation (Marshall and Manning). The files are in 252-byte chunks
with a 4-byte header. The file includes a header sector made up of the file number, a valid/invalid word, file name and up to 14 records where the last one is valid. The record contains a size, checksum and the last modification time. The reclamation is chosen by number of valid sectors still contained, only if no free sectors, where ties are broken by erase counts for some wear-leveling. Sectors are allocated sequentially within erase units and the next erase unit is selected from among available based on erase counts. This system looks almost solely at sector usage to determine reclamation, only using any wear-leveling in tie. This provides very little help to leveling the wear of blocks.

Another policy that can be implemented within other policies is a block recycling scheme called a migration (Lee, Kim and Kwon). This method is used to reduce the cost of a merge operation. Repeated write patterns can force costly merge operations. In the merge, a new empty log block is selected, and then it copies all the valid sectors from the old log block and the empty log block to the new log block. When a relatively small number of pages are written repeatedly, a migration can be used instead. In a migration, a new log block is selected but only the valid pages from the old log block are copied. This saves on the cost for copying the old log block. This operation can be preferable at a certain equilibrium point that can be calculated. An optimized version of using merges and migrations can be calculated for a particular flash memory system, allowing as much cost to be reduced using migrations appropriately (Lee, Kim and Kwon). While not a full policy, it is another piece that can be combined into other policies. Other methods can be optimized by including migrations into their designs.

Many of these file systems were designed as log systems, but not with flash memory directly in mind. Some use files of fixed sized chunks, where as flash memories
come in several sizes, and the size of the sectors is determined by the memory type, not the system on top of it. It gets even more difficult with variable length file sizes. There are log systems that however are designed to work directly upon flash memory designs. They are more like other flash memory techniques than the above log-based file systems.

Two other systems maintain a small number of log blocks in flash memory as temporary storage for overwrites. If a collision occurs, forwards new data to empty sectors in log blocks instead of erasing original data. The first scheme is BAST, or block associative sector translation. The FTL finds physical location by calculating the logical block number (lbn) from the logical sector number (lsn) with lsn div number of sectors per block. This then retrieves the physical block number (pbn) corresponding to lbn from the block-level mapping table. The FTL calculates the offset of the retrieved physical block where the sector data should be written by lsn mod number of sectors per block. If it is already written in, a free-block is allocated with the free-block list, and copies all other written sectors data block to the free block. When a large number of sector copies and erase operations happen, a merge operation occurs. It merges victim log block and data block. The merge occurs by first copying up to date sectors from two log blocks to a free block and then exchanging free block with the original block. It then updates block-level mapping table and removes the entry corresponding to the victim log block from sector-level mapping table. Finally, it erases both the victim and data blocks and then returns them to the free-block list (Lee, Park and Chung). Using log blocks in this manner lessen the average erase counts of the blocks. It allows for the delay of erases until a problem with the log block. Block level associativity does have two performance
problems: they experience numerous capacity misses causing block thrashing, and has low space utilization when replace from log buffer.

A system proposed to overcome the low space utilization of BAST is the fully-associative sector translation (FAST) (Lee, Park and Chung). This system makes the degree of associativity between logical sectors and log blocks higher, achieving better write performance. The sector being overwritten can be placed in any log block offset. The log blocks are divided into two areas: one log block for sequential writes and the other blocks for random writes. This can be seen in Figure 6, drawn from figures and descriptions in (Lee, Park and Chung). The log block on the left is for sequential writes, while the block on the right is for random writes. The random writes are place anywhere within the block, not worrying about the proper offset. With the block for sequential writes, blocks are placed in their proper offsets filling the block. A switch can be performed by exchanging the victim block with the data block when all the sectors are written to. This is much more efficient than the regular merge operation. Each type of block also has a separate sector-level mapping table. The Sequential Write (SW) log block mapping table keeps information on which logical block the current log block corresponds to and how many sectors currently are stored in the log block. The Random
Write (RW) log block mapping table records which sectors written and the offsets of RW log blocks, taking the simple approach of filling sectors in sequence. When a collision occurs, the sector is directed either to the SW log block if it will result in a switch or the RW log block otherwise. A merge in the SW log block differs from the RW log block. In the SW log block, it uses an optimized version, where it copies the data of empty sectors from the original block, and then exchange the updated log with the data block, erasing the data block and returning it to the free-block list. In the RW log block it performs the merge like in BAST. With the fully associative nature of this design, erases are delayed even longer and the average erase counts lessened. Using a log block for sequential writes allows a switch operation, taking just one erase compared to two for merge operations.
Chapter 3: RESEARCH GOALS AND METHODOLOGY

The research in this paper looks to help with flash based devices. There are many different implementations that can be run over the flash memory. Each of these methods have their own strengths and weaknesses. When a program wants to run on a flash device, it is hard to know how the memory will react. It is easily possible to see how it reacts to the hard drive on a computer simply by testing it. However, for flash memory it is not so simple. Flash memories have different properties. It would be useful to see how the program will affect the wear on the memory. This will also differ by device, depending on the implementation it uses. This paper looks to develop such a simulator, a simulator that can simulate different implementation of the memory and provide feedback as to the state of the memory. With such statistics, the program can be optimized for operating on the flash device. Therefore, the program can be changed instead of trying to develop a new implementation for the flash memory.

In order to achieve such a solution, there are several steps undertaken. The first is the research of the techniques for memory management. There are many different methods for flash memory, including techniques centered on wear leveling, efficient reclamation and log-based techniques. After examining the different techniques, several had to be selected for use in the simulator. The four techniques selected each have their own strengths, and all work to the same end goal of keeping the memory from failing as long as possible, but through different methods. The first of which is the simple erase counter (Lofgren, D. and Thelin). This method provides a naïve, simple approach to provide a baseline for comparison of other methods. The simple erase counter method looks to level the wear across the memory to keep erase counts from reaching the failure
level. This method directly compares the erase counts and uses a threshold to determine when to switch with the least worn-out block making it a good candidate for comparison, or baseline. The second method selected is Group-Based Wear Leveling (Jung, Chae and Jo). This method is selected for its approach to utilize wear leveling like the simple erase counter method but with a small memory footprint. While this is not an issue on a computer, this can be an important factor on a portable device that will utilize flash memory. It is also a newer proposed technique and while may not be currently implemented on current devices, its characteristics could potentially lead to its deployment in many devices. The third selected method is BAST (Lee, Park and Chung). Log-base techniques provided great potential for implementation for flash devices. Their design works with the memory’s capabilities and characteristics. BAST provides a technique specifically designed for flash memory. Through its use of log blocks, BAST delays erases, therefore lessening the total erase count, in order to extend the life of the memory and keep it from reaching its failure level. The last method selected is FAST (Lee, Park and Chung). BAST work well but like all methods, has some shortcomings. FAST is a newly designed method designed to overcome BAST problems, such as low space utilization. It again uses log blocks, but assigns some for sequential writes and other for random writes. This specialization leads to more optimization and an even less total erase count with the same purpose as BAST, to extend the life of the memory. This new design shows a lot of promise for flash memory devices.

After the selection of the implementation, the next step is to implement these within the simulator’s language, which is Java. They are implemented not only with the structures and values they need to hold, but also with values to provide tracking purposes
and to provide stats on the memory status after operation. The basic building blocks are needed, such as sectors and blocks to be built into a memory managed by the individual method implementations. Once the implementation is complete, the simulator to run these methods is developed. The simulator provides the selection of the proper memory type and the input to the implementations, such as writes and reads to particular logical sector numbers. The simulator is design for simple operation with a console user interface.

For the purpose of testing the designs to compare their performance (and the operation of the simulator) the simulator is run on traces. These traces are of common flash devices, so the traces themselves are relative to the workload the methods would have to run. These include a digital camera, mp3 player, movie player and portable media player. This allows consistent testing across the methods as well. This provides accurate data of the memories when run under these conditions of a normal operation of a flash device. When testing the methods, there are two key statistics. The first of which is average erase count. This shows the amount of wear on the device. It helps show the amount of erases placed on the device. The second key statistic is standard deviation. The standard deviation is, in simple terms, the average distance from the average. In this setting, the standard deviation is an accurate representation of wear leveling. The key factor for wear leveling is to create an even amount of wear across the whole memory, which corresponds to a low standard deviation, as then most blocks are close to the average erase count.

When comparing the methods there are several things expected to be seen. The first is the simple erase counter to have a fairly low standard deviation. It may also have a
higher average erase count because of frequent relocations. I expect the group-based wear
calibration method to perform closely to the simple erase counter, however slightly worse
due to the estimation by grouping blocks together. I do not expect it to be too high of a
difference however. Next, I expect BAST to have a much lower average erase count.
Since BAST uses log blocks to delay writes, it will result in less erases. However BAST
does not have much wear leveling implemented, so I expect the standard deviation to not
be as well. Finally is FAST. I expect it to perform similar to BAST. FAST should
outperform BAST on average erase count, due to its extra optimization for sequential
writes. However, it will be similar to BAST with the standard deviation, as it also
performs little wear leveling.
Chapter 4: SIMULATOR

4.1. Environment

The simulator was developed on a Windows machine. The computer runs on an Intel Core 2 Duo 2.0 GHz processor with 2 GB of RAM. The simulator was developed in Java 1.5 in Eclipse to simulate the flash memory of various sizes and configurations.

4.2. Implementation

The simulator was implemented in Java 1.5 to simulate the flash memory. The system was developed in an object-oriented design. The system was developed to simulate the algorithms and policies on different configurations of flash memories. The base design allows the memory to be configured to different settings, such as differing between SLC and MLC designs, varying the sizes of sectors and sectors per blocks. The algorithms were tested 2KB sectors with 64 sectors per block (SLC) and 2KB sectors with 128 sectors per block (MLC). These are the common configurations of these memories. While MLC is cheaper making it more common for actual use, it is also important to test the SLC configurations, as it provides a more stable medium in turns of failure levels. The amount of erasures needed to cause a failure is also adjusted between SLC and MLC, 100,000 and 10,000 respectively. The overall size of the memory can be adjusted, with the default testing being done on 512MB and 1GB (1024MB) memories. The simulator also has the ability to write data to memory, allowing it to run on more than just traces.

