Analysis of Garbage Collector Algorithms in Non-Volatile Memory Devices

THESIS

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Ananth Mahadevan Muralidharan

Graduate Program in Computer Science and Engineering

The Ohio State University

2013

Dissertation Committee:

Dr. Rajiv Ramnath, Advisor

Dr. Jayashree Ramanathan
Abstract

Non-volatile memory devices or flash, even with many advantages, still have a few problems such as the inability to update data in place. This necessitates the need for a garbage collector (GC) that can collect active data and create space by erasing flash blocks. However this is a very costly operation that increases the write latency thereby lowering the efficiency of the flash device. The frequency at which the GC is invoked by the underlying file system depends on the data’s traffic pattern as well as the fullness of the device. It is therefore important to study different GC algorithms for different traffic patterns and at varying fullness levels in order to find the most efficient one for a particular situation. In this report we study the efficiency of byte address non-volatile memory devices (such as NOR), under varying traffic patterns.

We study the algorithms using simulations coded in Matlab. A simulator for the flash file system as well as the GC algorithms and various applications traffic was developed and used for the study. We compare and contrast the efficiency and the time taken for the GCs at utilization levels ranging from 2% to 98%. We also model some of the algorithms analytically and find that our analytical results match our simulations. The performance results for five different GC algorithms for flash devices for three traffic/access patterns are presented in this report. The access patterns include long-tailed, uniform and bimodal distributions. The algorithms studied are a round-robin style first in first out (FIFE), a greedy least active clean (LAC), 3-Generation (3-Gen) GC, N-
Generation (N-Gen) GC (a generalized generation algorithm) and Eta-N-Generation (Eta-N-Gen) GC (a variation on N-Gen).

The results indicate that round-robin style GC algorithm (FIFE) and greedy algorithm (LAC) perform better in most of the scenarios than generational algorithms. This is counter-intuitive to the existing norms. LAC slightly underperforms the FIFE under heavy flash utilization. For long-tailed traffic – the canonical use case for generational algorithms – FIFE and LAC still perform better than generational algorithms. The reason is that, it is non-trivial to configure a generational algorithm to get the optimum performance for a particular traffic pattern. To optimize performance, the radio of the size of subsequent generations should be the same as ratio between cold data and the rest of the data. Since in most application cases we do not know this a priori, static optimal configuration of generational algorithms is impossible. However an adaptive algorithm which changes allocations between generations on the fly could achieve better efficiency. Further we find that for better efficiency, at low levels of utilization it is important to isolate “cold’ data well, but at higher utilization identifying and handling hot data (i.e., never move the hot data) is important.

Results from our study suggest that FIFE might work well for most of the application scenarios.
This document is dedicated to my father Sri. N. Mahadevan.
Acknowledgments

The wonderful time I had at OSU learning new things and meeting intelligent and interesting people would not have been possible without the support of my parents, brother, rest of my family and Maya foundation. Everything I have accomplished so far, I owe it to the sacrifices my parents have made for me. I am grateful to Dr. M.B. Anand who sowed the thought of doing a Masters’ degree into my mind. He helped me at so many different stages that there is not enough space to mention all of them here. Thanks also to my cousin Rajee for helping me during the application process and during my job search. I am equally indebted to Jeremy Bahr, Jossy Tonio, Vani Dharmalingam, William Miteff, Irma Resendez and Tamrat Tewoldeberhan for writing the recommendation letters.

Thanks to my advisor Dr. Rajiv Ramnath who during our weekly meetings, guided me in the right direction by asking lot of questions and helped me get the answers for them as well. I am grateful to Dr. Mukundan Sridharan (The Samraksh Company) who mentored and guided me during the course of this work. My sincere gratitude also to Dr. Kenneth Parker and Dr. Anish Arora (The Samraksh Company) for giving me the opportunity to work on this wonderful project. I am thankful to the professors and friends who helped me when I was stuck at a difficult problem. Last but not the least, thanks to The Ohio State University and everyone behind running this institution that helped me achieve what I came here for.
Vita

1998..............................................................Kikani Higher Secondary School

2002.............................................................B.E. ECE, Bharathiar University

Sep 2011 to May 2013 ..............................MS - CSE, The Ohio State University

Feb 2012 to Dec 2012 ......................Graduate Research Associate, CETI, The Ohio State University

Fields of Study

Major Field: Computer Science and Engineering
Table of Contents

Abstract........................................................................................................................................... ii

Acknowledgments........................................................................................................................... v

Vita................................................................................................................................................... vi

List of Tables ................................................................................................................................... viii

List of Figures .................................................................................................................................. ix

Chapter 1:  Introduction.................................................................................................................... 1

Chapter 2:  Background and Related Work...................................................................................... 5

Chapter 3:  Methodology.................................................................................................................. 12

Chapter 4:  Analytical Modeling..................................................................................................... 24

Chapter 5:  Simulation results.......................................................................................................... 37

Chapter 6:  Conclusion and Future Work ....................................................................................... 57

Bibliography ..................................................................................................................................... 59
List of Tables

Table 1: Frequently used terms................................................................................................. 1
List of Figures

Figure 1: Log structured file system ................................................................. 7
Figure 2: External fragmentation ......................................................................... 15
Figure 3: Internal fragmentation ......................................................................... 15
Figure 4: Matlab simulation flow-chart ............................................................... 16
Figure 5: Access of records for uniform distribution before sorting .................. 17
Figure 6: Access of records after sorting ............................................................. 17
Figure 7: Access of records for Pareto distribution ............................................... 18
Figure 8: Access of records for bimodal distribution ............................................ 18
Figure 9: Block fragmentation in a FIFE garbage collector .................................. 20
Figure 10: Block fragmentation in a 3-Generation GC ......................................... 21
Figure 11: Block fragmentation in a N-Generation GC ........................................ 22
Figure 12: Theoretical and simulation results for FIFE - uniform distribution ......... 28
Figure 13: Theoretical and simulation results for FIFE - uniform distribution
(exponential) ........................................................................................................ 29
Figure 14: 1-Generation GC theoretical model at different accesses .................... 35
Figure 15: 2-Generation GC theoretical model at different accesses .................... 35
Figure 16: 3-Generation GC theoretical model at different accesses .................... 36
Figure 17: N-Generation GC theoretical model at different accesses .................. 36
Figure 18: Efficiency vs. fullness for FIFE GC for uniform distribution ............... 39
Figure 19: Read/write/erase costs vs. fullness for FIFE GC ........................................ 39
Figure 20: Efficiency vs. fullness for LAC GC for uniform distribution ....................... 39
Figure 21: Read/write/erase costs vs. fullness for LAC GC ...................................... 39
Figure 22: Efficiency vs. fullness for 3-Gen GC (block split count ratio of 4:3:1) .......... 41
Figure 23: Read/write/erase costs vs. fullness for 3-Gen GC .................................... 41
Figure 24: Efficiency vs. fullness for 3-Gen GC (block split count ratio of 12:3:1) ...... 41
Figure 25: Efficiency vs. fullness for N-Gen GC ..................................................... 42
Figure 26: Read/write/erase costs vs. fullness for N-Gen GC .................................... 42
Figure 27: Efficiency vs. fullness for FIFE GC ....................................................... 44
Figure 28: Read/write/erase costs vs. fullness for FIFE GC ..................................... 44
Figure 29: Efficiency vs. fullness for LAC GC ....................................................... 44
Figure 30: Read/write/erase costs vs. fullness for LAC GC ..................................... 44
Figure 31: Efficiency vs. fullness (FIFE and LAC) - 500,000 runs ................................ 45
Figure 32: Efficiency vs. fullness for 3-Gen GC ..................................................... 47
Figure 33: Read/write/erase costs vs. fullness for 3-Gen GC .................................... 47
Figure 34: Efficiency vs. fullness for N-Gen GC ..................................................... 47
Figure 35: Read/write/erase costs vs. fullness for N-Gen GC .................................... 47
Figure 36: Efficiency vs. fullness for Eta-N-Gen GC .............................................. 48
Figure 37: Read/write/erase costs vs. fullness for Eta-N-Gen GC .............................. 48
Figure 38: Efficiency vs. fullness for uniform distribution ........................................ 51
Figure 39: Time taken for GC vs. fullness for uniform distribution ............................ 51
Figure 40: Efficiency vs. fullness for Pareto distribution (100,000 runs) .................... 51
Figure 41: Time taken for GC vs. fullness for Pareto distribution (100,000 runs) ........... 51
Figure 42: Efficiency vs. fullness for Pareto distribution (500,000 runs) .................... 52
Figure 43: Time taken for GC vs. fullness for Pareto distribution (500,000).................. 52
Figure 44: Efficiency vs. fullness for bimodal distribution (500,000) ......................... 52
Figure 45: Time taken for GC vs. fullness for bimodal distribution (500,000)............. 52
Figure 46: FIFE GC at different data accesses .......................................................... 55
Figure 47: LAC GC at different data accesses ............................................................ 55
Figure 48: 1-Gen GC at different data accesses ......................................................... 55
Figure 49: 2-Gen GC at different data accesses ......................................................... 55
Figure 50: 3-Gen GC at different data accesses ......................................................... 56
Figure 51: N-Gen GC at different data accesses ....................................................... 56
Chapter 1: Introduction

