IMPROVING APPLICATION RESPONSE TIMES OF NAND-FLASH BASED SYSTEMS

by

Sai Krishna Mylavarapu

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

December 2008

IMPROVING APPLICATION RESPONSE TIMES OF NAND-FLASH BASED SYSTEMS

by

Sai Krishna Mylavarapu

has been approved

November 2008

Graduate Supervisory Committee:

Aviral Shrivastava, Chair
Karamvir Chatha
Rida Bazzi

ACCEPTED BY THE GRADUATE COLLEGE

ABSTRACT

NAND Flash Memories are becoming ubiquitous with attractive features like low power consumption, compactness and ruggedness. Garbage Collection and Wear Leveling are two operations carried out by Flash Translation Layers (FTLs) that oversee Flash memory management. Both of these operations involve valid data movement and block erasures and are very time consuming, critically affecting application response times. In addition, since FTLs are unaware of dead data corresponding to deleted files at the file system level, the above two operations are carried out on dead data as well, resulting in significant and unnecessary overheads.

This thesis proposes a framework to improve application response times of NAND Flash based systems by enabling FTLs to understand file system level operations as well as interpret application characteristics. Proposed methods also achieve significant improvements in overall Flash management by increasing the longevity of Flash and do not necessitate any changes to existing system architectures. Experimental results presented show that, by interpreting and treating dead data at the FTL level and exploiting idle periods between I/Os in an application to proactively perform small-scale garbage collections in background, the proposed resource-efficient approach can improve application response times by 30% and memory write access times by 34.7%, besides reducing erasures by 29.7% on average.

To My Parents

ACKNOWLEDGEMENTS

I would like to thank Siddharth Chaudhuri for helping me out throughout and acknowledge Dr. Aviral Shrivastava for his guidance and cooperation.

TABLE OF CONTENTS

Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

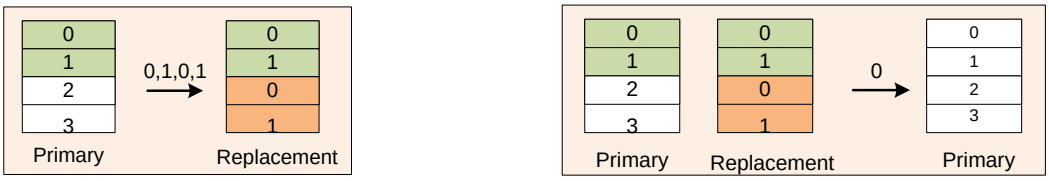
Flash memory is a non-volatile semiconductor memory that can be electrically erased and reprogrammed. With attractive features like low power consumption, compactness and ruggedness it is becoming ubiquitous. USB memory sticks, SD cards, Solid State Disks, MP3 players, Cell phones etc. are some of the well-known applications of the Flash memory technology.

1.1. NAND Flash Architecture

Flash is organized into blocks and pages. A block is a collection of 32 pages each of 512 bytes. Each page has a 16 byte out-of-band (OOB) area used for storing metadata. In addition to read and write, Flash also has erasure operation. Owing to the "Erase-before-rewrite" characteristic, a re-write to a page is possible only after the erasure of the complete block it belongs to, which is an extremely time consuming process. Another limitation of Flash is endurance: it can only withstand finite number of erasures, typically 100,000. To hide above characteristics from applications, a dedicated Flash management module called Flash Translation Layer (FTL) [7] may be employed. The primary responsibilities of an FTL are allocation, i.e., logical to physical address translation and cleaning, i.e., reclamation of invalid data.

1.2. NAND Flash Management

Available blocks in Flash are organized as Primary and Replacement blocks [6]. When a page rewrite request arrives, a primary block is assigned a replacement block. When the replacement block itself is full, and another rewrite is issued, a fold or merge operation needs to be performed, as depicted in Fig. 1. Valid data in old two blocks is consolidated



A. Overwrites to 0,1 pages redirected to replacement block

B. Further overwrite to page 0 triggering a Merge operation Initial primary and replacement blocks are freed.

Fig.1. Fold or Merge operation

and written to a new primary block and the former are freed subsequently. Also, after a series of rewrites, free space in the device falls below a critical limit and needs to be regenerated by garbage collecting or reclaiming the invalid data. At the end of this GC process, valid data is consolidated into primary blocks. Thus, a GC is a series of forced fold operations. On the other hand, WL process involves frequent data shuffling between highly erased and least erased blocks to achieve uniform wear. Thus, both GC and WL operations involve expensive erasures and data copying, and so are very time and energy intensive. Since Flash is not available while carrying out the above operations, applications can be potentially stalled, resulting in very poor response time characteristics. Especially, it was shown that a GC may take as long as 40sec [12].

1.2.1. Garbage Collection and Wear Leveling Overheads

In order to understand impact of GC and WL operations on application response times, a digital camera workload was ran on a 64MB Lexar Flash drive formatted as FAT32 [17] and fed resulting traces to Toshiba NAND Flash [21] simulator and measured WL and GC overheads. During the scenario a few media files of sizes varying between 2KB and 32MB were created and deleted. Application response times at various instances are plotted in Fig.2. The peak delays at the right extreme of the figure, which critically affect

application response times, correspond to instances where a GC is being carried out. Table I lists overheads due to wear leveling data belonging to deleted files (dead data), in terms of percentage increase in device delay, erasures, average memory write access time and folds.

Previous efforts [2] [8] [9] [11] [15] [16] [18] [22] [23] have focused on improving GC and WL efficiency to improve application response times. However, they have not directly attempted to minimize, or eliminate automatically triggered, unpredictable and lengthy GC delays. Another limitation of previous works can be seen at the file system level. Since file systems only mark deleted files at the time of deletion, but not actually erase corresponding dead data in Flash, FTL treats dead data as valid until specifically overwritten by new file data and carries out *useless data migration* on the same during GC and WL operations, resulting in unnecessary and significant Flash delays. Thus, various file systems are shown to be lengthy in response times in the presence of dead data [5]. Previous efforts that addressed this issue required significant changes to existing file system architecture [5].

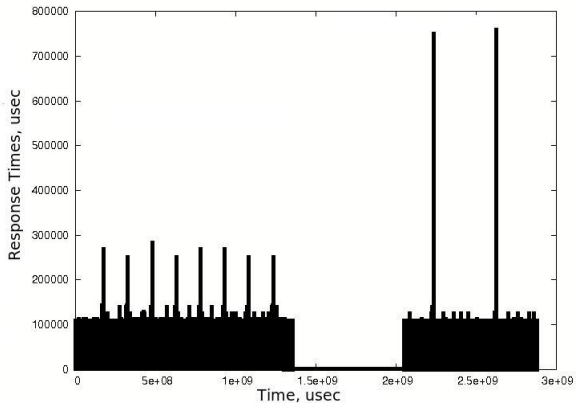


Fig.2: NAND Flash device delays

Table I: Effect of dead data on various metrics

Metric	**% increase due to dead data**
Device Delays	12
Erasures	11
W-AMAT	12
Folds	14

This thesis work proposes an FTL-level framework to improve application response times that does not require any interface changes to existing systems. This is achieved by enabling FTL to interpret file system operations and also to understand application characteristics. Proposed file system level method, FSAF: File System Aware FTL that enables FTL to recognize file deletion dynamically and resource-efficiently and handle dead data proactively to significantly reduce GC and WL overheads. This is achieved by tracking changes to the file system data structure in Flash, without necessitating any changes to existing file systems. The SLAC: Application SLack time Aware Garbage Collection scheme proposed understands application timing characteristics dynamically and exploits idle times between Flash requests to proactively perform fold operations in order to reduce, or even completely eliminate GCs. Also presented along with SLAC is an algorithm to perform these proactive fold operations at high efficiency (by minimizing the cost of folding) to further improve device delays.