4.3. Algorithms Simulated

Several algorithms were chosen for implementation. Many of the policies described earlier in Chapter 2 are not full solutions. They are algorithms and policies for
particular portions of the memory management. The underlying implementations can differ and different policies could potentially be combined. The first policy selected was the Simple Erase Counter (Lofgren, D. and Thelin). This was implemented in a naïve manner overall to provide a good baseline for measuring the wear-leveling and algorithms.

The second algorithm selected was the Group-Based algorithm (Jung, Chae and Jo). As mentioned in Chapter 3, this method aims to provide wear-leveling but in an efficient manner in regards to space. This makes it a slight combination of wear-leveling and efficiency. It is also a newly proposed algorithm for handling flash memory. It combines the blocks into groups and maintains information on the groups instead of individual blocks. These two algorithms are representative of the wear centric policies.

In order to represent the log-based solutions, the third selected algorithm was a log-based structure known as BAST (Lee, Park and Chung). This implements the log-based structured design for maintaining and assigning blocks to the log. This structure works by appending data onto the end of the log when it needs to be written. This makes the assigning blocks simpler, however makes for a more complicated process for the reclaiming of blocks while presenting merge operations that cause overhead.

The final algorithm selected was FAST (Lee, Park and Chung). This is a newly proposed log-based solution, proposed to exceed BAST by using fully associative placement. Most of the other solutions are file systems designed in the log manner of design, where BAST and FAST are specifically designed for flash memories. Log-based solutions are designed to hold of erases through log blocks, and FAST does this similar to a fully associative cache.
4.4. Simulator Operation

The operation of the simulator is kept simple and straightforward. It has an easy
to use console based user interface for selection of the memory implementations and
types. The simulator creates the appropriate memory implementation based on input from
the user. That implementation creates a Memory, which contains Blocks made up of
Sectors, or in the case of Group-Based Wear Leveling, the Memory is made of up Groups
which contain the Blocks that are made up of Sectors. This containment of creation can
be seen in Figure 7.

The simulator has several basic steps for operation, each of which are self
describing with output from the simulator. Step one, shown in Figure 8, the user selects
the appropriate implementation (i.e. BAST) by inputting the corresponding number. As
implementations are added, so can choices to this menu. In step 2, the user selects the
appropriate memory type (i.e. SLC 512KB), as shown in Figure 9. Other configurations
of the memory types can also be added at this same location, such as for larger memory

sizes. The third step is where the user selects the appropriate input (See Figure 10). Here the user can choose between one of the four tested traces or as program input. For program input, the complete translation from program to the simulator is not implemented, but the simulator would sit in a loop on a flag to be set when the program is finished and would simply wait for input from the program inside this loop. Next, if a trace was selected, the user would input the number of iterations for the trace to run in step 4 (See Figure 11). Finally, step 5 is where the user observes the output after the input has finished (See Figure 12). These statistics that are calculated can easily be added to and more information tracked during the execution of the simulator.

![Figure 8 - Simulator Step 1.](image-url)
Figure 9 - Simulator Step 2.

Figure 10 - Simulator Step 3.
Figure 11 - Simulator Step 4.

Figure 12 - Simulator Step 5.
Chapter 5: RESULTS

The simulation demonstrates many different properties. First, it demonstrates the behavior of the algorithms on the different memory types (SLC, MLC) and sizes. These are the two types of memory in use by commercial products. MLC are used more often due to the cost. Lower costs mean more profits and possibly lower prices to beat out the competition. Each algorithm is tested on several settings. First is the large SLC setting, of 512KB memory, with 2KB sectors, 64 sectors per block, and 100,000 times erasure level. The small setting of 512KB sectors is not commonplace in today’s devices so is not examined at this time. That small of a sector size with the overall growing size of today’s devices is not practical for implementation. The large setting is then also then tested with a 1GB size memory. Then is MLC memory setting of 512KB memory, with 2KB sectors, 128 sectors per block and only 10,000 time erasure level, and again tested with a 1GB size memory. The 512MB and 1GB sizes are common sizes among the mass market of devices. There are numerous MP3 players offered at low costs due to the smaller size of memory. While memory cards come in larger sizes, these are the more popular low end size more often used to start off, making them more common.

The tests are run on several traces, as used in (Jung, Chae and Jo). The traces correspond to an mp3 player (MP3), a digital movie player (MOV), a portable media player (PMP) and a digital camera (PIC). These are all common devices that use flash memory. Each trace provides a different trend of how they act upon the memory. MP3 players work with songs. These files differ in size based on factors such as song length and capture quality. The corresponding MP3 trace therefore has smaller sequential writes of varying lengths. A digital movie player is similar to an MP3 player in the nature of the
files, but movie files are of larger size, depending upon factors such as length and resolution. This results in longer sequential writes of various sizes. A digital camera takes pictures, of the same resolution unless changed by the user, so it will result in many files of the same size. The trace reflects this in being sequential writes of uniform length.

Finally, there are portable media players, which take a variety of files types. They can open MP3s, movies, pictures, among other many types of files. This is reflected in the trace as almost a combination of the previous three traces. These trends provide insight as to the results of the testing and to future use in programs. The nature of their writes can be compared to these trends and the appropriate results examined, and assess their program without even needing to test it or during the design phase. Since these are traces, there is no data to write to memory, just sizes of data, so for this experiment, dummy values of the write count are used. These traces are looped 10 times to create a larger number of writes. The key statistics looked at, as proposed earlier, are the average erase counts of the blocks and the standard deviation, which is an accurate measure of the wear leveling. By testing across these combinations of traces, memory implementations and memory types, we should be able to observe some trends based on their characteristics. These trends can better help understand the effect of the type of write sequences on the various memories, allowing optimization (of programs) to exploit the trends and best outcomes.

The first graph examined is the effect of threshold on the average erase count and the standard deviation for the Simple Erase Counter method. This is shown in Figure 13. The threshold, as explained in the background chapter, is the value used to compare if a block is too worn-out in comparison to another block, in this case the least worn-out
block. As the graph demonstrates, the smaller the threshold, the better the standard deviation, which is because wear leveling relocation will be used more often. This however leads to a higher average erase count because two erases are used instead of one in wear leveling relocation compared to regular relocation.

Next, the MP3 trace results are presented. As mentioned earlier, the MP3 trace contains sequential writes of various smaller sizes. The MP3 traces is graphed over average erase count (Figure 14) and standard deviation (Figure 15). Several things are noticeable. First, when looking at the average erase count, the y-axis is in a logarithmic scale, using base 10. This is because of the large gap between the Simple Erase Counter (abbreviated as S.E.C) and Group-Based Wear Leveling, and the log-based block techniques. Both BAST and FAST, because of their abilities to delay erases, have a much lower total erase count leading to the much smaller average erase count as seen in the graph. This is looked at in greater detail later. If the graph was done using a normal scale for the y-axis, the BAST and FAST bars would just be small spots on the bottom. With this scale, we can better see the differences across memory types. First, across all implementations, you can see that the larger memories perform better, which is as one
would expect. Next, you can see that in the traditional methods (Simple Erase Counter and Group-Based Wear Leveling) the SLC outperform the MLC memory type. This is due to the configuration. The MLC have 128 sectors per block, giving more opportunities for collisions within a block that cause relocations. This does not hold true for the log-based techniques (BAST and FAST), where more sectors per block give more optimization possible. The next thing that can be seen is that Simple Erase Counter and Group-Based Wear Leveling perform close to the same across all of the memory types where FAST slightly outperforms BAST across all memories.

The differences between the simpler methods and the log-based methods show up greatly in the erase counts. This is largely due to the log blocks delaying erases. Consider a block with 10 sectors, with each sector filled with the values 0 to 9 respectively, such as shown in Figure 16. Now assume you have a sequential write to overwrite this block. The
first write overwrites sector 0 with ‘00’. In the simple erase counter method, this sector is
invalidated, the valid data from this block (1 – 9) are copied to a new block, and then the
write is completed. Assume the next write is for sector 1 with the value of ‘01’
continuing to sector 9 with ‘09’. In the simple erase counter method, this takes 10 erases.

Now let’s consider the log-based methods, such as FAST. We start with the same block
as before, this time shown in Figure 17. When the write to sector 0 is performed and a
collision recognized, instead of copying the rest of the data to a new block, the sector is
invalidated, and a log block is fetched. Now, this write is performed to the log block. The
other 9 writes can be placed also in this log block. When the block is full after these
writes, the block can be switched with the old block and thus the sequential writes result
in only 1 erase, or in a worst case using the unoptimized merge, two erases.

The next trace examined is the MOV trace. The MOV trace is similar to the MP3
trace, however the sequential writes are of a larger size, spanning a greater number of
sectors and blocks. The behavior of the implementations on this trace is fairly similar for
the average erase count as the MP3 trace, only slightly higher levels as this trace incurs
more write operations leading to more erases (See Figure 18). In the standard deviation,
shown in Figure 19, the Simple Erase Counter performs about the same, independent of the size of the memory. The Group-Based Wear Leveling again has worse numbers for the MLC than the SLC memory types. In the log-based methods, both BAST and FAST have larger gaps between the sizes of memories. This shows the little wear leveling done in these two methods. The more erases that occur, the higher the standard deviation gets, therefore the more skewed the memory is in erase counts from least to most worn-out blocks.