Non-volatile memory devices are increasingly being used as primary storage devices. They have many advantages such as retaining data\(^1\) even during loss of power and low latency. They can be placed under two major categories – NAND or NOR. In general, NAND flashes have larger storage capacities but can only be addressed in blocks and are typically used as storage devices. NOR flashes on the other hand can be byte addressed, with a fast read time, and can be used either as storage or as primary memory from which code can be executed in embedded devices. Unlike RAM, where data can be over-written, flash devices cannot update record in-place. This means that when data has to be updated, the earlier one is marked invalid and the newer one is written to an empty region. In order to free up the user from managing the invalid data, a flash data management/file system is necessary for most flash devices.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>GC</td>
<td>Garbage Collection</td>
</tr>
<tr>
<td>NVM</td>
<td>Non-Volatile Memory</td>
</tr>
</tbody>
</table>

| Table 1: Frequently used terms |

Flash devices are typically divided into multiple blocks (sometimes they are also called ‘sectors’) into which data records (aka records) of varying sizes are stored. An

\(^1\) For purposes of this paper, data will be referred to as a “record”.

1
important component of a flash file system is the garbage collector (GC). A location needs to be erased before it can be written again and in most flashes only a whole block can be erased at a time. It is the job of a GC to select the block(s) the needs to be cleaned whenever space needs to be created on the device. If the select block(s) has any valid data, the GC also needs to select the block to which the active records will be moved. But the move and erase operations are costly in terms of energy and time, and the underlying algorithm can have a significant impact on the overall performance and efficiency of the system.

The frequency of GC invocation depends on the data traffic pattern. Depending on the application, data objects could have varying life-times. An application can update the same set of data records in periods of short bursts resulting in short life time of the data-objects and such data are classified as “hot” data. On the other hand some of the data might be updated very infrequently resulting in a long life-time and are called as “cold” data. Some applications might access all data records more or less uniformly and might have very little hot or cold data.

Different GC algorithms might work well for different kinds of traffic. Hence it is important to study the performance of a GC under varying traffic conditions. Thus, a good flash file system, of which GC is a critical part, usually has a number of conflicting requirements such as, minimization of read/write latency, maximization of energy efficiency (decrease the amount of data moved), wear-leveling (making sure all blocks are erased approximately same number of times), etc. This report is primarily concerned with the analysis of the efficiency of garbage collection algorithms for NOR flashes and
thereby verifying if flashes can be used as primary storage devices. Here we study the performance of different GC algorithms against different traffic patterns.

In order to simulate different traffic patterns we created applications that generate data that follow uniform, Pareto (long-tailed) and bimodal distributions. These distributions cover a large variety of application traffic and help us understand the performance of the GCs studied. A large number of GCs have been invented and studied in literature. Here we study some of the popular ones and have also innovated a few new algorithms. The most commonly used is an algorithm that always cleans blocks in sequence, starting with the very first one written. We call this algorithm first insert first erase (FIFE). The least active clean (LAC) is a type of greedy algorithm that always chooses the block that has the least amount of active records. We also consider several variants of the generational GC algorithms, which try to isolate the hot and cold data records, so that the cold data is not moved very often. Generally, a generational algorithm, divides the entire flash into a number of regions where data of similar life-times are stored together. We have studied 3 different generational algorithms in this report, 3-Gen, N-Gen and Eta-N-Gen. In 3-Gen, the flash is divided into 3 generations with hot data occupying the 1st generation and cold data, the lower two. Eta-N-Gen and N-Gen algorithms consider all blocks of the flash to be a generation with the coldest data occupying the last generation.

**Summary of Results:** In this report, we present our results of analyzing and measuring the performance of five different GC algorithms for three different data patterns. The performance is measured using various parameters such as write cost, erase cost and...
Efficiency. We also present an analytical model for some of the algorithms and explain the reason behind the results we have obtained.

We find that the LAC and FIFE algorithms outperform the generational algorithms in most cases and LAC slightly underperforms the FIFE under heavy flash utilization. We also find that even for long-tailed traffic – the canonical use case for generational algorithms – FIFE and LAC perform better than generational algorithms. The reason is that it is non-trivial to configure a generational algorithm to get the optimum performance for a particular traffic pattern. To optimize performance, the ratio of the size of subsequent generations should be the same as ratio between cold data and the rest of the data. Since in most application cases we do not know this a priori, static optimal configuration of generational algorithms is impossible. However an adaptive algorithm which changes allocations between generations on the fly could achieve better efficiency.