Experimental results demonstrate the proposed approach can improve application response times by 30% and memory write access times by 34.7%, besides reducing erasures by 29.7% on average. Individual results for SLAC and FSAF are also presented: SLAC reduces GCs by 80%, besides lowering average write access times and device latencies by 20% and the number of erasures by 8%. FSAF improves application response times and average write access times by 22% on an average, besides reducing erasures by 21.6%. It is also shown that both of the above comprehensive approaches can be implemented in a resource-efficient manner, and do not necessitate any interface changes.

CHAPTER 2

RELATED WORK

Several works to improve application response times have been proposed so far, by attempting to improve the efficiency of GC and WL operations. Also, file system level work has been done to handle dead data. It has to be noted that even though there exists a lot of work related to GC in programming languages, it cannot be carried over to Flash, owing to Flash characteristics. Also, a real-time GC policy for Flash was considered by Chang et al. [12]. This method, however, is specific for real-time systems.

2.1. Previous Works on Garbage Collection

Works on GC proposed so far for general purpose systems attempt to improve application response times by increasing the efficiency of the GC process, as follows. The greedy GC approach was investigated by Wu et al. [22]. To reduce GC costs, this method reclaims blocks with high dead page counts. This approach performs poorly with high locality of reference workloads, as it doesn't consider hot-cold data segregation. Kawaguchi et al. [11] came up with the cost-benefit policy, by considering both utilization and age of blocks. They also introduced block-level segregation of hot and cold data (frequently updated data is termed hot). Thus, the approach performed well for high locality workloads. Cost Age Time (CAT) policy [15] was considered by Chiang et al. that also focuses on reducing the wear on the device (increase endurance) apart from addressing segregation. The method uses a data redistribution method that works at a fine-grained page-level for efficient hot-cold data separation. Kim et al. [9] proposed a cleaning cost policy, which focuses on lowering costs and evenly utilizing Flash blocks.

Wear-leveling is achieved by dynamically separating cold data and hot data and periodically moving valid data among blocks. However, it has to be noted that all these approaches are device-centric: they do not take the application timing characteristics into consideration. In other words, GC may be triggered by FTL at a critical instance when an immediate response from Flash is expected.

A swap-aware GC policy [18] was introduced by Kwon et al. In order to minimize the GC time and extend the lifetime of the Flash based swap system, they implemented a new Greedy-based policy by considering different swapped out time of the pages. However, this approach necessitates a change in the existing system architectures, and is specific to few systems.

2.2. Previous Works on Wear Leveling

Various approaches based on dynamic wear leveling have been proposed [2] [13] that achieves wear leveling by trying to recycle blocks with small erase counts. In such approaches, an efficient way to identify hot data (frequently updated data) becomes important, and excellent designs were proposed, e.g., [8] [10] [13] [16]. Although dynamic wear leveling does have great improvement on wear leveling, the endurance improvement is stringently constrained by its nature: That is, blocks of cold data are likely to stay intact, regardless of how updates of non-cold data wear out other blocks. In other words, updates and recycling of blocks/pages will only happen to blocks that are free or occupied by non-cold data, where cold data are infrequently updated data. Static wear leveling is orthogonal to dynamic wear leveling. Its objective is to prevent any cold data from staying at any block for a long period of time so that wear leveling could be

evenly applied to all blocks. Static wear leveling approaches were also pursued [23] to treat both non-cold and cold data blocks. These approaches are all again device-centric, and do not consider application characteristics while triggering a WL operation.

2.3. Existing File System Level Works

To handle dead data, Kim et al. [5] proposed a new file system, MNFS, to achieve uniform write response times by carrying out block erasures immediately after file deletions. This method necessitates changes to existing system architectures.

Thus, efforts so far to improve application response times did not take application characteristics into account, or necessitate significant system interface changes.

CHAPTER 3

OPPORTUNITIES TO IMPROVE APPLICATION RESPONSE TIMES

To explore opportunities to avoid GC delays, idle times (slacks) between application requests of several benchmarks were analyzed. Fig.3 plots slacks at each I/O request, computed as the time between two subsequent requests of CellPhone benchmark. The dark horizontal line represents the time needed for an individual fold operation. One important observation we make from this graph is that several I/O requests have slack times that will allow fold operations to be performed in background, without any increase in the device latency. In other words, there may be significant opportunity to carry out regular GC operation in background without affecting application response times.

Fig. 4 depicts the file deletion operation in FAT32 file system. When a secondary storage like Flash is formatted, FAT32 allocates first few sectors to FAT32 table to serve as pointers to actual data sectors. When a file is created or modified, the table is updated to keep track of allocated /freed sectors of the file. However, when a file is deleted or shrunk, the actual data is not erased (this process is termed implicit file deletion). In over-writable media like hard disks, this poses no problem, as the new file data is simply overwritten over dead data. However, because Flash doesn't allow in-place updates, dead data resides inside Flash until a costly fold or GC operation is triggered to regain free space. Whereas, FTL carries out expensive WL operation regularly on dead data blocks. Thus, dead data results in significant GC and WL overhead, affecting application response times. If FTL can detect dead data dynamically upon file system operations,

we will be able to save on related dead data migration costs.

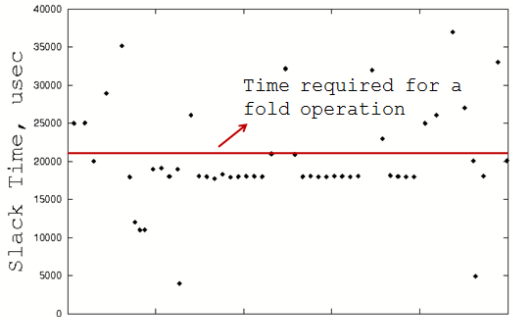


Fig.3: Slack time available for CellPhone benchmark

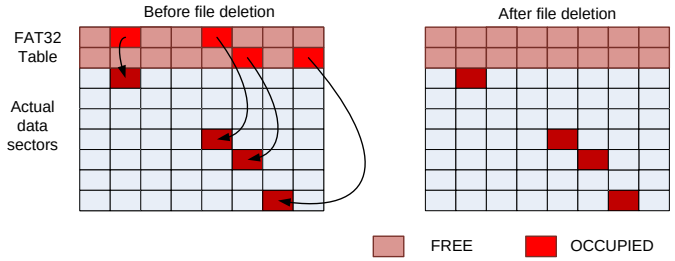


Fig.4: File deletion in FAT32 File System

CHAPTER 4

THE APPROACH

Two important intuitions are:

1. If GC is scheduled in the background, we will be able to achieve reclamation goals without affecting application response times

2. If we enable FTL to interpret file system operations, we can improve application response times without necessitating any changes to existing file systems

To this end, two different solutions to improve application response times are proposed that are compatible with existing systems. The first method, SLAC: Application SLack time Aware Garbage Collection scheme is an application-driven GC framework to carry out fine-grained GCs to improve application response times, which also performs Selective Folding to increase overall GC efficiency, to further improve response times. The second method, FSAF: File System Aware FTL is an FTL-based solution to efficiently recognize and also handle dead data dynamically. Even though results of FSAF are demonstrated on FAT32 file system, the method is equally applicable to all other file systems that perform implicit file deletions.