The third trace is the PIC trace. This is the trace from a digital camera, providing uniform length sequential writes. This trace shows some interesting trends. When looking at Figure 20 showing average erase count, for the traditional techniques, the larger memories are still better than the smaller ones, but only slightly. The differences are much smaller than other implementations. In the log-based techniques, the difference is even smaller, the counts are even closer. In Figure 21, we can see that the Simple Erase Counter is again very consistent despite memory size. The Group-Based Wear Leveling
technique has large differences again in the sizes of memories, performing much better with the larger size memory. The log-based techniques perform slightly better in this trace, however the trace has less total writes which contributes to this.

The final trace is the PMP trace. This trace is the portable media player, which is like a combination of all the other traces’ trends. It contains writes of all the other types: small, large, uniform size. This interestingly behaves similarly to the PIC trace, only slightly worse on average. In Figure 22, the average erase count of both of the traditional techniques has fewer differences in the memory sizes. The log-based techniques are again quite close, however, has more of a difference among them, stepping downward across the memory types. In Figure 23, the standard deviation stays consistent in the Simple Erase Counter, close among the SLC and among the MLC memory types. The Group-Based Wear Leveling also performs with closer values. The BAST and FAST schemes, both have skews between the memory sizes, however the BAST technique has a larger difference between the SLC and MLC types.
Overall, the log-based schemes outperform the traditional techniques. They have such a large difference in the average erase count that their lack of wear-leveling does not generally become a factor at a realistic level. With the high erase counts of the traditional techniques, they overpower and difference made by a lack of wear-leveling. For the wear-leveling of the log-based techniques to become a factor, the counts of the traditional techniques would be that much larger, making them impractical. This makes the log-based techniques the key candidate for implementation.
Chapter 6: CONCLUSION

Flash memory is growing in popularity due to its properties that make it suitable for portable devices. The popularity of those devices makes the memory type itself important. However the memory does not and cannot behave the same as traditional memory. This leads to the necessity for the development of new algorithms to harness the power for flash memory. These algorithms look to level wear and efficiently reclaim blocks for writing.

Examining these techniques shows the pros and cons of each of the various algorithms. As some techniques are very specialized in nature, the ability to combine the techniques to get the ideal behavior for the intended device is possible. Tailoring the amount of wear-leveling, efficient reclamation and memory space can be used to find the optimal levels necessary.

With such a simulator, high level programs can be tested across these different implementations. With the available statistics returned, the high level algorithms can be tweaked to be optimized for running on flash based devices. Sequential writes can be used to take advantage of switch operations. Algorithms can be restructured to avoid repeatedly changing one value. The behavior of the program can be changed to minimize the erases needed to achieve the same result. This can lead to better use of current flash technology. New programs can work smoother and faster on flash memory then previously possible, through testing on this simulator.
BIBLIOGRAPHY


package flashSim;

import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Scanner;
import java.util.StringTokenizer;

public class BAST {
    private ArrayList<Block> memory; // by pbn
    private ArrayList<Block> freeLogBlocks;
    private ArrayList<Block> logBlocks;
    private int[] blockLevelMap; // by lbn, stores pbn
    private String[][] sectorLevelMap; // lbn - pbn - lsns
    private int sectorsPerBlock;
    private int numBlocks;
    private int numLog;
    private Block extra;
    private int numErases;

    public BAST(int memSize, int sectorsPerBlock, int sectorSize, int failCount) {
        this.sectorsPerBlock = sectorsPerBlock;
        numLog = 31;
        numBlocks = memSize / (sectorsPerBlock * sectorSize);
        numErases = 0;
        blockLevelMap = new int[numBlocks - numLog - 1];
        sectorLevelMap = new String[numLog][3];
        for (int i = 0; i < numLog; i++) {
            sectorLevelMap[i][0] = "";
            sectorLevelMap[i][1] = "";
            sectorLevelMap[i][2] = "";
        }
        createMemory(memSize, sectorSize, failCount);
    }

    public void createMemory(int memSize, int sectorSize, int failCount) {
        memory = new ArrayList<Block>(memSize);
        freeLogBlocks = new ArrayList<Block>(numLog);
        logBlocks = new ArrayList<Block>(numLog);
        int logicalBlocks = numBlocks - numLog - 1;
        Block temp;
        for (int i = 0; i < logicalBlocks; i++) {
            temp = new Block(sectorsPerBlock, i, failCount, sectorSize);
            memory.add(temp);
            blockLevelMap[i] = i;
        }
        // loop log blocks
        Block temp2;
        for (int j = logicalBlocks; j < numBlocks - 1; j++) {
            temp2 = new Block(sectorsPerBlock, j, failCount, sectorSize);
            memory.add(temp2);
            freeLogBlocks.add(temp2);
        }
    }
}

APPENDIX

1. BAST.java
Block temp3 = new Block(sectorsPerBlock, numBlocks-1, failCount, sectorSize);
memory.add(temp3);
extra = temp3;

private int calcLBN(int lsn){
    int lbn = lsn / sectorsPerBlock;
    return lbn;
}

private int calcOffset(int lsn){
    int offset = lsn % sectorsPerBlock;
    return offset;
}

private int translate(int lbn){
    int pbn = blockLevelMap[lbn];
    return pbn;
}

private int getLogBlockNum(int lbn){
    int log = -1;
    for(int i=0; i<numLog; i++){
        if(!sectorLevelMap[i][0].equals("")){
            if(Integer.parseInt(sectorLevelMap[i][0])==lbn){
                log = Integer.parseInt(sectorLevelMap[i][1]);
                break;
            }
        }
    }
    return log;
}

private int getSLM(int lbn){
    int slm = -1;
    for(int i=0; i<numLog; i++){
        if(!sectorLevelMap[i][0].equals("")){
            if(Integer.parseInt(sectorLevelMap[i][0])==lbn){
                slm = i;
                break;
            }
        }
    }
    return slm;
}

String read(int lsn){
    int lbn = calcLBN(lsn);
    int offset = calcOffset(lsn);
    int pbn = this.translate(lbn);
    return memory.get(pbn).getData(offset);
}

public void write(int lsn, String data){
    int lbn = calcLBN(lsn);
    int offset = calcOffset(lsn);
    int pbn = this.translate(lbn);
    Block b = memory.get(pbn);

    //if empty -> write to sector
    if(b.getSector(offset).isEmpty()){

b.setData(offset, data);
} else {
    //invalidate current sector in current block
    b.get_sector(offset).invalidate();
    //if have log block
    int logNum = this.getLogBlockNum(lbn);
    if(logNum!=-1){
        int slm = getSLM(lbn);
        String lsns = sectorLevelMap[slm][2];
        //if in log block -> invalidate
        StringTokenizer st = new StringTokenizer(lsns, ",");
        int count = 0;
        Sector sectorTemp;
        while(st.hasMoreTokens()){
            int lsnTemp = Integer.parseInt(st.nextToken());
            if(lsn==lsnTemp){
                sectorTemp = memory.get(lbn).getSector(count);
                if(sectorTemp.isValid()){
                    sectorTemp.invalidate();
                }
            }
            count++;
        } //if room in log block -> write
        Block b2 = memory.get(logNum);
        int free = b2.freeSector();
        if(free!=-1){
            b2.setData(free, data);
            sectorLevelMap[slm][2] = sectorLevelMap[slm][2] + lsn + ",";
        } else {
            //log block full, merge then write
            this.merge(b, lbn, b2);
            this.write(lsn, data);
        } else {
            //no log block in use
            if(freeLogBlocks.size()!=0){
                //write to new log block
                Block newBlock = freeLogBlocks.remove(0);
                logBlocks.add(newBlock);
                newBlock.setData(0, data);
                //set map
                int enumb = 0;
                for(int emp=0; emp<numLog; emp++){
                    if(sectorLevelMap[emp][0].equals("")){
                        enumb = emp;
                        break;
                    }
                }
                sectorLevelMap[enumb][0] = String.valueOf(lbn);
                sectorLevelMap[enumb][1] = String.valueOf(pbn);
                sectorLevelMap[enumb][2] = String.valueOf(lsn) + ",";
            } else {
                //no log blocks available, merge -> write
                this.merge();
                this.write(lsn, data);
            }
        } else {
            //if have log block and log block is full
            //merge then write
            this.merge(logNum, b, b2);
            this.write(lsn, data);
        }
    } else {
        //check if any available
        //write to new log block
        Block newBlock = freeLogBlocks.remove(0);
        logBlocks.add(newBlock);
        newBlock.setData(0, data);
        //set map
        int enumb = 0;
        for(int emp=0; emp<numLog; emp++){
            if(sectorLevelMap[emp][0].equals("")){
                enumb = emp;
                break;
            }
        }
        sectorLevelMap[enumb][0] = String.valueOf(lbn);
        sectorLevelMap[enumb][1] = String.valueOf(pbn);
        sectorLevelMap[enumb][2] = String.valueOf(lsn) + ",";
    } else {
        //log block full, merge then write
        this.merge(b2, lsn, data);
    }
}
public void merge()
{
    // select victim
    int vLBN = 0;
    int usedNum = 0;
    Block logBlock = null;
    // log block with most used sectors
    Block temp;
    for(int i=0; i<logBlocks.size(); i++){
        temp = logBlocks.get(i);
        int free = temp.freeSector();
        if(free==-1){
            logBlock = temp;
            usedNum = numLog + 1;
            break;
        } else {
            if(free > usedNum){
                logBlock = temp;
                usedNum = free;
            }
        }
    }
    int lPBN = logBlock.getPBN();
    // get lbn from sectorLevelMap
    int tempPBN = 0;
    for(int j=0; j<numLog; j++){
        tempPBN = Integer.parseInt(sectorLevelMap[j][1]);
        if(tempPBN==lPBN){
            vLBN = Integer.parseInt(sectorLevelMap[j][0]);
            break;
        }
    }
    // get pbn from blockMap
    int pbm = blockLevelMap[vLBN];
    // get block from memory
    Block b1 = memory.get(pbm);  
    // merge blocks
    this.merge(b1, vLBN, logBlock);
}