The rest of the report is organized as follows: Section 2 gives details on related work. Section 3 outlines the algorithms studied and our experimental methodology. Section 4 presents some our analytical models for algorithms under different traffic models. Section 5 presents our results and in Section 6 we present our concluding remarks.
Chapter 2: Background and Related Work

*Flash Characteristics:*

Flash memory is a cost-effective solid-state non volatile storage technology that is widely being used in mobile and other embedded devices. Compared to traditional hard disk drives, flash devices have small size and low power consumption. There are 2 major classes of flash devices available today - NAND and NOR. NAND flash has a very small cell size and is mainly used for storage of large amounts of data as its cost-per-bit is very low compared to NOR. NAND flash is organized into blocks and each block is divided into pages. Block size of a typical NAND flash is 16KB and the page size can be 512B (32 pages in a block). Read and write operations take place on pages whereas erase happens on blocks. NOR flash on the other hand, has short read times and allow individual bits to be set. This has the advantage that it can execute code, a feature called eXecute-In-Place (XIP). [1]

Flash devices generally have fast read accesses but have very slow write accesses. Every block on a NAND flash can be written-to or erased only a limited number of times (in order of 1 million times or $10^6$ cycles). Writing to or erasing a block beyond this limit can result in “wearing out” of the flash, which can lead to write failures or can return invalid data for read operations. Data cannot be written over already written areas (called in-place update). Data can only be written to areas that have already been erased.
Therefore if data has to be updated on the flash, it must be written to a new area and the old data is marked invalid. This is called out-of-place-update. After many cycles of writes, the entire flash is fragmented with valid and invalid data and the flash quickly runs out of space for new data. This is when the system invokes a garbage collector whose task is to collect all valid data in a block, write it to a new location and erase the old block. This paves the way for new data to be written to the flash. But this operation of moving data to a new location and erasing a block is costly; hence a good GC algorithm has to keep moves and delete operations to a minimum.

**Summary of Existing Literature**

**Popular flash File Systems:** Due to increasing capacity and reduction in accessing times of RAM, reads have become quite fast and writes take up bulk of the time in a typical embedded device. Therefore there is a need for a file system that provides fast write access. A log\(^2\) structured file system (LFS) is one such system that writes data sequentially in a log-like manner [2] as shown in Figure 1. This sequential nature of writing records reduces the write time. But in order for a log-structured file system to operate efficiently, it needs to have large amounts of free space to write new data. When the file system runs out of space for new records, it invokes the garbage collector which creates space by consolidating active data from fragmented data blocks and then erasing the blocks freed.

---

\(^2\) Updates to data are written sequentially in a continuous stream
A journaling flash file system 2 (JFFS2) – another popular flash file system – divides the flash into regions called blocks and writes records to journals inside the blocks. When a block becomes full, it searches the entire flash for another block. When JFFS2 is mounted after a shutdown, the entire device is scanned and each record found is recorded into an in-memory tree-like data structure. When an updated version of a record is found, the earlier version of the record in the tree is updated. Thus the tree that is built finally, describes the device exactly the way it was before the shutdown occurred. However, the cost to build and maintain this data structure increases linearly with the size of the file system. Hence JFFS2 is not feasible for memory of sizes greater than a gigabyte [3]. Unlike JFFS2 which has a high initial cost of building the file system tree at mount time, LogFS stores the tree on the flash device itself. Placing the tree on the device reduces the mount time and the run-time memory requirements [4].

Yet another flash file system (YAFFS2) was designed for the newer NAND flash devices. It is a log-structured file system which has a good write performance. Newer devices have the constraint that pages should be written to only once. YAFFS2
overcomes this limitation by marking every updated page with a monotonically increasing sequence number. When YAFFS2 scans the device and encounters devices with the same block number, it chooses the one with the highest sequence number [5].

UBIFS which stands for unsorted block images is similar to JFFS2 and performs better for large NAND flashes. UBIFS maintains two trees: one to index all nodes and another to keep track of free space [3]. Flash-friendly file system (F2FS) introduced by Samsung is based on the log structured file system (LFS) design [6]. Unlike JFFS2 and LogFS, F2FS is focused towards consumer flash storage devices such as SD cards. This file system gathers together records that need to be written, into large sequential writes. This makes it much easier for the underlying flash translation layer\(^3\) to find sufficient space in the flash.

**Popular garbage collection algorithms for heaps:** Given below are the garbage collection algorithms used in three popular virtual machines – Java’s JVM, .NET’s CLR and Android’s Dalvik.

Java’s JVM uses generational garbage collection. While naive GC algorithms examine every object on the heap, Java’s generational algorithm makes use of empirically observed characteristics of the data to reduce the amount of work done. Some objects live for the entire duration of the application while some objects live for a very short duration (a function within a class). To manage such situations, the heap is divided into generations (young and tenured). The young generation is divided into an Eden and

\(^3\) Purpose of the FTL is to provide the functionality of a block device on top of flash memory. It hides the erase-before-write characteristic of a flash and makes the flash memory appear as a disk drive to the overlying system.
two survivor spaces. When the young generation becomes full, the live objects are moved to the survivor space. This is known as a minor collection. When the survivor space becomes full, objects are moved to the tenured space. This is known as a major collection. Allowing more time between minor collections allows short-lived objects to die and hence the overall performance of the GC increases [7].

The Common Language Runtime (CLR) of the .NET framework handles GC by dividing the memory into three generations (generation 0, 1 and 2). Generation 0 is the youngest generation and contains short-lived objects such as temporary variables. Generation 2 contains long-lived objects whereas generation 1 serves as a buffer between the short and long-lived objects. Most of the GC happens in generation 0 and objects that survive the collection are moved further up the generations. The GC has an initial marking phase that creates a list of live objects. A compaction phase follows that moves the objects that have survived the GC to the older generations. Between the mark and compaction phase is the relocating phase that updates the references to the objects that are compacted [8].

Android’s Dalvik Virtual Machine uses a mark-and-sweep or tracing, garbage collection algorithm. The mark-and-sweep algorithm has two phases – mark followed by a sweep. In the mark phase, all accessible objects are marked. In the sweep phase, space occupied by all unmarked objects or garbage are reclaimed. One of the advantages of this algorithm is that it is able to find and collect garbage even when there are reference cycles. But the disadvantage is that the program execution stops when this algorithm runs [9] [10].
Existing literature on non-volatile memory garbage collection: A few of the GC algorithms that have been studied for log-structured file systems are - Greedy, Cost-Benefit analysis and Cost-Age Time (CAT) [11] [12] [13]. Greedy considers those segments for garbage collection that have the least amount of active records or in other words can yield the maximum amount of space. Cost-Benefit considers segments that yield the maximum space but at the same time, are above a certain age. Age-Threshold considers segments for garbage collection that are above a certain threshold for the “age”. When data is moved from one segment to another by GC, the age of the new segment is set to the age of the segment plus one. However, when a segment is erased and then new data is written to it, the age is reset to 0.

CAT claims to reduce the number of erase operations performed on a block while evenly wearing out the flash at the same time. It considers the age of the data, the cleaning cost and the number of times a segment has been erased. A segment is chosen such that the formula

\[ CleaningCost \times \frac{1}{Age} \times CountOfCleaning \]

is minimized. CAT also considers different ways of redistributing data (within the same block and across several blocks). Grouping data across segments based on their hotness degree is similar to generational algorithms. These algorithms can be grouped into two types. On set of algorithms claim that longer the age of a block, the more likely they are to have inactive data. Another set of algorithms claim that the longer the data is not accessed, the higher they can be moved to generations.
Most of the inefficiency comes from thinking of hot data as cold (moving hot data too soon). In generational algorithms, the problem is that a large amount of hot data is being marked as cold. Very soon after the data is moved to a higher generation, data tends to get inactivated. This results in wasted cost.

WECO (WEar COncscious) [14] looks at creating GC techniques that focus on both performance and wear-consciousness. The core principles are to separate hot and cold data to improve performance and to properly select victim blocks\(^4\) to improve performance. The crux of the paper is to have a balanced approach towards wear-levelling and performance. When one is low, the other is increased so that both performance and life-time of the flash is increased. Another approach is to dynamically reorganize data and group them based on their temperature (Hot or Cold) [15]. Frequently accessed data are clustered towards the center and the less accessed data are placed towards the edges.