4.1. SLAC: Application Slack Time based Garbage Collection

At every request to Flash, SLAC predicts idle time until next request (termed slack) to determine how many folds can be done within the slack and selects blocks for folding

that have least cleaning costs to maximize garbage collection efficiency. SLAC framework is integrated with the FTL that interfaces the application.

4.1.1. Slack Prediction

SLAC has a history-based prediction mechanism to arrive at an estimate of next slack, as shown in Fig 5. The goal of slack prediction is not only to arrive at a good estimate of slack, but also achieve this at a minimal overhead. Both slack average D_{avg} and fluctuation D_{dev} are taken into consideration while performing slack prediction. As read and write requests arrive, SLAC maintains a list of last n slacks. At every request, running-average slack, D_{avg} and a measure of deviation D_{dev} are derived from their old values and the latest time stamp. If the deviation D_{dev} is within a small threshold ε, D_{avg} is projected to be the next slack. If the fluctuation is beyond ε, the last slack S_n is projected as the next slack. The next step is to calculate how many fold operations possible during this slack, F_m, which is calculated as:

$$F_m = S/t_f$$

Where, S = predicted slack and t_f = time for each fold operation.

***slack_prediction*:**

1. Compute the n^{th} slack from time stamps: $s_n = (t_n - t_{n-1})$
2. Derive new $Davg = DavgOld - (|s_1 - s_n| * \frac{1}{n})$
3. Compute $D_{dev} = \sum_{i=1}^{n} |s_i - D_{avg}| * \frac{1}{n}$
4. If $s_n < t_f$: return ZERO
5. If $D_{dev} < \varepsilon$ and $s_n > t_f$: return $Davg$
6. If $D_{dev} > \varepsilon$ and $s_n > t_f$: return s_n

Fig.5: Slack Prediction.

In turn, time for each fold operation, t_f is calculated as:

$$t_f = 2 * n_p * t_{ro} + n_p * t_{rp} + n_p * (t_{wp} + t_{wo}) + 2 * t_e$$

Where,

n_p = pages per block, t_{ro} = OOB read time, t_{wo} = OOB read time, t_{rp} = page read time, t_{wp} = page write time, t_e = block erase time.

These parameters are taken from the datasheet supplied by the Flash manufacturer. It has to be noted that t_f represents worst case fold time. Depending upon the amount of valid data, actual fold time varies.

The two parameters to tune in this algorithm are n and ε. Bursty nature of I/O requests of various benchmarks and general purpose workloads suggests that prediction should be based upon recent rather than old slack data. This suggests that small values of n work well. The accuracy of this history-based predictor was explored over a wide range of n and ε. The experimental results show that best prediction happens at $n = 4$ and $\varepsilon = 5000\ \mu sec$.

Even though SLAC can be equipped with other slack prediction mechanisms as in [4] care needs to be taken about their resource overheads. As shown in the experimental results, the algorithm in Fig. 5 achieves good prediction at a minimal resource overhead.

An important aspect of SLAC is that it automatically switches off issuing selective folds at higher request-arrival rates. In such cases, underlying FTL automatically triggers garbage collection when the free space in the device falls below a critical limit. This makes sure that SLAC never worsens application response times.

4.1.2. Selective Folding

Once F_m, the possible number of folds is determined, selective folding algorithm chooses blocks to fold from available blocks. In order to improve overall garbage collection efficiency, we have to identify blocks with minimal cleaning costs. Costs associated with a fold operation are because of erasures and valid data copying. In a fold operation, two erasures are a necessity, and so we need to concentrate on reducing the valid data copying cost. This is minimized when the dead page count in a block is maximum, i.e., when minimum valid data needs to be copied. In addition, SLAC makes sure that the selected blocks are also hot, i.e. updated frequently. Thus, folds that will anyway be triggered in the near future are performed in the slack, eliminating any need for application stalls.

The selective folding algorithm is presented in Fig 6. After F_d is determined, i.e. blocks that give maximum garbage collection efficiency are known, we need to see if all these folds can be carried out within the predicted slack. If allowed number of folds F_m exceeds F_d, all the blocks in L_h are folded. On the other hand, if there are more foldable blocks than are allowed ($F_m < F_d$), SLAC sorts the list of hot blocks L_h and returns the list of F_m blocks from the sorted list.

One threshold to tune here is the dead page count d_{Th}. Higher values of d_{Th} allow for very efficient garbage collection but lesser number of folds at a slack, whereas lower values mean vice versa. Since higher garbage collection efficiency means lesser number of block erasures and block copying overhead, it is intuitive to go for a higher value of

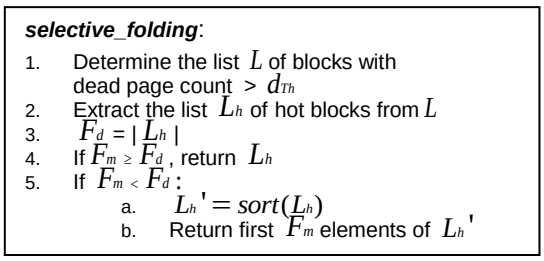


Fig. 6: Selective Folding.

d_{Th}. The experimental results also confirm this intuition, as shown in experimental section. By setting d_{Th} to 32 dead pages, SLAC executes only at highest efficiency achievable by the FTL's garbage collection policy. On the other hand, sorting needed in step 5a of Fig.5 is eliminated, because all blocks contain equal number (32) of dead pages. Also, unlike the other garbage collection policies, SLAC employs scanning rather than sorting of blocks based on dead page count, reducing the algorithmic overhead considerably. Whereas other garbage collection policies sort the whole block list every time for performing garbage collection, SLAC employs scanning, and might only have to sort a relatively small list L_h of blocks if $F_m < F_d$.

4.2. FSAF: File System Aware FTL

FSAF monitors write requests to FAT32 table to interpret any deleted data dynamically, subsequently optimizing GC and WL algorithms accordingly. Also, depending upon the size of dead content and the Flash utilization, proactive dead data reclamation is carried out.

4.2.1. Dead Data Detection

Dead data detection is carried out by FSAF dynamically as files are deleted by the application. Since the file system does not share any information with the FTL regarding

file management, the only way we can interpret file system information at FTL is by understanding the formatting of Flash and keep track of changes to the file system data structure residing on Flash. The goal of dead data detection is to carry out this process efficiently without affecting performance.

The format of Flash can be understood by reading the first sector on Flash, called Master Boot Record (MBR) and the first sector in the file system called FAT32 Volume ID. The LBA_Begin field of the MBR reveals the location of the FAT32 Volume ID sector. Subsequently, the location of the FAT32 table can be determined as follows:

$$FAT32\_Begin\_Sector = LBA\_Begin + BPB\_RsvdSecCnt$$

The size of the FAT32 table is given by the field $BPB\_FATSz32$. Both $BPB\_RsvdSecCnt$ and $BPB\_FATSz32$ are read from the FAT32 Volume ID sector.

Once the size and location of the FAT32 table are determined, dead sectors can be recognized by monitoring writes to the table. FAT32 stores the pointer to each data sector allocated to a particular file in corresponding locations in the FAT32 table. To delete a particular file, all the pointers to data sectors are freed up by zeroing out their content. In other words, dead sectors resulting from shrinking or deleting a file can be found out by reading corresponding pointers prior to their zeroing out. If all the sectors in a block are dead, the whole block is marked as dead. Thus, FSAF needs to maintain a buffer for reading FAT32 table sector before it is zeroed out by the file system. Fig. 7 depicts the algorithm.