public void merge(Block b1, int lbn1, Block b2){
    // copy valid sectors from both blocks to new block
    int blockSize = b1.getSize();
    for(int i=0; i<blockSize; i++){
        Sector s = b1.get Sector(i);
        if(s.isValid()){
            extra.setData(i, s.getData());
        }
    }
    blockSize = b2.getSize();
    Sector s;
    for(int j=0; j<blockSize; j++){
        s = b2.get Sector(j);
        if(s.isValid()){
            extra.setData(j, s.getData());
        }
    }
    // update mapping
    blockLevelMap[lbn1] = extra.getPBN();
    for(int k=0; k<numLog; k++){
        if(!sectorLevelMap[k][0].equals(""))
        {
            break;
        }
    }
}
if(Integer.parseInt(sectorLevelMap[k][0])==lbn1){
    sectorLevelMap[k][0] = "";
    sectorLevelMap[k][1] = "";
    sectorLevelMap[k][2] = "";
    break;
}
//erase b1 and b2
b1.erase();
b2.erase();
numErases++;
numErases++;
//make b1 new extra, b2 log block
extra = b1;
freeLogBlocks.add(b2);
}

public void stats(){
    System.out.println("Printing Stats...");
    System.out.println("Total Erases: "+numErases);
    double sum = 0;
    int wornOut = 0;
    int ec = 0;
    Block b;
    //calc avg, wornOut
    for(int i=0; i<memory.size(); i++){
        b = memory.get(i);
        ec = b.getEraseCount();
        sum += ec;
        if(b.getWear())
            wornOut++;
    }
    double avg = sum / memory.size();
    System.out.println("Average Erase Count: "+avg);
    double dev = 0;
    double devSum = 0;
    for(int j=0; j<memory.size(); j++){
        b = memory.get(j);
        dev = b.getEraseCount() - avg;
        dev = dev * dev;
        devSum += dev;
    }
    dev = devSum / memory.size();
    double stndDev = Math.sqrt(dev);
    System.out.println("Standard Deviation: "+stndDev);
    System.out.println("Number of Worn Out Blocks: "+wornOut);
}

2. Block.java

package flashSim;

import java.util.ArrayList;

public class Block {
    private ArrayList<Sector> sectors;
}
private boolean wornOut;
private int eraseCount;
private int failCount;
private int sectorsPerBlock;
private int pbn;

public Block(){
    this(32);
}

public Block(int pbn){
    this(1, pbn, 10000, 512);
}

public Block(int sectorsPerBlock, int pbn, int failCount, int sectorSize){
    this.sectorsPerBlock = sectorsPerBlock;
    this.pbn = pbn;
    wornOut = false;
    eraseCount = 0;
    this.failCount = failCount;

    sectors = new ArrayList<Sector>(this.sectorsPerBlock);
    for(int i=0; i < this.sectorsPerBlock; i++){
        Sector temp = new Sector(sectorSize, pbn, i);
        sectors.add(temp);
    }
}

Sector getSector(int num){
    Sector temp = (Sector) sectors.get(num);
    return temp;
}

void setSector(int num, Sector sector){
    sectors.set(num, sector);
}

String getData(int num){
    return sectors.get(num).getData();
}

void setData(int num, String data){
    sectors.get(num).setData(data);
}

int getPBN(){
    return this.pbn;
}

void erase(){
    Sector temp;
    for(int i=0; i < this.sectorsPerBlock; i++){
        temp = this.getSector(i);
        temp.erase();
    }
    eraseCount++;
    this.checkWear();
}

void checkWear(){
    if(eraseCount >= failCount)
        wornOut = true;
}
int getEraseCount(){
    return eraseCount;
}

int getSize(){
    return sectors.size();
}

boolean getWear(){
    return wornOut;
}

public boolean equals (Object otherObject){
    Block other = (Block) otherObject;
    return this.pbn == other.getPBN();
}

int freeSector(){
    int num = -1;
    for(int i=0; i<sectorsPerBlock; i++){
        Sector temp = this.getSector(i);
        if(temp.isEmpty()){
            num = i;
            break;
        }
    }
    return num;
}

boolean validSector(){
    boolean valid = false;
    for(int i=0; i<sectorsPerBlock; i++){
        Sector temp = this.getSector(i);
        if(temp.isValid()){
            valid = true;
            break;
        }
    }
    return valid;
}

public String toString(){
    String temp = "PBN: " + this.getPBN() + "; Erase Count: " + this.getEraseCount() + "n";
    temp += "[";
    Sector s;
    for(int i=0; i<sectors.size(); i++){
        s = sectors.get(i);
        if(s.isEmpty()){
            temp += "empty";
        }
        temp += s.getData();
        temp += ", ";
    }
    temp += "]";
    return temp;
}
package flashSim;

import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Scanner;
import java.util.StringTokenizer;

public class FAST {
    private ArrayList<Block> memory; // by pbn
    private ArrayList<Block> freeLogBlocks;
    private ArrayList<Block> logBlocks;
    private int[] blockLevelMap; // by lbn, stores pbn
    private int[][] sectorLevelMapSW; // lbn - pbn - #sectors
    private int[] sectorLevelMapRW; // lbn - pbn - lsn - sectorOffset - blockOffset
    private int numLog;
    private int numBlocks;
    private int sectorsPerBlock;
    private Block extra;
    private int numErases;

    public FAST (int memSize, int sectorsPerBlock, int sectorSize, int failCount){
        this.sectorsPerBlock = sectorsPerBlock;
        this.numLog = 31;
        this.numBlocks = memSize / (sectorsPerBlock * sectorSize);
        this.numErases = 0;

        blockLevelMap = new int[this.numBlocks-this.numLog-1];
        sectorLevelMapSW = new int[this.numLog][3];
        for(int i=0; i<this.numLog; i++){
            sectorLevelMapSW[i][0] = -1;
            sectorLevelMapSW[i][1] = -1;
            sectorLevelMapSW[i][2] = 0;
        }
        sectorLevelMapRW = new ArrayList<int[]>(31);
        createMemory(memSize, sectorSize, failCount);
    }

    public void createMemory(int memSize, int sectorSize, int failCount){
        memory = new ArrayList<Block>(memSize);
        freeLogBlocks = new ArrayList<Block>(this.numLog);
        logBlocks = new ArrayList<Block>(this.numLog);
        int logicalBlocks = this.numBlocks - this.numLog - 1;

        for(int i=0; i<logicalBlocks; i++){
            Block temp = new Block(sectorsPerBlock, i, failCount, sectorSize);
            memory.add(temp);
            blockLevelMap[i] = i;
        }

        // loop log blocks
        for(int j=logicalBlocks; j<numBlocks-1; j++){
            Block temp2 = new Block(sectorsPerBlock, j, failCount, sectorSize);
            memory.add(temp2);
            freeLogBlocks.add(temp2);
        }
    }

    public void erase(int blockIndex, int sectorIndex){
        // implementation
    }

    public void write(int blockIndex, int sectorIndex, int value){
        // implementation
    }

    public void read(int blockIndex, int sectorIndex){
        // implementation
    }

    public void printMemory(){
        // implementation
    }

    public int numBlocks(){
        return this.numBlocks;
    }

    public int sectorsPerBlock(){
        return this.sectorsPerBlock;
    }

    public int numLog(){
        return this.numLog;
    }

    public int numErases(){
        return this.numErases;
    }

    public int blockLevelMap(int blockIndex){
        return this.blockLevelMap[blockIndex];
    }

    public int sectorLevelMapSW(int lbn, int pbn){
        return this.sectorLevelMapSW[lbn][pbn];
    }

    public int sectorLevelMapRW(int lbn, int pbn, int lsn, int sectorOffset, int blockOffset){
        return this.sectorLevelMapRW[lbn][pbn][lsn][sectorOffset][blockOffset];
    }
}

3. FAST.java
Block temp3 = new Block(sectorsPerBlock, numBlocks-1, failCount, sectorSize);
memory.add(temp3);
extra = temp3;

private int calcLBN(int lsn)
{
    int lbn = lsn / sectorsPerBlock;
    return lbn;
}

private int calcOffset(int lsn)
{
    int offset = lsn % sectorsPerBlock;
    return offset;
}

private int translate(int lbn)
{
    int pbn = blockLevelMap[lbn];
    return pbn;
}

private int getSWLogBlockNum(int lbn)
{
    int log = -1;
    for(int i=0; i<numLog; i++)
    {
        if(sectorLevelMapSW[i][0]!=-1)
        {
            if(sectorLevelMapSW[i][0]==lbn)
            {
                log = sectorLevelMapSW[i][1];
                break;
            }
        }
    }
    return log;
}

private int getRWLogBlockNum(int lbn)
{
    int log = -1;
    if(sectorLevelMapRW.isEmpty())
    { return log;
    }
    for(int i=0; i<sectorLevelMapRW.size(); i++)
    {
        int[] temp = sectorLevelMapRW.get(i);
        if(temp[0]!=-1)
        {
            if(temp[0]==lbn)
            {
                log = temp[1];
                break;
            }
        }
    }
    return log;
}

private int getRWOldOffset(int lsn)
{
    int offset = -1;
    for(int i=0; i<sectorLevelMapRW.size(); i++)
    {
        int[] temp = sectorLevelMapRW.get(i);
        if(temp[0]!=-1)
        {
            if(temp[2]==lsn)
            {
                offset = temp[3];
                break;
            }
        }
    }
}
return offset;
}

private int getRWLogOffset(int lsn)
{
    int offset = -1;
    for(int i=0; i<sectorLevelMapRW.size(); i++)
    {
        int[] temp = sectorLevelMapRW.get(i);
        if(temp[0]!=-1)
        {
            if(temp[2]==lsn)
            {
                offset = temp[4];
                break;
            }
        }
    }
    return offset;
}