Greedy GC algorithms have good performance but do not consider wear-levelling of the blocks. Also erasing blocks is one of the costliest operations performed on a flash device. Greedy algorithms can be extended by also considering the number of times a block has been erased and the update time of the pages in each of the blocks [11].

\(^4\) Block that is selected for garbage collection.
Chapter 3: Methodology

The main objective of our study is to understand the algorithm with the maximum efficiency for a certain pattern of data. By observing real-world applications we note that most of them can be mimicked by a combination of statistical distributions such as uniform, Pareto and bimodal. We use these distributions to create data traffic and study the performance of garbage collection under these patterns.

We focus mainly on the cost of write operations, since read is usually very fast and the efficiency of modern flash devices are dominated by writes. We took a two-pronged approach in our study of the algorithms. One was to study the performance analytically by deriving equations and formulae for the efficiency of an algorithm under a given access pattern. The other was to simulate the applications, the flash and the algorithms in Matlab and running different experiments. Various parameters are captured that help in plotting the efficiency chart. Finally the simulation model was compared against the theoretical model.

Performance Metrics: The following performance metrics are used in our study to understand the performance of a GC algorithm.

- Fullness - This is the ratio of the sum of all active records in the flash to the size of the flash. This is an important criterion in determining the performance of a flash and hence is used as the basis for all of our experiments.
• Useful write cost – The size of the record that is passed by the application to the flash file system to be written to the flash.

• Actual write cost – The actual amount of work done by the flash before the record is written. This work could involve moving active records from one block to another block and erasing the earlier block. The more work done by the flash before records are written, less is its efficiency.

• Efficiency - Efficiency is the ratio of the Useful to the Actual Write Cost. This is one of the core criteria that we attempt to measure in our study.

• Erase cost – The total erase cost for all blocks over the life-time of the flash is captured. This helps in comparing the erase cost with the move cost.

• Time taken for an algorithm to complete – The time taken by an algorithm in creating space in the flash is captured. This is used to determine which algorithm performs better as the flash fullness increases.

• Count of erase operations – This is the total count of erase operations that take place on a certain block. This metric is captured for all blocks and gives an indication on the uniformity of erases across all blocks.

**Matlab flash Simulation:** The GC and the applications were implemented in Matlab 2011b. The tests were run on the Ohio supercomputer center’s cluster. The fullness level ranges from 2% to 98% and the simulations were run for 500,000 read/write accesses per fullness level. The simulations are done for all three traffic patterns for all five GC algorithms. We used the rand function in Matlab which generates numbers with uniform distribution. For Pareto, we added a weight (that followed a long-tail distribution) to the
numbers which decides the usage of a record. For Bi-model distribution we first
generated two uniform distributions, weighted them differently, combined them,
normalized and then used them to generate the access patterns.

Our simulations used application which has only write operations, although our
simulator can simulate both reads and writes. After the records are generated, based on a
coin toss (random number between 0 and 1), it can be decided whether the next operation
will be a read or a write. Thus the percentage of read vs write can be varied smoothly
across the entire spectrum in the simulator. Every time the GC is accessed, details such as
amount of bytes moved, blocks erased, are captured. These details are then used to plot
the required graphs.

We assume that the size of a data record can vary from 4 to 244 bytes and the size
of the header is 12 bytes. Therefore the size of a record that gets stored in the flash ranges
from 16 to 256 bytes. A record is stored such that it does not traverse the boundary
between two blocks and is aligned to the block boundaries. If the space remaining in a
block is less than that of the incoming record, a dummy record is used to fill the gap, and
the incoming record is stored in a new block (see Figure 2). Also, if the space at the end
of a block after storing a record is less than the space of the header, more space is
allocated to the record as depicted in Figure 3. This alignment of records to the block
boundaries helps in persistence of records, as after an unexpected shutdown, the list of
records can be recreated by traversing through the blocks. The first entry in every block
will be a record and at the end of the traversal, the system would have created the list of
files and their locations.
Figure 4 shows the work flow in the Matlab simulator. The ‘main’ program controls the number of times the simulation is to be run and which type of application is simulated among other things. It also collects the results returned by the application and generates the graphs. It invokes the application program which contains the implementation for uniform, Pareto or bimodal type of distributions. Based on the toss of a coin, it decides if the next operation will be a read or a write. Every read and write operation invokes the emulator which implements functionalities such as reading/writing a record, maintaining the indices of the records within the flash and so on. When space runs out, the emulator invokes the garbage collector whose primary purpose is to free up space by moving around active records and erasing a block of inactive records. It is successful if it finds sufficient space for the next record to be inserted. It returns an exception back to the emulator if it is not able to find space in the entire flash.
**Application Classes**: A controlled simulation environment was created in order to analyze the performance of different garbage collection algorithms. The application classes considered here – uniform, Pareto and bimodal – do not reflect the traffic pattern in real-world applications, but helps us understand how the algorithms perform under different accesses [2]. Figure 5Error! Reference source not found. indicates the access time of records that follow uniform distribution. The record IDs range from 1 to 100 and
as can be observed, they are accessed randomly. **Figure 6** on the other hand is obtained after sorting the records in increasing order. The graph shows a linear rise in the plot which means that all variables are equally likely to be picked up by the application. The time between accesses of the same variable is also uniform. This means that once a variable is accessed, it will be accessed in an equally likely manner.

![Figure 5: Access of records for uniform distribution before sorting](image1)

![Figure 6: Access of records after sorting](image2)

**Figure 7** indicates the density values for all variables that follow the Pareto distribution. Variables that follow this pattern display temporal locality of access. The graph – where the variables have been sorted - shows an exponential rise in the plot towards the end. This means that variables towards the end are more likely to be picked up by the application (higher density values) than those at the beginning (spatial locality). The time between accessing the same variable also follows a Pareto distribution, which means that a variable once accessed, may be accessed again either frequently or infrequently at another instance in time.
Figure 7: Access of records for Pareto distribution

Figure 8 shows the density values for the variables that follow bimodal distribution. The x-axis denotes the record IDs while the y-axis denotes the density values of the records. There are two distinct peaks where one set of variables has a higher probability of access than the other set. Every variable within the two distributions are accessed uniformly.

Figure 8: Access of records for bimodal distribution
**Algorithms Studied:** As mentioned previously, move and erase operations are costly. Erase operations cannot happen on individual records and has to happen on entire blocks. There are several options to select a block to be garbage collected. It could be the one that was written to first, or the one that yields the maximum space. We considered both of these options and call the algorithms FIFE - first insert first erase and LAC - least active clean. LAC is a type of greedy algorithm that always chooses the block that has the least amount of active records and hence contributes the lowest to the move cost. Based on the “hotness” or “coldness” of the data, the entire flash can be divided into different regions with cold data occupying a certain portion and the hot data the other portion. Such algorithms are called generational algorithms and are quite popular. We created three different generational algorithms – 3-Gen, N-Gen and Eta-N-Gen. In 3-Gen, the flash is divided into three generations with hot data occupying the 1st generation and cold data, the lower two. Eta-N-Gen and N-Gen algorithms consider all blocks of the flash to be a generation with the coldest data occupying the last generation. Five different GC algorithms are considered which has been described below. The first two are round-robin style of GCs while the other three are generational type of algorithms.