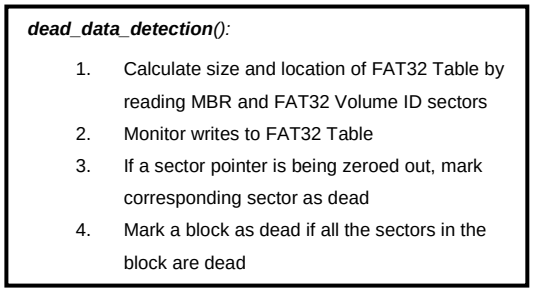


Fig. 7 Dead data detection algorithm.

4.2.2. Avoidance of Dead Data Migration

Once dead sectors are recognized, GC and WL algorithms are instructed to avoid copying their content during regular operation of Flash. Thus, dead data migration is avoided during valid data copy occurring while carrying out GC and WL operations.

4.2.3. Proactive Reclamation

When larger files or files occupying contiguous sectors are deleted, dead data occupies complete blocks. Since these blocks do not contain any valid data, they can be reclaimed without any copying costs, unlike blocks that require valid data copy during normal folding operation. Thus, reclaiming such blocks is inherently a highly efficient operation in comparison to a forced fold operation during a GC. Thus, when the free space in Flash falls below a critical threshold, instead of proceeding with costly GC operation, dead blocks can be reclaimed to delay or avoid GC by regenerating free space dynamically.

However, application response times still might suffer when all the dead data is reclaimed together, owing to costly erasure operations. In order to avoid this, proactive reclamation of dead blocks is taken up. FTL triggers GC higher Flash utilizations [11], i.e., when the free space in the device is below a critical limit, and continues folding until free space

reaches another threshold. In other words, to avoid delays due to GC, free space in the device should be kept above the GC threshold. So, dead block reclamation should be scheduled when Flash utilization is reasonably high, but not high enough to trigger a GC operation. On the other hand, number of dead blocks proactively reclaimed must be as small as possible, as expensive erasure operations can impact application response times. Yet another important factor to be taken into consideration is the amount of dead data in Flash - this decides whether or not proactive reclamations need to be run.

The proactive reclamation algorithm is as presented in Fig. 8. We first check whether the dead content is greater than a threshold δ. If not, GC and WL are informed to avoid useless dead data migration by marking dead sectors. If dead content is greater than δ, we check whether system utilization is higher than μ, i.e. whether at least μ percentage of blocks is already used. In such a case, we proceed to reclaim dead blocks proactively apart from avoiding dead data migration. Thus, dead block reclamation proceeds until number of dead blocks reach another threshold Δ.

Even though proactive reclamation improves application response times by avoiding or delaying costly GC operation, it should be scheduled in such a way that application stall time is minimal (since proactive reclamation is a series of erase operations, it can be time consuming). In other words, parameters δ, μ and Δ should be carefully configured such that reclamation is highly efficient. Large values for δ and μ avoid frequent reclamation, but might impose a lot of reclamation activity. Small values for Δ mean smaller reclamation activity, but frequent triggers for reclamation. To arrive at reasonable values for these parameters, the effect of varying these parameters on various performance

metrics was explored, as presented in the next section. The results confirm the intuition at best performance is achieved at high values of δ and μ low values of Δ.

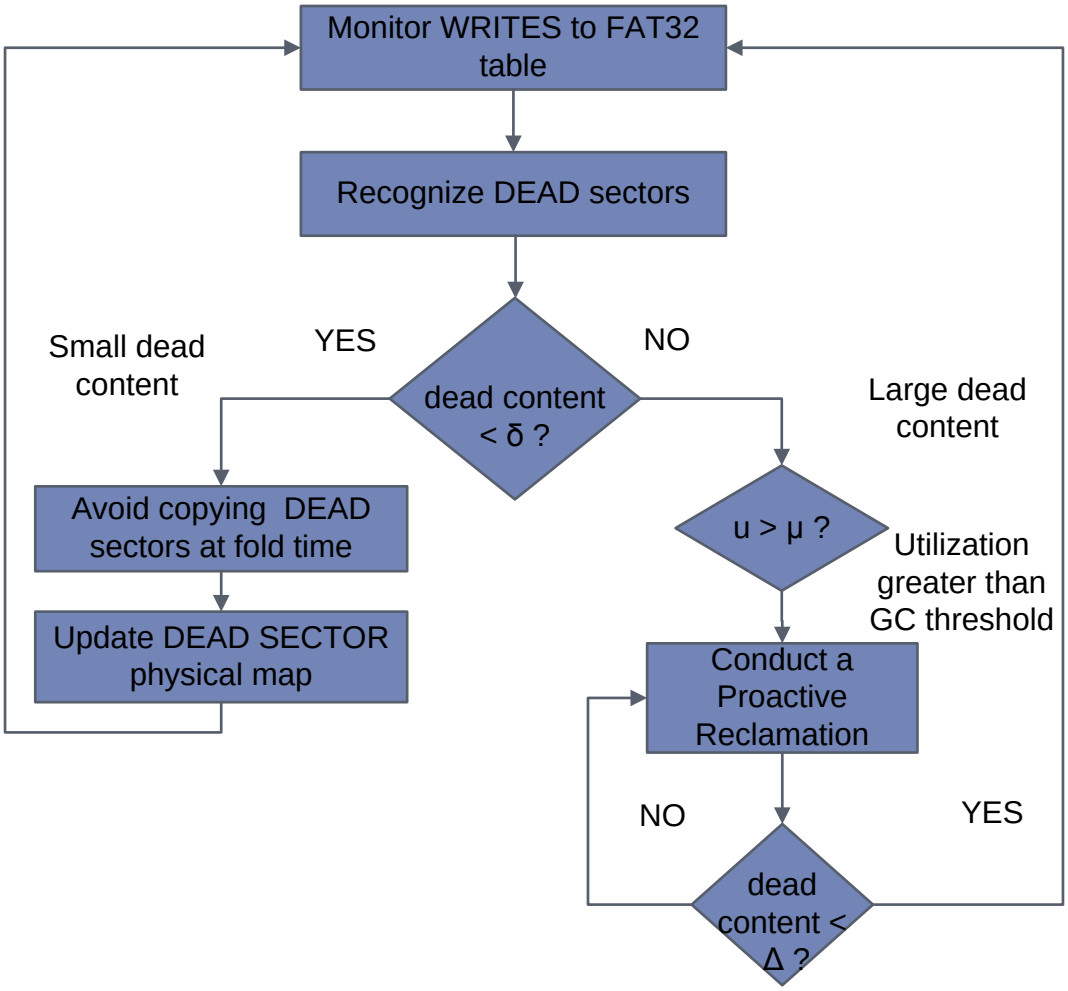


Fig. 8: Proactive Reclamation

CHAPTER 5

EXPERIMENTAL SETUP

Trace-driven approach was used for the experimentation. Several benchmarks with different slack-time characteristics and localities of reference were used for trace collection on a PC running on Linux 2.6.18. Benchmarks were run from a FAT-32 formatted Flash USB stick connected to the PC and usbmon utility [25] was used to extract the timing, sector and request type information from USB traffic, forming the application trace. The collected traces were fed to a simulated Toshiba NAND Flash [21]. Log-based NFTL [9] was realized on top of it and implemented Greedy [22] and Cost-benefit [11] GC policies. CAT [15] policy was not considered, as the original paper was proposed for page-mapped implementation, which is not viable for Flash sizes of today. SLAC was finally integrated with the setup.