String read(int lsn)
{
    int lbn = calcLBN(lsn);
    int offset = calcOffset(lsn);
    int pbn = this.translate(lbn);
    return memory.get(pbn).getData(offset);
}

public void write(int lsn, String data)
{
    int lbn = calcLBN(lsn);
    int offset = calcOffset(lsn);
    int pbn = this.translate(lbn);
    Block b = memory.get(pbn);
    //if empty -> write to sector
    if(b.getSector(offset).isEmpty())
    {
        b.setData(offset, data);
    } else {
        //invalidate current sector in current block
        b.getSector(offset).invalidate();

        //if have SW log block
        int logNum = this.getSWLogBlockNum(lbn);
        if(logNum!=-1)
        {
            //check if at correct offset
            Block b2 = memory.get(logNum);

            int offsetEnd = b2.freeSector();

            //if less then, merge
            if(offset<offsetEnd)
            {
                if(this.getRWLogBlockNum(lbn)==-1)
                    //this.merge(b, lbn, b2);
                else
                    this.merge2(b, lbn, b2);
            } else
            {
                this.write(lsn, data);
            }
        }

        //if equal, write to spot
        if(offset==offsetEnd)
        {
            b2.setData(offset, data);
            //update map
            for(int m=0; m<numLog; m++)
            {
                if(sectorLevelMapSW[m][0]==lbn)
                    sectorLevelMapSW[m][2]++;
            }
        }
    }
}
break;
    }
}

if (offset == (b2.getSize() - 1))
    this.switchB(b, lbn, b2);

//if greater, write to RW
if (offset > offsetEnd) {
    logNum = this.getRWLogBlockNum(lbn);
    if (logNum != -1) {
        Block b3 = memory.get(logNum);

        //if in log block -> invalidate
        int lOffset = this.getRWLogOffset(lsn);
        if (lOffset != -1) {
            b3.getSector(lOffset).invalidate();
            //find next available sector
            int free = b3.freeSector();
            if (free != -1) {
                b3.setData(free, data);
                int t = new int[5];
                int temp;
                for (int i = 0;
                        i < sectorLevelMapRW.size(); i++) {
                    temp = sectorLevelMapRW.get(i);
                    if (temp[2] == lsn) {
                        t =
                        sectorLevelMapRW.remove(i);
                        break;
                    }
                }
                t[4] = free;
                sectorLevelMapRW.add(t);
            } else {
                //if no free space, block is full
                //merge then write
                if (this.getSWLogBlockNum(lbn) == -1) {
                    this.merge(b, lbn, b3);
                } else {
                    this.merge2rw(b, lbn, b3);
                    this.write(lsn, data);
                }
            }
        } else {
            //if no log blocks in use
            //check if any available

            free2;
        }
    } else {
        //not in block
        int free2 = b3.freeSector();
        if (free2 != -1) {
            b3.setData(free2, data);
            //update map
            int[] t2 = {lbn, b3.getPBN(), lsn, offset, free2};
            sectorLevelMapRW.add(t2);
        }
    }
}
else {
    //if no log blocks in use
    //check if any available

} else {
    //if no log blocks in use
    //check if any available
if(freeLogBlocks.size()!=0) {
    Block newBlock = freeLogBlocks.remove(0);
    logBlocks.add(newBlock);
    //determine SW or RW
    if(offset==0) {
        //SW
        newBlock.setData(0, data);
        //set map
        int enumb = -2;
        for(int emp=0; emp<numLog; emp++) {
            if(sectorLevelMapSW[emp][0]==-1) {
                enumb = emp;
                break;
            }
        }
        sectorLevelMapSW[enumb][0] = lbn;
        sectorLevelMapSW[enumb][1] = newBlock.getPBN();
        sectorLevelMapSW[enumb][2] = 1;
    } else {
        //RW
        newBlock.setData(0, data);
        //set map
        int[] tmap = {lbn, newBlock.getPBN(), lsn, offset, 0};
        sectorLevelMapRW.add(tmap);
    }
} else {
    //no SW block
    logNum = this.getRWLogBlockNum(lbn);
    if(logNum!=-1) {
        Block b3 = memory.get(logNum);
        //if in log block -> invalidate
        int IOffset = this.getRWLogOffset(lsn);
        if(IOffset!=-1) {
            b3.getSector(IOffset).invalidate();
            //find next available sector
            int free = b3.freeSector();
            if(free!=-1) {
                b3.setData(free, data);
                int[] t = new int[5];
                int[] temp;
                for(int i=0; i<sectorLevelMapRW.size(); i++) {
                    temp = sectorLevelMapRW.get(i);
                    if(temp[2]==lsn) {
                        t =
                }
            }
        }
    }
} else {
    //no log blocks available
    this.merge();
    this.write(lsn, data);
    }
}
}

if(sectorLevelMapSW[emp][0]==-1) {
    enumb = emp;
    break;
}

sectorLevelMapSW[enumb][0] = lbn;
sectorLevelMapSW[enumb][1] = newBlock.getPBN();
sectorLevelMapSW[enumb][2] = 1;

sectorLevelMapRW.add(tmap);
break;

} else{
  //if no free space, block is full
  //merge then write
  if(this.getSWLogBlockNum(lbn)==-1)
    this.merge(b, lbn, b3);
  else
    this.merge2rw(b, lbn, b3);

  this.write(lsn, data);
}

} else{
  //not in block
  int free2 = b3.freeSector();
  if(free2!=-1){
    b3.setData(free2, data);
    //update map
    int[] t2 = {lbn, b3.getPBN(), lsn, offset, free2};
    sectorLevelMapRW.add(t2);
  }
}

} else{
  //if no log blocks in use
  //check if any available
  if(freeLogBlocks.size()!=0){
    Block newBlock = freeLogBlocks.remove(0);
    logBlocks.add(newBlock);
    //determine SW or RW
    if(offset==0){
      //SW
      newBlock.setData(0, data);
      //set map
      int enumb = -2;
      for(int emp=0; emp<numLog; emp++){
        if(sectorLevelMapSW[emp][0]==-1){
          enumb = emp;
          break;
        }
      }
      sectorLevelMapSW[enumb][0] = lbn;
      sectorLevelMapSW[enumb][1] =
      newBlock.getPBN();
      sectorLevelMapSW[enumb][2] = 1;
    } else{
      //RW
      System.out.println("RW 0");
      newBlock.setData(0, data);
      //set map
      int[] tmap = {lbn, newBlock.getPBN(), lsn,
                   offset, 0};
      sectorLevelMapRW.add(tmap);
    }
}
public void merge()
{
    // select victim
    int vLBN = -2;
    int usedNum = 0;
    Block logBlock = null;
    // log block with most used sectors
    int free;
    Block temp;
    for(int i=0; i<logBlocks.size(); i++)
    {
        temp = logBlocks.get(i);
        free = temp.freeSector();
        if(free==-1)
        {
            logBlock = temp;
            usedNum = numLog + 1;
            break;
        }
        else
        {
            if(free > usedNum)
            {
                logBlock = temp;
                usedNum = free;
            }
        }
    }
    int lPBN = logBlock.getPBN();
    boolean swt = false;
    boolean rwt = false;
    // get lbn from sectorLevelMapSW
    int tempPBN = -2;
    for(int j=0; j<numLog; j++)
    {
        tempPBN = sectorLevelMapSW[j][1];
        if(tempPBN==lPBN)
        {
            vLBN = sectorLevelMapSW[j][0];
            swt = true;
            //break;
        }
    }
    int vLBN2 = -2;
    // get lbn from sectorLevelMapRW
    int[] temp2;
    for(int i=0; i<sectorLevelMapRW.size(); i++)
    {
        temp2 = sectorLevelMapRW.get(i);
        if(temp2[1]==lPBN)
        {
            vLBN2 = temp2[0];
            rwt = true;
            break;
        }
    }
}
Block old = logBlock;

if(vLBN == -2 && vLBN2 == -2){
  for(int i=0; i<logBlocks.size(); i++){
    temp = logBlocks.get(i);
    free = temp.freeSector();
    if(!temp.equals(old)){
      if(free == -1){
        logBlock = temp;
        usedNum = numLog + 1;
        break;
      } else {
        if(free > usedNum){
          logBlock = temp;
          usedNum = free;
        }
      }
    }
  }
  IPBN = logBlock.getPBN();

  swt = false;
  rwt = false;

  //get lbn from sectorLevelMapSW
  tempPBN = -2;
  for(int j=0; j<numLog; j++){
    tempPBN = sectorLevelMapSW[j][1];
    if(tempPBN == IPBN){
      vLBN = sectorLevelMapSW[j][0];
      swt = true;
    }
  }

  vLBN2 = -2;
  //get lbn from sectorLevelMapRWs
  for(int i=0; i<sectorLevelMapRW.size(); i++){
    temp2 = sectorLevelMapRW.get(i);
    if(temp2[1] == IPBN){
      vLBN2 = temp2[0];
      rwt = true;
      break;
    }
  }
}

int tempLBN = -1;
boolean swt2 = false;
if(!swt && rwt){
  for(int j2=0; j2<numLog; j2++){
    tempLBN = sectorLevelMapSW[j2][0];
    if(tempLBN == vLBN2){
      swt2 = true;
    }
  }
}
boolean rwt2 = false;
if(!rwt && swt){
    for(int i=0; i<sectorLevelMapRW.size(); i++){
        temp2 = sectorLevelMapRW.get(i);
        if(temp2[0]==vLBN){
            rwt2 = true;
            break;
        }
    }
}

//get pbn from blockMap
int pbn;
if(swt){
    pbn = blockLevelMap[vLBN];
} else {
    pbn = blockLevelMap[vLBN2];
}