**First insert first erase (FIFE):** This makes use of three pointers which facilitate the read/write operations (Figure 9). The “log” pointer always points at the location where the next write operation can take place. The “clean” pointer indicates the location in the flash until which there are no active records. In other words, it points to the next inactive record. The “erase” pointer points at the block which was last erased [16]. The
flash region between the log and the erase pointers is the one where incoming records can be written to.

The data is always written starting from the first block. When the flash reaches a pre-defined fullness level, the GC is invoked. The GC compacts the oldest block and keeps moving forward along the blocks until sufficient space is created to store the record. The GC returns an error when, after traversing the blocks, it is not able to find sufficient space.

![Figure 9: Block fragmentation in a FIFE garbage collector](image)

**Least active clean (LAC):** LAC - least active clean: When the GC is invoked, it finds out the block that has the least number of active records and compacts that block. The compacted block is then used for the next write operation. **Figure 9** that shows the state of the flash at steady state for the FIFE GC indicates that the inactive records are spread evenly throughout the device. Also shown is the log, clean and erase points.
Three Generation (3-Gen): In this GC, the entire flash is divided into three generations. The ratio of the blocks in each generation can be 12:3:1 or 4:3:1. Data is always written to the first generation and when it becomes full, the active records are moved to the 2nd generation. When the 2nd generation becomes full, its active records are moved to the 3rd generation. This allows the GC to separate the hot from the cold data. The intuition is that this reduces the movement of active records which is a drawback of the round-robin algorithms.

Figure 10 shows the state of the flash at steady state for the 3-Gen GC. The flash is divided into three generations with a major portion of the blocks in the first generation. The second and third generations have fewer numbers of blocks. We experimented with block distribution ratios of 12:3:1 and 4:3:1. Note that there are more inactive records in the 1st generation than in the other two generations.

![Figure 10: Block fragmentation in a 3-Generation GC](image)

N-Generation (N-Gen): Each of the blocks in the flash are considered as a generation in this algorithm (see Figure 11). Active records are moved to higher
generations and the coldest record is in the highest generation. Intuitively, this algorithm should work very well for applications that have a clear difference between data accesses (high vs. low locality of reference).

**Figure 11: Block fragmentation in a N-Generation GC**

**Eta-N-Generation (E-N-Gen):** Eta-N-Generation: This is similar to the N-Gen algorithm, except that the amount of data that is moved is decided by a factor called eta ($\eta$). Eta denotes the fullness level of the flash when the GC is invoked. The algorithm, instead of moving all active records from one generation to the other, moves $(1-\eta)$ amount of records. The intuition behind this algorithm is that it reduces the move cost per invocation of the GC, therefore improving the overall efficiency.

**Gamma-N-Generation (Γ-N-Gen):** After observing that the Eta-N-Generation algorithm did not perform as well as the N-Generation algorithm, the eta algorithm was modified to remove its shortcomings such as high move cost and high GC run-time. The result was the Gamma-N-Generation algorithm. Gamma (Γ) is defined as given below:
\[ \Gamma = (1 - \eta)((\text{TotalBlocks} - \text{current generation}) / \text{TotalBlocks}) \]

\( \Gamma \) amount of records are moved from one generation to another during an invocation of the GC.

**Summary of methodology:** The GC that will be implemented will be the one that has the best efficiency for all three traffic patterns. Depending on the outcome of this project, a single GC might be implemented or a hybrid approach will be chosen where one type of GC is used for certain fullness levels and another type for other levels of fullness.

While the current state-of-the-art research in the field of garbage collection algorithms, focus on multiple aspects such as wear-levelling and energy consumption, this project focuses exclusively on the efficiency of an algorithm. This project aims to provide the foundation upon which other aspects of an algorithm can be studied.
Chapter 4: Analytical Modeling

Given below is a mathematical analysis of the first insert first erase algorithm for a uniform and Pareto distribution of accesses. Also analyzed is the 2-generation algorithm for a uniform access pattern.

There are several metrics that indicate the performance of the garbage collection (GC) algorithms. They are plotted and the different graphs that are generated are as follows:

- **Efficiency vs. fullness (normal scale):**
  Efficiency is the primary parameter for deciding which GC is better than the rest. It denotes the amount of work done in order to write data. The lesser the work done, the better is the GC.

- **Efficiency vs. fullness (log scale):**
  This is the log (base 10) representation of the above graph. These denote how quickly or slowly the efficiency of a GC changes.

- **GC time taken vs. fullness:**
  Total time taken by GC beginning from the time it is invoked until sufficient space to store a record is found.

- **Actual read/write/erase costs vs. fullness:**
  Actual write cost is used to calculate the efficiency of the GC algorithm. Comparing the write cost with the erase and read costs and the overall efficiency helps in understanding why a given GC performs the way it does.
**Notation:** The list of the symbols used and their explanation is given below. The symbols used are common for both first insert first erase (FIFE) – uniform and Pareto distribution models.

- **B** – Total blocks in the flash.
- **BlockSize** – Size of a flash block.
- **N** – Count of active records.
- **AvgRecSize** - Average size of a record.
- **Fullness** - Ratio of the sum of all active records in flash to size of flash 
  \[
  \frac{N \times \text{AvgRecSize}}{B \times \text{BlockSize}}.
  \]
- **R_b** - Average records per block \(\frac{\text{BlockSize}}{\text{AvgRecSize}}\).
- **X_i** -- Total number of active records in the \(i\)th block.
- **E(X)** - Expected number of active records in the block that is to be cleaned.
- **A_{max}** - Maximum number of accesses possible before cleaning a block.
- **N_{GC}** - Total number of times GC is invoked.

**FIFE uniform:** The mathematical analysis starts with finding the efficiency for the FIFE algorithm and for the uniform distribution case. This sets the foundation to analyze the rest of the algorithms. There are several variables that need to be calculated before the efficiency of an algorithm is finally determined and the steps explained below provide the details.
Average number of records per block is simply the ratio of the size of a block to the average size of a record. We assume that every block will have one dummy record each and hence it is subtracted as shown in the equation below.

\[ R_b = \left\lfloor \frac{\text{BlockSize}}{\text{AvgRecSize}} \right\rfloor - 1 \]

In order to determine the efficiency, the maximum accesses possible before cleaning a block \( A_{max} \) needs to be found out. In simple terms, this is the product of the average active records per block and \((B-1)\) (since one block is always kept empty).

\[ A_{max} = R_b \times (B - 1) \]

The probability for an existing record to be accessed is \(1/N\) for uniform distribution. Then the probability of not accessing or missing a record, denoted by \(P_m\), is:

\[ P_m = (N - 1)/N \]

Let the expectation of active records per block that is going to be cleaned be denoted by \(E(X)\).

Expectation is then,

\[ E(X) = (P_m)^{A_{max}} \times R_b \]

The intuition behind the equation is that, a record must be missed for \(A_{max}\) times before the block is chosen for cleaning again. Multiplying by \(R_b\) gives the value for all records in a certain block.

The total number of times garbage collector (GC) is invoked decides the total move cost. During one cycle, when active records are moved from one block \(x\) to another \((x+1)\), next invocation of the GC depends on the amount of free space in block \(x+1\). Hence the relation for number of times GC is invoked is:
\[ N_{GC} = \frac{\text{totalWrites}}{(R_b - E(X))} \]

Where \((R_b - E(X))\) gives the number of free slots where records can be stored and \text{totalWrites} is the total possible number of times, records can be written to the flash over its life-time.