Various benchmarks were used to evaluate the approach. MP3 and MPEG benchmarks were obtained by running different media files and issuing writes to Flash simultaneously. Other benchmarks, JPEG, and MAD were taken from MiBench. Also, other file system benchmarks simulating Event Recorder, Fax and Cell Phone were also run on the experimental setup. For evaluating FSAF, benchmarks those benchmarks that represent most frequently encountered file system scenarios on removable Flash storage media such as SD cards in applications like digital cameras, mp3 players, digital camcorders and memory sticks:

s1: Huge sized file creation and deletion

s2: Medium sized file creation and deletion

s3: Small sized file creation and deletion

In order to simulate real-world scenarios, Flash was brought to 80% utilization and the size of Flash for each benchmark was set to 64 MB. FTL was configured to start GC when the number of free blocks falls below 10% of total number of blocks and stop GC as soon as percent free blocks reaches 20% of total number of blocks. WL is triggered whenever the difference between maximum and minimum erase counts of blocks exceeds 15. The size of files used in various scenarios was varied between 32MB to 2KB.

CHAPTER 6

EXPERIMENTAL RESULTS

6.1. Configuring SLAC Parameters

The parameters of n (number of samples) and ε (fluctuation threshold) need to be configured to run SLAC. Slack prediction algorithm was ran with various values n of for all the benchmarks, and found that the intuition of smaller values of n performing better than larger values. The reason behind this can be explained as follows. Since future samples tend to be influenced more by recent past, we can follow the application patterns closely with n as small as possible, but big enough to accommodate fluctuations in the recent sample data.

After performing various experiments, we fixed n at 4 and ε at 5000 μsec. Other prediction approaches were evaluated like weighted moving average, with recent samples assigned higher weights than the older. Certainly more sophisticated approaches as in [4] can be taken up, which will enhance slack prediction, but come with a higher algorithmic and resource overheads.

To determine the best value for the dead count threshold, d_{Th} the variation in the number of erasures, folds and FTL-triggered garbage collections against various threshold values was plotted, for SLAC implementation on Greedy and Cost-benefit policies. Fig. 9 depicts the sample experimental results for the CellPhone benchmark. It was observed

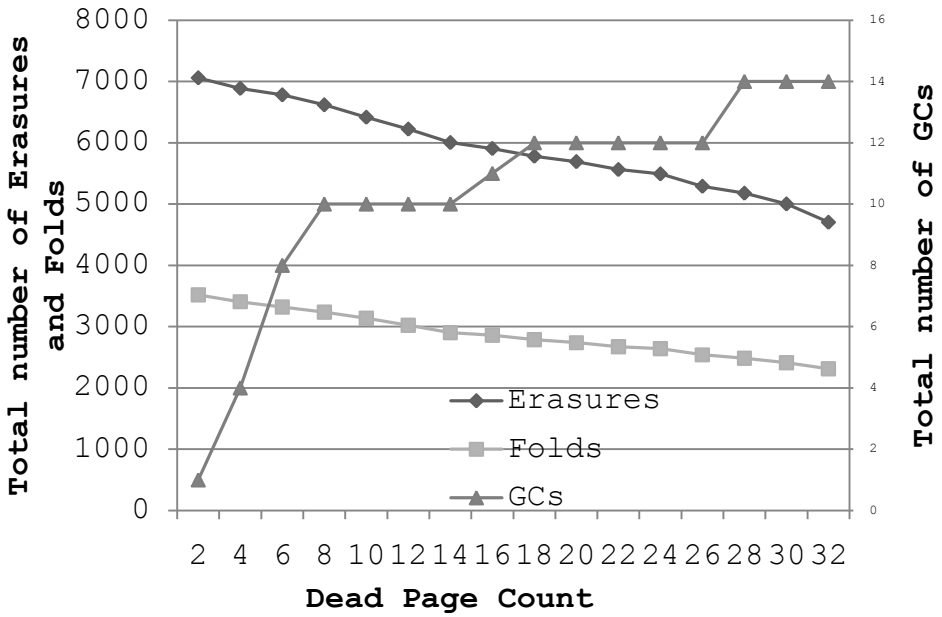


Fig.9. Variation in Erasures, Folds and garbage collections against dead page count

when d_{Th} was increased from 2 to 32, erasures and folds drop significantly. However, since higher threshold allows lesser folds during slack, we see an increase in the number of FTL-triggered garbage collections with higher d_{Th}. This concurs with the hypothesis of increasing garbage collection efficiency with the increasing values of d_{Th}. Thus, it was set to 32, i.e. hot blocks only with dead page count equal to 32 are considered by SLAC for folding. From the specification [21], fold time t_f was calculated to be 20128 usec.

6.2. Configuring FSAF Parameters

The parameters δ, μ and Δ need to be configured to run FSAF. Proactive reclamation algorithm was ran with various values of δ and μ for all the benchmarks, and results supported the intuition that higher values for these parameters result in higher performance. By setting these to high as possible, proactive reclamation is triggered only when the system is low in free space, but runs frequently enough to generate sufficient

free space. Thus, δ was set to 0.2 and μ to 0.85, i.e. when the dead data size exceeds 20% of the total space and system utilization is 85%, proactive reclamation is triggered.

To determine the best value for Δ, it was observed variation in the total application response times, number of erasures, and garbage collections against various sizes of reclaimed dead data, represented by $\delta` ( = (\delta - \Delta))$. Owing to lack of space, related results were omitted. It was observed that when $\delta`$ was increased from 0 to 0.18, Flash delays and erasures decrease initially and increase afterwards, as the reclamation activity increases. However, number of garbage collections remains the same. Thus, $\delta`$ needs to be set to a small positive value. This concurs with the hypothesis that small values for $\delta`$ are better than large values. So, Δ was set to 0.18.

In essence, FSAF is configured to proactively reclaim dead data as soon as dead content becomes more than 20% of the total Flash size when Flash utilization is greater than 85%, and reclaims 2% of dead blocks at each invocation.

6.3. Improvement in Application Response Times

Fig. 10 shows application response times of different benchmarks for Greedy as well as the combined approach (COMBO) that includes both FSAF and SLAC techniques. Depending upon the timing characteristics as well as the deleted file content, we see that different benchmarks differ in their total application response times. Fig. 11 shows average page write access times for each benchmark, for Greedy and COMBO approaches. By detecting dead data dynamically and scheduling GCs in the background, we see that COMBO achieves significant reduction in both application response times

and page write access times.