//get block from memory
Block b1 = memory.get(pbn);

//merge blocks
if(swt && !rwt2)
    this.merge(b1, vLBN, logBlock);
if(rwt && !swt2)
    this.merge(b1, vLBN2, logBlock);
if(swt && rwt2)
    this.merge2(b1, vLBN, logBlock);
if(rwt && swt2)
    this.merge2rw(b1, vLBN2, logBlock);

public void merge(Block b1, int lbn1, Block b2){
    //copy valid sectors from both blocks to new block
    int blockSize = b1.getSize();
    for(int i=0; i<blockSize; i++){
        Sector s = b1.getSector(i);
        if(s.isValid()){
            extra.setData(i, s.getData());
        }
    }
    blockSize = b2.getSize();
    for(int j=0; j<blockSize; j++){
        Sector s = b2.getSector(j);
        if(s.isValid()){
            extra.setData(j, s.getData());
        }
    }

    //update mapping
    blockLevelMap[lbn1] = extra.getPBN();
    //inverseMap[]
    for(int k=0; k<numLog; k++){
        if(sectorLevelMapSW[k][0]!=-1){
            if(sectorLevelMapSW[k][0]==lbn1){
                sectorLevelMapSW[k][0] = -1;
                sectorLevelMapSW[k][1] = -1;
                sectorLevelMapSW[k][2] = 0;
                //System.out.println(k);
            }
        }
    }
}
break;
}
}

int i2 = 0;
for(int i=0; i<sectorLevelMapRW.size(); i++){
    int[] temp = sectorLevelMapRW.get(i2);
    if(temp[0]!=-1){
        if(temp[0]==lbn1){
            sectorLevelMapRW.remove(i2);
        } else{
            i2++;
        }
    }
}

logBlocks.remove(b2);

// erase b1 and b2
b1.erase();
b2.erase();
numErases++;
numErases++;

// make b1 new extra, b2 log block
extra = b1;
if((freeLogBlocks.size()+logBlocks.size())<numLog){
    freeLogBlocks.add(b2);
} else{
}

}

public void merge2(Block b1, int lbn1, Block b2){
// copy valid sectors from both blocks to new block

    int blockSize = b2.getSize();
    for(int j=0; j<blockSize; j++){
        Sector s = b2.getSector(j);
        if(s.isValid()){ extra.setData(j, s.getData());
    }
    
    blockSize = b1.getSize();
    for(int i=0; i<blockSize; i++){
        Sector s = b1.getSector(i);
        if(s.isValid()){ extra.setData(i, s.getData());
        if(!s.isValid()){
            if(extra.getSector(i).isEmpty()){ extra.setData(i, s.getData());
                extra.getSector(i).invalidate();
            }
        }
    }

    // update mapping
    blockLevelMap[lbn1] = extra.getPBN();
    for(int k=0; k<numLog; k++){
        if(sectorLevelMapSW[k][0]!=-1){

if(sectorLevelMapSW[k][0] == lbn1) {
    sectorLevelMapSW[k][0] = -1;
    sectorLevelMapSW[k][1] = -1;
    sectorLevelMapSW[k][2] = 0;
    break;
}
}

logBlocks.remove(b2);

// erase b1 and b2
b1.erase();
b2.erase();
numErases++;
numErases++;

// make b1 new extra, b2 log block
extra = b1;
if((freeLogBlocks.size() + logBlocks.size()) < numLog) {
    freeLogBlocks.add(b2);
}

public void merge2rw(Block b1, int lbn1, Block b2) {
    // copy valid sectors from both blocks to new block
    int offsetOld = -1;
    int offsetNew = -1;
    Sector s;
    for(int j = 0; j < sectorLevelMapRW.size(); j++) {
        int[] temp = sectorLevelMapRW.get(j);
        if(temp[0] != -1) {
            if(temp[0] == lbn1) {
                offsetOld = temp[3];
                offsetNew = temp[4];
                s = b2.getSector(offsetNew);
                extra.setData(offsetOld, s.getData());
            }
        }
    }

    int blockSize = b1.getSize();
    for(int i = 0; i < blockSize; i++) {
        s = b1.getSector(i);
        if(s.isValid()) {
            extra.setData(i, s.getData());
        } else {
            if(extra.getSector(i).isEmpty()) {
                extra.setData(i, s.getData());
                extra.getSector(i).invalidate();
            }
        }
    }

    // update mapping
    blockLevelMap[lbn1] = extra.getPBN();

    int k2 = 0;
    for(int k = 0; k < sectorLevelMapRW.size(); k++) {
        int[] temp = sectorLevelMapRW.get(k);
        if(temp[0] != -1) {
            int[] temp = sectorLevelMapRW.get(k);
            if(temp[0] == lbn1) {
                offsetOld = temp[3];
                offsetNew = temp[4];
                s = b2.getSector(offsetNew);
                extra.setData(offsetOld, s.getData());
            }
        }
    }

    logBlocks.remove(b2);
    b1.erase();
b2.erase();
    numErases++;
    numErases++;

    // make b1 new extra, b2 log block
    extra = b1;
    if((freeLogBlocks.size() + logBlocks.size()) < numLog) {
        freeLogBlocks.add(b2);
    }
}
if(temp[0]==lbn1) {
    sectorLevelMapRW.remove(k2);
} else {
    k2++;
}

logBlocks.remove(b2);
// erase b1 and b2
b1.erase();
b2.erase();
numErases++;

// make b1 new extra, b2 log block
extra = b1;
if((freeLogBlocks.size()+logBlocks.size())<numLog) {
    freeLogBlocks.add(b2);
}

public void switchB(Block b1, int lbn1, Block b2) {
    // erase b1, make b2 new block
    // update mapping
    blockLevelMap[lbn1] = b2.getPBN();
    // inverseMap[]
    for(int k=0; k<numLog; k++) {
        if(sectorLevelMapSW[k][0]!=-1) {
            if(sectorLevelMapSW[k][0]==lbn1) {
                sectorLevelMapSW[k][0] = -1;
                sectorLevelMapSW[k][1] = -1;
                sectorLevelMapSW[k][2] = 0;
                break;
            }
        }
    }

    logBlocks.remove(b2);
    // erase b1
    b1.erase();
    numErases++;
    // make b1 log block
    if((freeLogBlocks.size()+logBlocks.size())<numLog) {
        freeLogBlocks.add(b1);
    }
}

public void printMemory() {
    System.out.println("Printing Memory...");
    Block b;
    for(int i=0; i<memory.size(); i++) {
        System.out.println(memory.get(i));
    }
}

public void stats() {
    System.out.println("Printing Stats...");
}
System.out.println("Total Erases: " + numErases);

double sum = 0;
int wornOut = 0;
int ec = 0;

Block b;
//calc avg, wornOut
for(int i=0; i<memory.size(); i++){
    b = memory.get(i);
    ec = b.getEraseCount();
    sum += ec;
    if(b.getWear())
        wornOut++;
}

double avg = sum / memory.size();

System.out.println("Average Erase Count: " + avg);

double dev = 0;
double devSum = 0;
for(int j=0; j<memory.size(); j++){
    b = memory.get(j);
    dev = b.getEraseCount() - avg;
    dev = dev * dev;
    devSum += dev;
}
dev = devSum / memory.size();
double stndDev = Math.sqrt(dev);

System.out.println("Standard Deviation: " + stndDev);

System.out.println("Number of Worn Out Blocks: " + wornOut);

4. Group.java

package flashSim;

import java.util.ArrayList;

public class Group {

    private double totalAvg;
    private double partialAvg;
    private int roundRobinIndex;
    private int size;
    private int n; //number not swapped
    private ArrayList<Block> blocks;

    public Group(int size, ArrayList<Block> blocks) {
        this.size = size;
        this.n = size;
        this.blocks = blocks;
        roundRobinIndex = 0;
        double temp = 0.0;
    }
}
for(int i=0; i<
size; i++)
{
    temp += blocks.get(i).getEraseCount();
}
totalAvg = temp / this.size;
partialAvg = totalAvg;

ArrayList<Block> getBlocks()
{
    return blocks;
}

Block getBlock()
{
    Block temp = blocks.get(roundRobinIndex);
    roundRobinIndex = (roundRobinIndex + 1) % size;
    return temp;
}

Block getBlock(int i)
{
    return blocks.get(i);
}

Block getCheckBlock()
{
    return blocks.get(roundRobinIndex);
}

int getRR()
{
    return roundRobinIndex;
}

double getpAvg()
{
    return partialAvg;
}

double gettAvg()
{
    return totalAvg;
}

void updateRR()
{
    roundRobinIndex = (roundRobinIndex + 1) % size;
}

void updateAvg(int ecOld, int ecYoung)
{
    totalAvg = totalAvg + (ecOld - ecYoung)/size;
    n-=1;
    if(n==0)
    {
        partialAvg = totalAvg;
        n = size;
    }
    else
    {
        partialAvg = ((partialAvg * n) - ecYoung)/(n-1);
    }
}

void swapBlock(Block oldB, Block newB)
{
    blocks.remove(oldB);
    blocks.add(newB);
    //this.updateAvg(oldB.getEraseCount(), newB.getEraseCount());
}
5. GoupBasedWear.java

package flashSim;

import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Scanner;
import java.util.StringTokenizer;

public class GroupBasedWear {

    private static final int THRESHHOLD = 30; // as recommended for balanced performance
    private int[] map;
    private int[] inverseMap;
    private ArrayList<Block> memory;
    private Block extra;
    private Group youngest;
    private ArrayList<Block> free;
    private int numFree;
    private ArrayList<Group> groups;
    private int numGroups;
    private ArrayList<int[]> extraMap;
    private int numErases;
    private int sectorsPerBlock;