Next important factor in determining the efficiency is to find the expectation of the cost of moving records from one block to another. The expected move cost for one record is the product of the expectation of a record to be active and its size. Multiplying this term with the number of times the GC is invoked gives the final value:

\[ E(\text{MoveCost}) = E(X) \times \text{AvgRecSize} \times N_{GC} \]

The useful write cost is cost involved in writing data to the flash without any overhead of moving or erasing records:

\[ \text{UsefulWriteCost} = \text{totalWrites} \times \text{AvgRecSize} \]

The actual write cost is the cost of erasing and moving records in order to write a new record. Since the erase cost is constant, we ignore it while calculating the actual write cost and therefore, it is just simply adding the expected move cost to useful move cost:

\[ \text{ActualWriteCost} = \text{usefulWriteCost} + E(\text{MoveCost}) \]

Finally, efficiency of a flash device is:

\[ \text{Efficiency} = \frac{\text{usefulWriteCost}}{\text{ActWriteCost}} \]

**Figure 12** shows the theoretical plot against the simulated results (for 500,000 accesses). The theoretical plot is the first line at the top and the 5 lines below that are for different access patterns. \(N\) records were pre-filled and only \(X\%\) of them was accessed. The
theoretical line is close to the simulation results when 100% of the records are accessed. As can be observed, the drop in efficiency of the theoretical model closely mimics that of the simulation results for 100% access up to about 40% fullness. However, the drop is not fast enough to resemble the rest of the simulation results.

But, when we modeled the expectation of active records per block that is going to be cleaned - $E(X)$ – assuming an exponential increase $1 - e^{\lambda x}$, we observed a much closer resemblance to the simulation results as shown in Figure 13. In the exponential equation, $x$ is the fullness of the flash and $\lambda$ is the mean of the expectation of records in the flash. Value of $\lambda$ is found using the expression: $\log(1 - R_b)/x$, where the fullness $x$ is
considered to be 100%. The resulting $\lambda$ is used for calculating $E(X)$ for rest of the fullness levels.

**Figure 13: Theoretical and simulation results for FIFE - uniform distribution (exponential)**

**FIFE Pareto:** The analysis for FIFE – Pareto distribution is the same as FIFE – uniform distribution, except that the expectation of accessing a record is the mean of a random variable for Pareto distribution, which is given by:

$$E(X) = \frac{\alpha X_m}{\alpha - 1} \text{ if } \alpha > 1$$

Therefore, expectation of a miss for Pareto distribution is:
Even for FIFE – Pareto distribution, just like FIFE – uniform, the expected accesses of the flash before cleaning a block is the same as the maximum accesses.

The equation of expectation of active records per block remains the same for Pareto application, as given below. But please do note that the equation for $P_m$ is different as noted above.

$$E(ARB_c) = (P_m)^{A_{max}} \ast R_b$$

The rest of the equations for calculating efficiency remain the same.

**2-Gen uniform:** The analysis for the 2-Gen uniform algorithm is non-trivial as it involves calculating the move cost across two generations. During steady state, the amount of records that can be moved from the first generation to the second depends on the amount of active records in the second generation. Given below is the analysis of the 2-Gen algorithm for the uniform distribution case. The list of symbols used for the 2-generation model is given below.

- **B** – Total blocks in the flash.
- **BlockSize** – Size of a flash block.
- **N** – Count of active records.
- **AvgRecSize** - Average size of a record.
- **$G_1$** - Size of generation 1.
- **$G_2$** - Size of generation 2.
- **$X_1, X_2$** - Total number of active records in generation 1 and 2 respectively.
- Records moved over to generation 2 during GC.
- $\delta_1 = X_1 - \delta_2$ - Records left over in generation 1 after GC.
- Fullness - Ratio of the sum of all active records in flash to size of flash 
  \[
  \frac{N \times \text{AvgRecSize}}{(B \times \text{BlockSize})}
  \]
- $R_{\text{gen1}} = G_1/\text{AvgRecSize}$ – Total records that can be accommodated in gen 1.
- $R_{\text{gen2}} = G_2/\text{AvgRecSize}$ – Total records that can be accommodated in gen 2.
- $E(X_1)$ - Expectation of active records in generation 1.
- $E(X_2)$ – Expectation of active records in generation 2.
- $A_1 = (G_1/\text{AvgRecSize} - \delta_1)$ – Number of accesses possible between successive
  cleans of gen 1.
- $A_2 = (G_2/\text{AvgRecSize} - \delta_2)$ – Number of accesses possible between successive
  cleans of gen 2.
- $N_{\text{GC}}$ - Total number of times GC is invoked.

The 2-Gen GC moves all active records from the first generation to the second. But
the amount of records that can be moved depends on the free space in the second
generation. The expectation of active records in the first generation therefore is the sum
of records present in the first and second generations:

\[
E(X_1) = \delta_1 + \delta_2
\]

The total number of accesses of records in generation 1 between successive cleans ($A_1$)
can also be written as: $R_{\text{gen1}} - \delta_1$. After moving $\delta_2$ records over to generation 2, there
are $\delta_1$ records left over in generation 1. Therefore, the total number of accesses of records
is the difference between the records that can be fitted in generation 1 and the records left over after the move to generation 2.

The probability for an existing record to be accessed is $1/N$ for uniform distribution. Then the probability of not accessing or missing a record, denoted by $P_m$, is:

$$P_m = (N - 1)/N$$

The expectation of active records in generation 1 can be written as below:

$$E(X_1) = (P_m)^{A_1} \ast R_{gen1} - 1$$

Intuitively, the expectation is the total number of times a record will be missed while accessing the entire set of records in generation 1 (1 is subtracted to account for a dummy record).

Similarly, the expectation of records in the second generation is:

$$E(X_2) = (P_m)^{A_2} \ast R_{gen2} - 1$$

Number of accesses between cleans of gen 2 is, $A_2 = A_1 \ast R_{21}$ where the factor $R_{21}$ is the ratio of cleans between gen 1 and 2. This factor depends on the amount of free space in generation 2 as well as on the number of active records in generation 1. When there are more active records in generation 1, then the GC is invoked more often and $R_{21}$ is subsequently low. This in effect reduces the number of accesses between cleans ($A_2$).

$$R_{21} = FS_2 / X_1$$

where, $FS_2$ is the free space remaining in gen 2. It is given by the equation:

$$FS_2 = R_{gen2} - X_2$$
The total number of times GC is invoked decides the total move cost. During one cycle, active records are moved from one generation \((x)\) to the next \((x+1)\), and the next invocation of the GC depends on the amount of free space in generation \(x\). Hence the relation for number of times GC is invoked is given by:

\[
N_{GC} = \frac{totalWrites}{(R_{gen1} - E(X_1))}
\]

where, \(totalWrites\) is the total possible records that can be written to the flash over its life-time. In clearer terms, the equations states that the total number of times GC is invoked is the ratio of the total number of times a record can be written, to the difference between the total records that can be accommodated in generation 1 and the expectation of the active records.