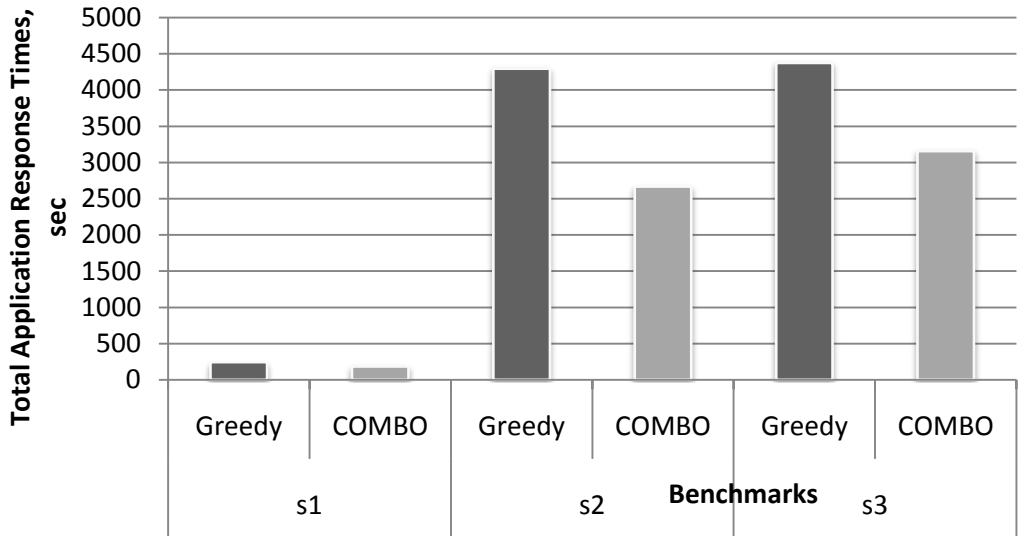


Fig.10: Total application response times for various benchmarks – Greedy vs. COMBO

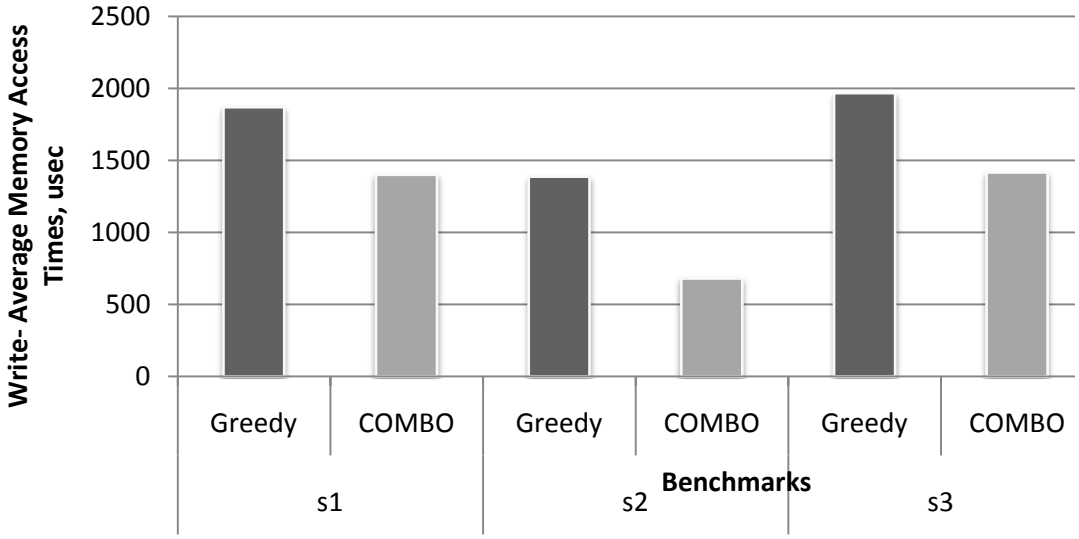


Fig.11: Average memory write-access times for various benchmarks for Greedy and COMBO

In order to understand the above composite gains in detail, presented are individual results and discussion. Fig.12 depicts the average page write access times for each benchmark. Each of the histogram for a benchmark represents Greedy, SLAC-Greedy, Cost-benefit and SLAC-Cost-benefit policies respectively. An average of benchmarks for each of these methods is also presented. As per the specification [21], a page write is supposed to take 400 usec, but we see that, in reality the values are much higher. The reason behind this behavior is the delays associated with garbage collection operations. It has to be noted that the effect of garbage collection overhead becomes even more pronounced at higher Flash utilizations. One can see from Fig.12 that SLAC implementations of Greedy and Cost-benefit policies show improvement over the normal Greedy and Cost-benefit garbage collection policies. This is because, by carrying out garbage collection activity in the background and also selectively folding, SLAC policy significantly decreases write access times compared to Greedy and Cost-benefit implementations. From Fig. 12, we can also observe variation in the average write access time across benchmarks. This is majorly because of two reasons: variation in the locality of reference and difference in the slack times available to each benchmark. We observe that maximum gains can be obtained when a benchmark exhibits sequential write access patterns, also with a reasonable slack. MP3 benchmark, thus gains maximum by as much as 51% with SLAC. On the contrary, gains are less when there is no slack. However, we see that SLAC achieves 18.2% improvement on the average. It is important to know that additional writes and reads to pages and OOBs are issued by the FTL itself during the process of folding while copying valid pages. In other words, application-issued writes

trigger FTL to issue more writes during the process of folding. Since SLAC always folds blocks with minimum cleaning costs, the above delays are also reduced automatically, contributing to the reduction of Flash access times.

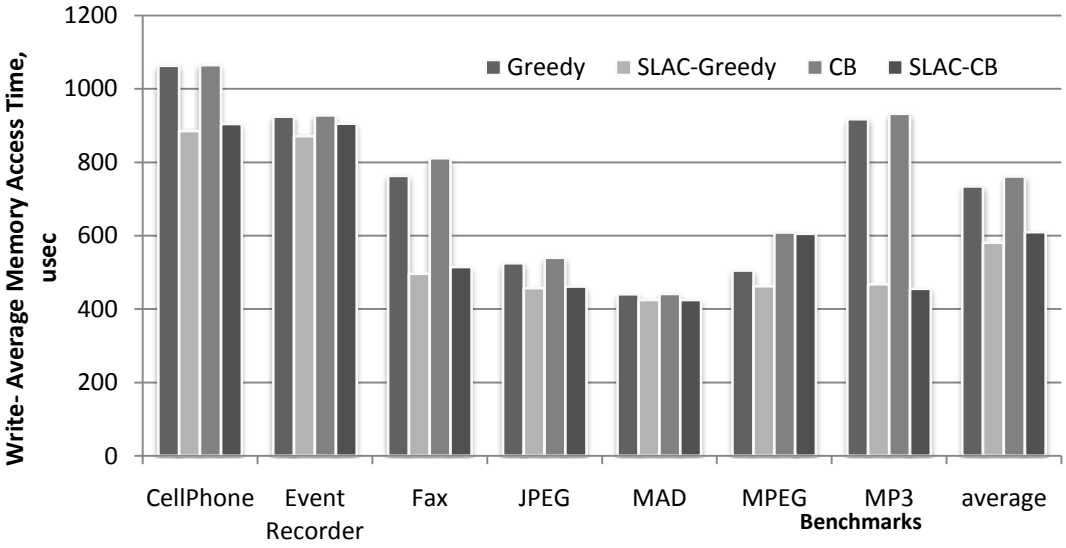


Fig.12. Average page-write access times with various garbage collection policies

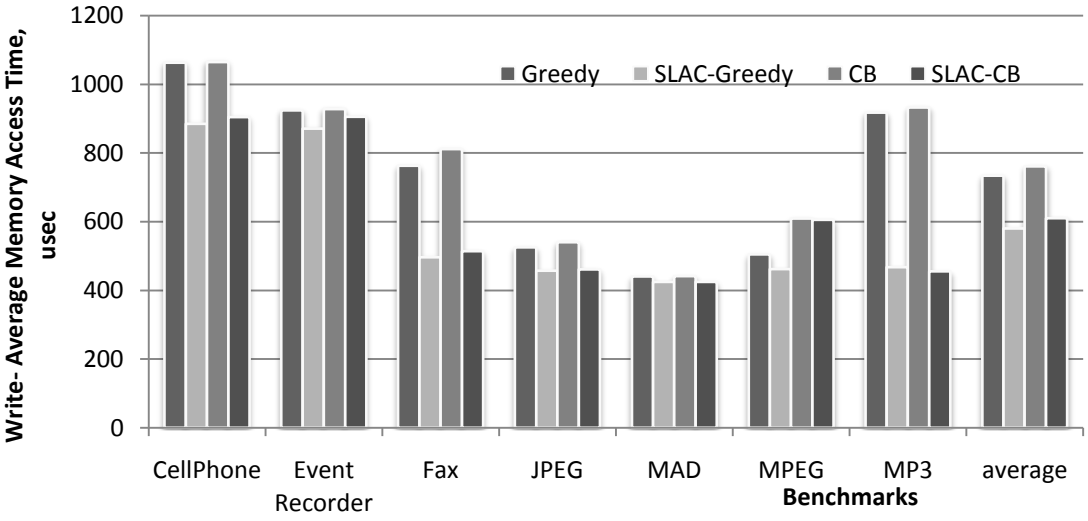


Fig.13. Normalized total device delays with various garbage collection policies.