    public GroupBasedWear(int memSize, int sectorsPerBlock, int sectorSize, int failCount, int groupSize){
        numFree = 15; // make divisible by groupSize + 1
        extraMap = new ArrayList<int[]>(16);
        free = new ArrayList<Block>(numFree);
        this.sectorsPerBlock = sectorsPerBlock;
        int numBlocks = memSize / (sectorsPerBlock * sectorSize);
        map = new int[numBlocks-numFree];
        inverseMap = new int[numBlocks];

        numGroups = numBlocks / groupSize;
        System.out.println("numGroups "+ numGroups);
        groups = new ArrayList<Group>(numGroups);

        numErases=0;

        createMemory(memSize, sectorsPerBlock, sectorSize, failCount, groupSize);
    }

    void createMemory(int memSize, int sectorsPerBlock, int sectorSize, int failCount, int groupSize){

        memory = new ArrayList<Block>(memSize);
        int numBlocks = memSize / (sectorsPerBlock * sectorSize);
        int logicalBlocks = numBlocks - numFree -1;

        for(int i=0; i<logicalBlocks; i++){
            Block temp = new Block(sectorsPerBlock, i, failCount, sectorSize);
            memory.add(temp);
            map[i] = i;
            inverseMap[i] = i;
        }

        int j = 0;
        while (j<logicalBlocks){

    }

}
ArrayList<Block> blocks = new ArrayList<Block>(groupSize);
for(int k=0; k<groupSize; k++)
{
    blocks.add(memory.get(j));
    j++;
}
Group tempG = new Group(groupSize, blocks);
groups.add(tempG);
}

determineYoungest2();
for(int l=logicalBlocks; l<(numBlocks-1); l++)
{
    Block temp2 = new Block(sectorsPerBlock, l, failCount, sectorSize);
    memory.add(temp2);
    free.add(temp2);
    inverseMap[l] = -1;
}
Block temp3 = new Block(sectorsPerBlock, numBlocks-1, failCount, sectorSize);
memory.add(temp3);
extra = temp3;
inverseMap[numBlocks-1] = -1;

int calcLBN(int lsn){
    int lbn = lsn / sectorsPerBlock;
    return lbn;
}

int calcOffset(int lsn){
    int offset = lsn % sectorsPerBlock;
    return offset;
}

int translate(int lbn){
    int pbn = map[lbn];
    return pbn;
}

String read(int lsn){
    int lbn = calcLBN(lsn);
    int offset = calcOffset(lsn);
    int pbn = this.translate(lbn);
    return memory.get(pbn).getData(offset);
}

void write(int lsn, String data){
    int lbn = calcLBN(lsn);
    int offset = calcOffset(lsn);
    int pbn = this.translate(lbn);
    //if empty, write to sector
    Block b = memory.get(pbn);
    if(b.getSector(offset).isEmpty()){
        b.setData(offset, data);
    } else {
        //check if any free in use
        int extra2 = -1;
        int ebn = -1;
        if(!extraMap.isEmpty()){
            //TODO: Implement logic
        }
    }
}
for(int n=0; n<extraMap.size(); n++){
    if(extraMap.get(n)[0] == pbn)
        extra2 = n;
}

if(extra2 != -1){
    ebn = extraMap.get(extra2)[1];
    Block c = memory.get(ebn);
    if(c.getSector(offset).isEmpty()){
        c.setData(offset, data);
    } else{
        c.getSector(offset).invalidate();
        merge(pbn, ebn);
        this.reclaim(pbn);
        this.write(lsn, data);
    }
}

//check if free available
if(!free.isEmpty()){
    Block ex = free.remove(0);
    int exbn = ex.getPBN();
    int[] tempArray = new int[2];
    tempArray[0] = pbn;
    tempArray[1] = exbn;
    extraMap.add(tempArray);
    ex.setData(offset, data);
    b.getSector(offset).invalidate();
}

//invalidate, reclaim, write
b.getSector(offset).invalidate();
this.reclaim(pbn);
this.write(lsn, data);

void merge (int pbn, int ebn){
    Block b = memory.get(pbn);
    Block c = memory.get(ebn);
    int size = b.getSize();
    Sector temp;
    Group g = null;
    for(int n=0; n<groups.size(); n++){
        Group tGroup = groups.get(n);
        if(tGroup.getBlocks().contains(new Block(pbn))){
            g = groups.get(n);
            break;
        }
    }

    //copy from b and c to extra
    for(int i=0; i<size; i++){
        temp = b.getSector(i);
        if(temp.isValid() && !temp.isEmpty()){
            extra.setData(i, temp.getData());
        }
    }
    for(int i=0; i<size; i++){
        temp = c.getSector(i);
    }
}
if(temp.isValid() && !temp.isEmpty()){
    extra.setData(i, temp.getData());
}

// make b new extra
int lbn = inverseMap[pbn];
inverseMap[pbn] = -1;
int newPBN = extra.getPBN();
inverseMap[newPBN] = lbn;
map[lbn] = newPBN;
g.swapBlock(b, extra);
b.erase();
extra = b;
numErases++;

// c back to free list
int num = -1;
for(int n=0; n<extraMap.size(); n++){
    if(extraMap.get(n)[0]==pbn)
        extraMap.remove(n);
}
c.erase();
inverseMap[c.getPBN()] = -1;
free.add(c);
numErases++;

String read(int lbn, int offset){
    int pbn = this.translate(lbn);
    return memory.get(pbn).getData(offset);
}

void determineYoungest(){
    Group g;
    double avg = 0.0;
    double tempAvg = 0.0;
    g = groups.get(0);
youngest = g;
    avg = g.getPAvg();
    for(int i=1; i<(numGroups-numFree-1) ; i++){
        g = groups.get(i);
        tempAvg = g.getPAvg();
        if(tempAvg < avg){
            avg = tempAvg;
            youngest = g;
        }
    }
}

void reclaim(int pbn){
    Block b = memory.get(pbn);
    Group g = null;
    for(int n=0; n<groups.size(); n++){
        Group tGroup = groups.get(n);
        if(tGroup.getBlocks().contains(new Block(pbn))){
            g = groups.get(n);
        }
    }
}
if((b.getEraseCount()-youngest.getpAvg())>THRESHHOLD){
    Block update = youngest.getBlock();
    if(update.getPBN()==b.getPBN())
        update = youngest.getBlock();

    int size = b.getSize();
    Sector temp;

    youngest.updateAvg(update.getEraseCount(), extra.getEraseCount());
    for(int i=0; i<size; i++){
        temp = update.getSector(i);
        if(temp.isValid() && !temp.isEmpty()){
            extra.setData(i, temp.getData());
        }
    }
    int lbn = inverseMap[update.getPBN()];
    inverseMap[update.getPBN()] = -1;
    int newPBN = extra.getPBN();
    inverseMap[newPBN] = lbn;
    map[lbn] = newPBN;
    youngest.swapBlock(update, extra);
    update.erase();
    extra = update;
    numErases++;}

} else {
    g.updateAvg(b.getEraseCount(), extra.getEraseCount());
    int size = b.getSize();
    Sector temp;
    for(int i=0; i<size; i++){
        temp = b.getSector(i);
        if(temp.isValid() && !temp.isEmpty()){
            extra.setData(i, temp.getData());
        }
    }
    int lbn = inverseMap[pbn];
inverseMap[pbn] = -1;
int newPBN = extra.getPBN();
inverseMap[newPBN] = lbn;
map[lbn] = newPBN;
g.swapBlock(b, extra);
b.erase();
extra = b;

/*if(youngest.getBlocks().contains(extra)){
   this.determineYoungest();
}*/
numErases++;

public void stats(){
   System.out.println("Printing Stats...");
   System.out.println("Total Erases: " + numErases);
   double sum = 0;
   int wornOut = 0;
   int ec = 0;
   Block b;
   //calc avg, wornOut
   for(int i=0; i<memory.size(); i++){
      b = memory.get(i);
      ec = b.getEraseCount();
      sum += ec;
      if(b.getWear())
         wornOut++;
   }
   double avg = sum / memory.size();
   System.out.println("Average Erase Count: " + avg);
   double dev = 0;
   double devSum = 0;
   for(int j=0; j<memory.size(); j++){
      b = memory.get(j);
      dev = b.getEraseCount() - avg;
      dev = dev * dev;
      devSum += dev;
   }
   dev = devSum / memory.size();
   double stndDev = Math.sqrt(dev);
   System.out.println("Standard Deviation: " + stndDev);
   System.out.println("Number of Worn Out Blocks: " + wornOut);
}

6. Sector.java

package flashSim;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class Sector {
    private boolean valid;
    private boolean empty;
    private String data;
    private int size;
    private String fileName;

    public Sector(int size, int pbn, int i) {
        this.size = size;
        valid = true;
        empty = true;
        data = "";
        fileName = "data/" + pbn + "." + i + ".data";
    }

    void invalidate() {
        valid = false;
    }

    void validate() {
        valid = true;
    }

    void erase() {
        validate();
        data = "";
        empty = true;
        valid = true;
    }

    void setData(String data) {
        empty = false;
        try {
            BufferedWriter out = new BufferedWriter(new FileWriter(fileName));
            out.write(data);
            out.close();
        } catch (IOException e) {
            System.out.println("Error writing to " + fileName);
        }
    }