Next important factor in determining the efficiency is to find the expectation of the cost of moving records from one generation to the next and even within the same generation. The expectation of the total move cost is then:

\[
E(MoveCost) = \left((E(X_1) \times AvgRecSize) + (E(X_2) \times AvgRecSize)\right) \times N_{GC}
\]

Useful write cost is the cost involved in writing data to the flash without any overhead of moving or erasing records, which is given by:

\[
UsefulWriteCost = totalWrites \times AvgRecSize
\]

The final term in the efficiency calculation - actual write cost - is the cost of erasing and moving records in order to write a new record. Since the erase cost is constant, we ignore it while calculating the actual write cost and therefore, actual write cost is:
Finally, efficiency of the flash device for 2-generation is:

\[
Efficiency = \frac{usefulWriteCost}{ActualWriteCost}
\]

The above given analytical model can be generalized and extended to the 3-generation and the N-generation models.

**Figure 14** through **Figure 17** compares the theoretical plots for the 1-gen, 2-gen, 3-gen and the N-gen algorithms with their respective simulation results (for 100,000 accesses). As can be observed, the theoretical plot closely resembles the simulation plots.
Figure 14: 1-Generation GC theoretical model at different accesses

Figure 15: 2-Generation GC theoretical model at different accesses
Figure 16: 3-Generation GC theoretical model at different accesses

Figure 17: N-Generation GC theoretical model at different accesses
Chapter 5: Simulation results

This chapter shows the simulation results for all algorithms and for all access patterns. Since efficiency of an algorithm is the most important criteria, we show the graphs that compare efficiency against fullness. For individual algorithms, the read, write and erase costs are shown, while for the combined results (comparing all algorithms) the run time of an algorithm against increasing fullness is shown. All simulation results were for 100,000 cycles, unless indicated otherwise.

Uniform Distribution:

First insert first erase (FIFE): Figure 18 shows a gradual drop in efficiency for the FIFE algorithm as fullness increases. This is because as fullness increases, the amount of inactive records per block decreases and the garbage collection (GC) has to work harder by moving more records and traversing many blocks to find space. Figure 19 depicts the read/write/erase costs and shows that majority of the work is done while erasing blocks. Erasing blocks increases exponentially as fullness increases. This is because the probability of a block being full of active records is more and hence the algorithm has to move more records and subsequently erase more blocks. The dip at the end of Figure 19 and Figure 21 for the erase cost can be ignored as it is due to the simulation getting over when the flash is completely full and the GC is no longer able to find any more space.
Least active clean (LAC): Figure 20 shows the efficiency graph for LAC. The plot shows a similar pattern to that of FIFE. As shown in Figure 21, once again the majority of the work is done when erasing blocks.
Figure 18: Efficiency vs. fullness for FIFE GC for uniform distribution

Figure 19: Read/write/erase costs vs. fullness for FIFE GC

Figure 20: Efficiency vs. fullness for LAC GC for uniform distribution

Figure 21: Read/write/erase costs vs. fullness for LAC GC
3-Generation: Figure 22 and Figure 24 show the efficiency graph for the 3-generation model. Figure 22 is for a block split ratio of 4:3:1 between the three generations whereas Figure 24 is for the 3-gen model with a block split ratio of 12:3:1. The amount of blocks in the first generation (where new records are written) is less than that in FIFE. As can be observed, the drop in efficiency is much faster for three generation GCs when compared with FIFE and LAC algorithms. This is because the 3-gen algorithm has to move the entire set of active records from the first generation to the higher generations as well as erase the first generation whenever it is invoked. When there is no sufficient space in the higher generations, they have to be compressed\(^5\) which further reduces the efficiency.

Comparing Figure 22 and Figure 24, it can be observed that the efficiency when the first generation has 96 blocks is higher than that for 64 blocks. This shows that larger the first generation, the higher is the efficiency.

N-Generation: Figure 25 for the N-Generation algorithm shows a sharp drop in efficiency even at lower fullness levels. This is because only one block is used for writing new records and the GC has to keep moving records to higher generations to make space for new incoming records. Intuitively this algorithm should have high efficiency for data with high locality of reference. We conducted a few experiments on this premise by generating data that has high locality of reference (see Figure 18). Even in such a case, none of the efficiency plots were close to those of FIFE, LAC and 3-generation algorithms. This once again corroborates the fact that lower the space for writing new records (first generation) the lower is the efficiency.

---

\(^5\) Active records are grouped together. This involves erasing the generation before the records are written.
Figure 22: Efficiency vs. fullness for 3-Gen GC (block split count ratio of 4:3:1)

Figure 23: Read/write/erase costs vs. fullness for 3-Gen GC

Figure 24: Efficiency vs. fullness for 3-Gen GC (block split count ratio of 12:3:1)
Figure 25: Efficiency vs. fullness for N-Gen GC

Figure 26: Read/write/erase costs vs. fullness for N-Gen GC
**Pareto Distribution:**

**FIFE:** Figure 27 and Figure 29 are similar to those for the uniform distribution, except that at lower fullness levels, the drop in efficiency is much faster. This is due to the cost involved in moving records that are active during GC. For a uniform distribution, there is a higher chance for every record to be invalidated, whereas in Pareto, fewer records are inactivated, but more often. Pareto distribution has more active data than the uniform distribution and hence contributes more towards the overall move cost.

**LAC:** Figure 29 shows the efficiency graph for the LAC algorithm for 100,000 simulations. At this simulation count, there is a prominent difference between FIFE and LAC. But a more interesting graph is shown in Figure 31 which compares the efficiency plots of FIFE and LAC for 500,000 simulations. At this simulation count, the difference between the two is hardly noticeable. At low simulation counts, the move cost of FIFE is less whereas that of LAC saturates quite early. But when the simulation count is increased, FIFE’s move cost starts to catch up with that of LAC. This is a very important result and shows that FIFE is slightly more efficient than LAC even at very high simulation counts. The simulation count of 500,000 is well above the life-cycle count of a flash device.
Figure 27: Efficiency vs. fullness for FIFE GC

Figure 28: Read/write/erase costs vs. fullness for FIFE GC

Figure 29: Efficiency vs. fullness for LAC GC

Figure 30: Read/write/erase costs vs. fullness for LAC GC
Figure 31: Efficiency vs. fullness (FIFE and LAC) - 500,000 runs
**3-Generation:** Figure 32 shows a similar behavior to that of uniform distribution. These set of graphs are counter-intuitive and therefore have to be analyzed a little deeper. For instance, in 3-Generation GC, when the first generation becomes full and when the GC is invoked, the active records in Gen-1 are moved to Gen-2. This pattern continues and at steady state, Gen-2 has both active and inactive records and Gen-3 also has the same, but has a higher distribution of active records. This means that there is a higher compaction cost involved in the second and third generations. Though more space is created during one cycle of GC, the cost per GC over the life-time of the flash is almost the same for Pareto and for uniform.

**N-Generation:** Figure 34 shows similar behavior to the earlier results for uniform distribution. This result goes against the intuition, as even when there is a clear difference between data access patterns, the efficiency still remains low.