Total device delays for each benchmark and also the benchmark average are given in Fig.13, after normalizing to Greedy method. The total device delay includes delays incurred due to reads, writes as well as garbage collections. We can observe that improvements in device delays after employing SLAC are similar to improvements in write access times. This is because of the fact that read access times of Flash are much lower than write access times, and also because reads are normally cached. Thus, SLAC aims at reducing write access times.

FSAF results are presented here. Fig.14 depicts total application response times for each of the benchmark for both greedy and FSAF approaches. We observe that the FSAF approach improves response times by 22% on the average, and 32% for the scenario s2 compared to the greedy approach. From Fig. 14, we can observe that there is a variation in the total response times for different scenarios, owing to the content and distribution. We observe that maximum gains can be obtained when dead data occupies contiguous rather than randomly distributed sectors, as in the scenario s2. However, we see that FSAF achieves 22% improvement on the average.

It has to be noted that the total device delay includes delays incurred due to reads, writes issued by the application as well as those issued during carrying out garbage collection and wear leveling activity. When file system issues reads and writes and folding and wear leveling are triggered, additional writes and reads to pages and OOBs are issued by the FTL during the process of valid data copying. In other words, total writes carried out are more than application-issued writes. Since FSAF always avoids dead data migration and directly reclaims dead blocks, device delays are reduced, contributing to the reduction of

Flash access times and hence application response times.

Fig. 15 depicts average memory write access times (W-AMAT) for different scenarios for both greedy and FSAF approaches. We can observe that improvements in W-AMAT after employing FSAF are similar to improvements in response times. This is because of the fact that read access times of Flash are much lower than write access times, and also because reads are normally cached. The variation in the average write access time across benchmarks is owing to dead data content.

It has to be noted that the response times suffer majorly at higher Flash utilizations when garbage collection operations are triggered out to regenerate free space. So, if enough free space can be generated at higher utilizations, we can delay or even avoid costly garbage collections. FSAF achieves the same by dead data reclamation at higher utilizations. On the other hand, wear leveling overhead because of dead data, which is incurred at all Flash utilizations, is avoided by FSAF by avoiding *dead data migration*.

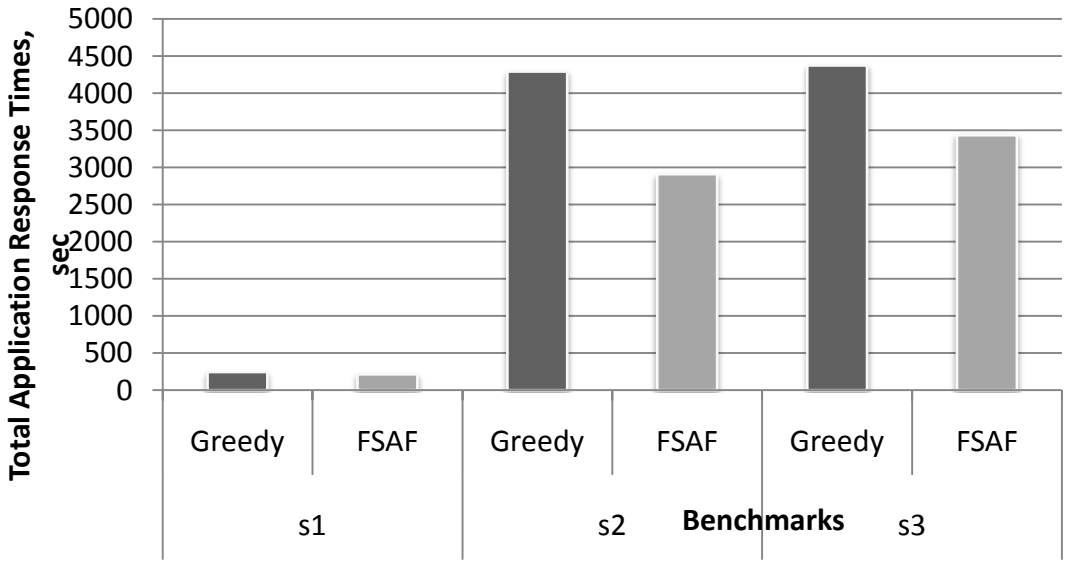


Fig.14: Total application response times for various benchmarks

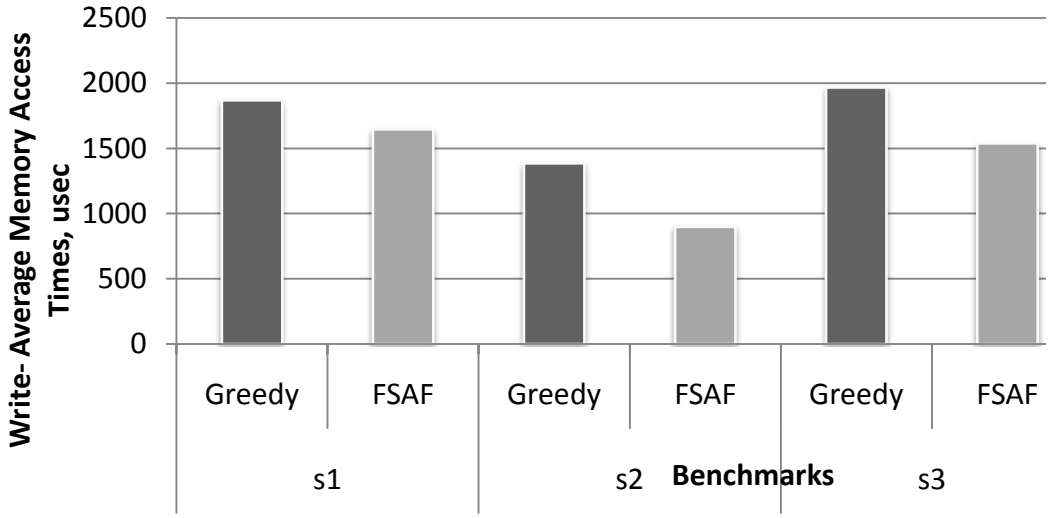


Fig.15: Average memory write-access times for various benchmarks

6.4. Improvement in GCs and Erasures

Table II presents the number of garbage collections, erasures and folds for each benchmark, for both Greedy and COMBO approaches. We see that COMBO significantly reduces garbage collections, folds and erasures, essentially contributing both to improved response times as well as increased Flash life time. Presented are individual results and discussion for SLAC and FSAF approaches also, to see how each method improves Flash management.