    String getData() {
        data = "";
        String temp;
        try {
            BufferedReader in = new BufferedReader(new FileReader(fileName));
            while ((temp = in.readLine()) != null) {
                data += temp;
            }
            in.close();
        } catch (IOException e) {
            System.out.println("Error reading " + fileName);
        }
    }
}
7. SimpleEraseCounter.java

```java
package flashSim;

import java.util.ArrayList;
import java.util.StringTokenizer;
import java.io.IOException;
import java.io.FileReader;
import java.io.Scanner;

//based on technique patented by Lofgren et al

public class SimpleEraseCounter {
    private static final int THRESHHOLD = 1000;
    private int[] map;
    private int[] inverseMap;
    private ArrayList<Block> memory;
    private Block extra;
    private Block least;
    int sectorsPerBlock;
    private int numErases;
    private int numSec;

    public SimpleEraseCounter(int memSize, int sectorsPerBlock, int sectorSize, int failCount) {
        System.out.println("Threshold: " + THRESHHOLD);
        int numBlocks = memSize / (sectorsPerBlock * sectorSize);
        this.sectorsPerBlock = sectorsPerBlock;
        numSec = numBlocks * sectorsPerBlock;
        map = new int[numBlocks - 1]; //1 extra
        inverseMap = new int[numBlocks];
        numErases = 0;

        createMemory(memSize, sectorsPerBlock, sectorSize, failCount);
    }

    void createMemory(int memSize, int sectorsPerBlock, int sectorSize, int failCount) {
        memory = new ArrayList<Block>((memSize);
        int numBlocks = memSize / (sectorsPerBlock * sectorSize);
        int logicalBlocks = numBlocks - 1;

        for (int i = 0; i < logicalBlocks; i++) {
            Block temp = new Block(sectorsPerBlock, i, failCount, sectorSize);
            memory.add(temp);
            map[i] = i;
            inverseMap[i] = i;
            least = temp;
        }
    }
}
```
Block temp2 = new Block(sectorsPerBlock, numBlocks-1, failCount, sectorSize);
memory.add(temp2);
extra = temp2;
inverseMap[numBlocks-1] = -1;

int calcLBN(int lsn){
    int lbn = lsn / sectorsPerBlock;
    return lbn;
}

int calcOffset(int lsn){
    int offset = lsn % sectorsPerBlock;
    return offset;
}

int translate(int lbn){
    int pbn = map[lbn];
    return pbn;
}

void write(int lsn, String data){
    int lbn = calcLBN(lsn);
    int offset = calcOffset(lsn);
    int pbn = this.translate(lbn);
    //if empty, write to sector
    Block b = memory.get(pbn);
    if(b.getSector(offset).isEmpty()){
        b.setData(offset, data);
    } else {
        //invalidate, reclaim, write
        b.getSector(offset).invalidate();
        this.reclaim(pbn);
        this.write(lsn, data);
    }
}

String read(int lsn){
    int lbn = calcLBN(lsn);
    int offset = calcOffset(lsn);
    int pbn = this.translate(lbn);
    return memory.get(pbn).getData(offset);
}

void determineLeast(){
    int pbn;
    int eraseCount;
    int tempEC;
    int size = memory.size();
    Block b = memory.get(0);
    pbn = b.getPBN();
    eraseCount = b.getEraseCount();
    for(int i=1; i<size; i++){
        b = memory.get(i);
        tempEC = b.getEraseCount();
        if(b.getPBN() != extra.getPBN()){
            if(tempEC < eraseCount){
                pbn = i;
                eraseCount = tempEC;
            }
        }
    }
}
void reclaim(int pbn){
    Block b = memory.get(pbn);
    if((b.getEraseCount()-least.getEraseCount())>THRESHOLD){
        int size = b.getSize();
        //least -> extra
        Sector temp;
        for(int i=0; i<size; i++){
            temp = least.getSector(i);
            if(temp.isValid() && !temp.isEmpty()){
                extra.setData(i, temp.getData());
            }
        }
        int lbn = inverseMap[least.getPBN()];
        inverseMap[least.getPBN()] = -1;
        int newPBN = extra.getPBN();
        inverseMap[newPBN] = lbn;
        map[lbn] = newPBN;
        least.erase();
        extra = least;
        numErases++;
        this.determineLeast();
    } else {
        int size = b.getSize();
        Sector temp;
        for(int i=0; i<size; i++){
            temp = b.getSector(i);
            if(temp.isValid() && !temp.isEmpty()){
                extra.setData(i, temp.getData());
            }
        }
        int lbn = inverseMap[pbn];
        inverseMap[pbn] = -1;
        int newPBN = extra.getPBN();
        inverseMap[newPBN] = lbn;
        map[lbn] = newPBN;
        b.erase();
        extra = b;
        numErases++;
    }
}
if(b.getPBN()==least.getPBN()){
    this.determineLeast();
    numErases++;
}

public void stats(){
    System.out.println("Printing Stats...");
    System.out.println("Total Erases: " + numErases);
    double sum = 0;
    int wornOut = 0;
    int ec = 0;
    Block b;
    //calc avg, wornOut
    for(int i=0; i<memory.size(); i++){
        b = memory.get(i);
        ec = b.getEraseCount();
        sum += ec;
        if(b.getWear())
            wornOut++;
    }
    double avg = sum / memory.size();
    System.out.println("Average Erase Count: " + avg);
    double dev = 0;
    double devSum = 0;
    for(int j=0; j<memory.size(); j++){
        b = memory.get(j);
        dev = b.getEraseCount() - avg;
        dev = dev * dev;
        devSum += dev;
    }
    dev = devSum / memory.size();
    double stndDev = Math.sqrt(dev);
    System.out.println("Standard Deviation: " + stndDev);
    System.out.println("Number of Worn Out Blocks: " + wornOut);
}

public void printMemory(){
    System.out.println("Printing Memory...");
    Block b;
    for(int i=0; i<memory.size(); i++){
        System.out.println(memory.get(i));
    }
    System.out.println("Printing Extra...");
    System.out.println(extra);
    System.out.println("Printing Least...");
    System.out.println(least);
}
}
8. Simulator.java

package flashSim;

import java.io.FileReader;
import java.io.IOException;
import java.util.Scanner;
import java.util.StringTokenizer;

public class Simulator {
    public static void main(String[] args) {
        //*************
        //Select memory
        //*************

        //***Simple Erase Counter***
        System.out.println("SEC - SLC Large 1");
        SimpleEraseCounter mem = new SimpleEraseCounter(512000, 64, 2, 100000);
        int numSectors = 255936;

        //System.out.println("SEC - SLC Large 2");
        //SimpleEraseCounter mem = new SimpleEraseCounter(1024000, 64, 2, 100000);
        //int numSectors = 511936;

        //System.out.println("SEC - MLC 1");
        //SimpleEraseCounter mem = new SimpleEraseCounter(512000, 128, 2, 10000);
        //int numSectors = 255872;

        //System.out.println("SEC - MLC 2");
        //SimpleEraseCounter mem = new SimpleEraseCounter(1024000, 128, 2, 10000);
        //int numSectors = 511872;

        //***Group-Based***

        //System.out.println("GB - SLC 1");
        //GroupBasedWear mem = new GroupBasedWear(512000, 64, 2, 100000, 4);
        //int numSectors = 254976;

        //System.out.println("GB - SLC 2");
        //GroupBasedWear mem = new GroupBasedWear(1024000, 64, 2, 100000, 4);
        //int numSectors = 510976;

        //System.out.println("GB - MLC 1");
        //GroupBasedWear mem = new GroupBasedWear(512000, 128, 2, 10000, 4);
        //int numSectors = 253952;

        //System.out.println("GB - MLC 2");
        //GroupBasedWear mem = new GroupBasedWear(1024000, 128, 2, 10000, 4);
        //int numSectors = 509952;

        //***BAST***

        //System.out.println("BAST - SLC 1");
        //BAST mem = new BAST(512000, 64, 2, 100000);
int numSectors = 253952; /*
/*System.out.println("BAST - SLC 2");
BAST mem = new BAST(1024000, 64, 2, 100000);
int numSectors = 509952; */

/*System.out.println("BAST - MLC 1");
BAST mem = new BAST(512000, 128, 2, 10000);
int numSectors = 251904; */

/*System.out.println("BAST - MLC 2");
BAST mem = new BAST(1024000, 128, 2, 10000);
int numSectors = 507904; */

// *** FAST *** //

/*System.out.println("FAST - SLC 1");
FAST mem = new FAST(512000, 64, 2, 100000);
int numSectors = 253952; */

/*System.out.println("FAST - SLC 2");
FAST mem = new FAST(1024000, 64, 2, 100000);
int numSectors = 509952; */

/*System.out.println("FAST - MLC 1");
FAST mem = new FAST(512000, 128, 2, 10000);
int numSectors = 251904; */

/*System.out.println("FAST - MLC 2");
FAST mem = new FAST(1024000, 128, 2, 10000);
int numSectors = 507904; */

************
// Select trace
************
String inputFile = "pic.diskmon";
//String inputFile = "mov.diskmon";
//String inputFile = "mp3.diskmon";
//String inputFile = "mp3.diskmon";

************
// Set repeat
************
int repeat = 1;

String line;
StringTokenizer st;
int sector = 0;
String data = "";
int lsn = 0;
int count = 0;
int size = 0;
String request = "";
try {

for(int i=repeat; i<10; i++){
    FileReader reader = new FileReader(inputFile);
    Scanner in = new Scanner(reader);
    while(in.hasNextLine()){
        line = in.nextLine();
        st = new StringTokenizer(line);
        while(st.hasMoreTokens()){
            count++;
            st.nextToken(); //#
            st.nextToken(); //time
            st.nextToken(); //duration
            st.nextToken(); //disk
            request = st.nextToken(); //request
            sector = Integer.parseInt(st.nextToken());
            size = Integer.parseInt(st.nextToken());
        }
        lsn = sector % numSectors;
        //write for size
        int temp2 = 0;
        for(int j=0; j<size; j++){
            temp2 = (lsn+j) % numSectors;
            if(request.equalsIgnoreCase("write")){
                mem.write(temp2, data);
            }
            if(request.equalsIgnoreCase("read")){
                mem.read(temp2);
            }
        }
    }
} catch (IOException e){
    System.out.println("Error reading file: " + e);
}
System.out.println("Number of writes: " + count);

//***********
//print stats
//***********
mem.stats();
System.out.println("Completed.");
}