**Eta-N-Generation:** Figure 36 for Eta-N-Generation also does not show any improvement in efficiency. But as will be shown in a later graph, the run time for this algorithm is higher than N-Generation’s at higher fullness levels.
Figure 32: Efficiency vs. fullness for 3-Gen GC

Figure 33: Read/write/erase costs vs. fullness for 3-Gen GC

Figure 34: Efficiency vs. fullness for N-Gen GC

Figure 35: Read/write/erase costs vs. fullness for N-Gen GC
Figure 36: Efficiency vs. fullness for Eta-N-Gen GC

Figure 37: Read/write/erase costs vs. fullness for Eta-N-Gen GC
Efficiency Vs Fullness - Combined Graphs:

Figure 38 through Figure 45 show the combined plots for the various GCs (for 500,000 simulation count). As can be observed, the FIFE and LAC algorithms occupy the top half of the chart and the generational algorithms occupy the bottom half. The 3-Gen algorithm comes in the middle. The reason behind this has been mentioned in the previous results.

Uniform distribution: Figure 39 shows the time taken by the different GC algorithms. As expected, the algorithms with lower efficiency have a higher run time than those with a higher efficiency. GC is invoked much more often in generational algorithms and creates very less space per invocation. FIFE and LAC algorithms on the other hand, create more space per invocation and hence are invoked less often.

Pareto distribution: Figure 42 shows the results for Pareto distribution and the graphs are similar to that of uniform distribution. FIFE and LAC have a clear difference at 100,000 simulations (Figure 40), but as explained before the difference almost is negligible when the simulation count is increased to 500,000. An interesting point is that, FIFE and LAC’s graph for Pareto fall faster at lower fullness levels than that for uniform. At the same time, FIFE’s plot for Pareto falls slower than that for uniform. FIFE for Pareto distribution is better for higher fullness because it chooses the oldest block which happens to have the least number of active records. While on the other hand, LAC also chooses the block with the least active records. If it were to wait a bit more, the hot data would have become cold. Therefore, at lower fullness levels, efficiency depends on the way cold data is handled and at higher fullness levels, efficiency is dependent on the way hot data is handled.
**Bimodal distribution:** Figure 44 for Bimodal distribution also has no surprises and looks the same as the previous graphs for uniform and Pareto.

An important insight is that, when the number of write accesses increases, FIFE and LAC essentially become the same algorithm. FIFE implicitly ages the blocks and chooses the oldest block which happens to have the least amount of records. LAC always chooses greedily chooses the same block and hence the reason behind the both the algorithms mimicking each other when the run count tends to infinity.
Figure 38: Efficiency vs. fullness for uniform distribution

Figure 39: Time taken for GC vs. fullness for uniform distribution

Figure 40: Efficiency vs. fullness for Pareto distribution (100,000 runs)

Figure 41: Time taken for GC vs. fullness for Pareto distribution (100,000 runs)
Figure 42: Efficiency vs. fullness for Pareto distribution (500,000 runs)

Figure 43: Time taken for GC vs. fullness for Pareto distribution (500,000)

Figure 44: Efficiency vs. fullness for bimodal distribution (500,000)

Figure 45: Time taken for GC vs. fullness for bimodal distribution (500,000)
In order to find out if there is a data access pattern that could make generational algorithms much more efficient than FIFE and LAC, it was decided to vary the data access pattern. The flash was pre-filled with all “N” records and then only X% of them was accessed. In other words, data access patterns went from 20% to 100% (in increments of 20%). The idea behind this was to understand how the algorithms perform when (N-X)% of records are never accessed at all. Intuitively, generational algorithms should perform much better than LAC and FIFE as (N-X) % of records are moved to the oldest generations. However, the experimental results turn out to be different than what we expected. The plot for 20% access (lower most line) in Figure 46 for FIFE is higher than the corresponding lines for 1-Gen (Figure 48), 2-Gen (Figure 49), 3-Gen (Figure 50) and N-Gen (Figure 51).

An interesting observation is that between FIFE (Figure 46) and LAC (Figure 47). The lines for the accesses are reversed in both. While LAC has the lowest efficiency for 100% access of records, FIFE has its corresponding lowest efficiency for 20% access of records. This is because LAC being a greedy algorithm always chooses the block which has the least amount of active records. When only 20% of records are accessed, it always chooses between a certain set of blocks for garbage collection. Rest of the blocks is never considered. The amount of data transferred is hence very less and this explains its high efficiency value. For FIFE, since blocks are considered for GC in a round-robin fashion, there are more active records per block to move when only 20% of records are
accessed. Whereas the count of active records to move are reduced when 100% of the records are accessed.

The reason behind the lack of demarcation between data accesses for 1-Gen, 2-Gen and 3-Gen algorithms is as follows. After the records are pre-filled and when the GC is invoked for the first time, the active records are moved to higher generations and the first generation is erased for further writes. However, every time the GC is invoked, the same set of records in the higher generations are moved again and again resulting in the lack of demarcation between different percentages of access.

As Figure 51 indicates, N-Gen GC has a demarcation between 20% and 100% accesses with a higher efficiency for 20% and the lowest for 100% access of records. This is because when only 20% of the records are accessed, the rest 80% of the records which occupy the higher generations are never moved and only 20% of the records are moved. Whereas for 100% access, records in the higher generations are invalidated and hence the GC has to move more records to free up space.
Figure 46: FIFE GC at different data accesses

Figure 47: LAC GC at different data accesses

Figure 48: 1-Gen GC at different data accesses

Figure 49: 2-Gen GC at different data accesses
Figure 50: 3-Gen GC at different data accesses

Figure 51: N-Gen GC at different data accesses
Chapter 6: Conclusion and Future Work

Generational algorithms are typically considered to have better performance for application traffic with a long-tail distribution [7] [10]. However, our experiments indicate that this is not the case and that generational algorithms have poor efficiency irrespective of the data access pattern. This is one of the major results of this study.

We find that the LAC and FIFE algorithms outperform the generational algorithms in most cases and LAC slightly underperforms the FIFE under heavy flash utilization. We also find that even for long-tailed traffic – the canonical use case for generational algorithms – FIFE and LAC perform better than generational algorithms. The reason is that it is non-trivial to configure a generational algorithm to get the optimum performance for a particular traffic pattern. To optimize performance, the ratio of the size of subsequent generations should be the same as ratio between cold data and the rest of the data. Since in most application cases we do not know this a priori, static optimal configuration of generational algorithms is impossible. However an adaptive algorithm which changes allocations between generations on the fly could achieve better efficiency.

A second insight from our study is that at lower levels of utilization it seems that handling of cold data is important. That is, we need to make sure that the cold data is isolated and is not moved. But as the level of utilization starts to increase, the importance
of hot data increases. At higher levels of utilization it is more important to identify hot
data and make sure that it never gets moved, since waiting for little bit time will actually
make them invalid. At lower levels of utilization moving hot data is less probable
because the flash has enough free space that the hot data will expire before there is a need
to clean the block.

The results from our study suggest that for most of applications the FIFE garbage
collection algorithm might suffice.

Future work: In a previous work [17], instrumentation was added to few applications
available in the TOSSIM [18] library to capture access pattern of the variables used in the
applications. The result was stored in the form of a CSV file and was used as input in the
current work. However, the access patterns were identified off-line by looking at the
graphs generated. Some of the future directions for this work are as follows:

- To add instrumentation to some more TOSSIM applications.
- Real-time identification of the access pattern, based on which an appropriate
garbage collection algorithm can be chosen.
- To combine the instrumentation project and current work. This would help in
quantifying the performance of the GC algorithms for some real-world
applications.
Bibliography


http://docs.tinyos.net/tinywiki/index.php/TOSSIM.


http://docs.tinyos.net/tinywiki/index.php/TinyOS_2.x_index_of_contributed_code.