TABLE II: Improvement in erasures, garbage collections and folds with both methods applied

	Erasures			GCs			Folds		
Bench-mark	Gree-dy	COM-BO	%Decre-ase	Gree dy	COM-BO	%Decre-ase	Gre-edy	COM-BO	%Decre-ase
s1	4907	4211	**14.18**	10	0	**100.00**	2294	1560	**32.00**
s2	2631	1324	**49.68**	11	1	**90.91**	1249	597	**52.20**
s3	5384	3219	**40.21**	25	5	**80.00**	2541	1563	**38.49**

Table III provides improvements with respect to the number of FTL-triggered garbage collections and erasures for all approaches, for each benchmark for SLAC. The most important observation from this table is that, on an average, FTL-triggered garbage collections reduce by 80%. This means the elimination of most of the undesirable peaks in the device response times depicted in Fig.2. Benchmarks MAD and MPEG were not considered in the calculation of the average: the percent improvement is high but number of garbage collections before and after employing SLAC is very small. Table II also provides number of erasures for each method. These results underline another important benefit of employing SLAC approach: the reduction in the number of erasures. Erasures determine the life time of a Flash and by reducing them we can achieve longer Flash life times. Reduced number of erasures also means significant energy reduction, as an erasure is the costliest of all Flash memory operations.

Benefits of SLAC approach can be understood by observing the way a normal FTL performs garbage collection. An FTL triggers garbage collection upon free block count reaching certain critical threshold. At such an instance, blocks are sorted by a metric decided by the garbage collection policy, and are subsequently reclaimed in the sorted order until enough free blocks are generated. For example, in Greedy approach, blocks are sorted by their dead page count, where as in Cost-benefit approach, they are sorted by cost-benefit value. Because of this, the cost of each fold operation may be different. SLAC, on the other hand, picks up blocks only with maximum benefits (whose dead page count is d_{Th}), doing away with costly sorting operation. The benefits associated manifest themselves in the reduction of the number of erasures, and improved garbage collection

TABLE II. SLAC: Improvements in number of FTL-triggered garbage collections and erasures

	FTL-triggered GCs		Erasures			FTL-triggered GCs		Erasures		
Bench-mark	Greedy	SLAC-Greedy	Greedy	SLAC-Greedy	%Decrease	Cost-benefit	SLAC-Cost-benefit	Cost-benefit	SLAC-Cost-benefit	%Decrease
CellPhone	23	**14**	5020	5000	**0.4**	28	**12**	5020	5000	**0.4**
Event Recorder	14	**13**	3345	3288	**1.7**	17	**14**	3343	3318	**0.75**
Fax	111	**19**	7659	7292	**4.79**	111	**19**	7659	7292	**4.79**
JPEG	21	**6**	1449	1410	**2.69**	26	**7**	1449	1423	**1.79**
MAD	2	**0**	134	96	**28.36**	2	**0**	134	96	**28.36**
MPEG	38	**7**	2647	2581	**2.49**	1	**0**	1756	1315	**33.54**
MP3	78	**0**	25414	25078	**1.32**	97	**0**	25414	25056	**1.41**

efficiency by reducing number of writes and reads during folding. However, one can observe that the reduction in the number of erasures is much less compared to the reduction in number of garbage collections. This is because of the fact that even though some FTL-triggered garbage collections are taken up in the slack, essentially the same amount of cleaning activity needs to be performed in both of the approaches.

It has to be noted that SLAC gains are heavily dependent upon the timing or slack characteristics of applications. When there is no slack, no micro garbage collections can

be performed. For example, the Event recorder benchmark is very write intensive: it records event data to Flash upon sudden influx of events. Continuous writes to the Flash in such cases trigger FTL to perform garbage collections automatically. Thus, we can see that improvement is less for event recorder with SLAC approach. This also explains why SLAC may not eliminate all FTL-triggered garbage collections. However, it has to be noted that SLAC does not worsen response times: it stops performing micro-garbage collections as soon as it detects very high request rates.

Table IV provides improvements with FSAF with respect to the number of garbage collections, erasures and folds for each benchmark, for both greedy and FSAF methods. The most important observation from this table is that, on an average, FSAF reduces number of erasures by 21.6%, by avoiding erasures associated with wear leveling dead data. Since erasures determine the life expectancy of Flash, endurance is proportionally improved. Reduced number of erasures also means significant energy reduction, as an erasure is the costliest of all Flash memory operations.

Table IV: FSAF: Improvement in erasures, garbage collections and folds

	Erasures			GCs			Folds		
Bench-mark	Gree-dy	FSAF	%Decre-ase	Gree dy	FSAF	%Decre-ase	Gree-dy	FSAF	%Decre-ase
s1	4907	4347	**11.41**	10	7	**30.00**	2294	1979	**13.73**
s2	2631	1760	**33.11**	11	5	**54.55**	1249	792	**36.59**
s3	5384	4293	**20.26**	25	14	**44.00**	2541	1976	**22.24**

Also, garbage collections are also reduced by 43% on the average compared to greedy method. This is achieved by generating enough free space in the device by performing proactive reclamation. In other words, this means the elimination of undesirable peaks in the device response times depicted in Fig.2. Similarly, folds are reduced by employing FSAF. By reclaiming dead blocks proactively, FSAF eliminates the need for creating replacement blocks for dead blocks, and thus, unnecessary fold operations are eliminated.

It has to be noted that FSAF approach also results in lesser algorithmic overhead. An FTL triggers garbage collection upon free block count reaching certain critical threshold. At such an instance, blocks are sorted by a metric decided by the garbage collection policy, and are subsequently reclaimed in the sorted order until enough free blocks are generated. For example, in Greedy approach, blocks are sorted by their dead page count. FSAF, on the other hand directly erases dead blocks, i.e., blocks only with maximum benefits, doing away with costly sorting operation. The benefits associated manifest themselves in the reduction of the number of erasures, and improved garbage collection efficiency by reducing number of writes and reads during folding.

It has to be noted that FSAF gains are heavily dependent upon the dead data content and distribution. However, since FSAF naturally switches to regular wear leveling and garbage collection operations when there is no dead data, its performance is at least as good as the normal case.

6.5. Overheads

In SLAC, the overhead associated with slack prediction is $O(n)$, arising mainly from the calculation of D_{dev}. Since n is set to a very small value, it is clear that the overhead is minimal. Selective folding uses FTL's block list, and may need to maintain a very small list of blocks to be folded in the slack. It also needs to scan blocks, which is an $O(k)$ operation, where is k the number of blocks. However, it has to be noted that by carrying out efficient folds in slack, selective folding reduces or eliminate the garbage collection burden on the FTL, so as to decrease the overall garbage collection overhead. Also, by setting d_{Th} to 32, we eliminate the need of any sorting activity during selective folding.

The overhead associated with FSAF comes from dead data detection and proactive reclamation. To detect dead data, FSAF needs to monitor writes to only three sections of Flash: the MBR, Volume ID and the FAT32 table itself. By reading and storing MBR and Volume ID at every format time, need for constructing formatting information at every Flash plug-in is eliminated. To detect which sector is being deleted, FSAF needs to maintain a buffer of size of maximum one sector. Also, finding out which sector is being deleted is an $O(s)$ operation, where s is the number of sector pointers stored in a single sector of the FAT32 table. Subsequent addition and deletion from the dead data list are all $O(1)$ operations. Thus, algorithmic overhead introduced by FSAF is only $O(s)$ per write. Since typically there are only 128 pointers per sector, this overhead is very minimal. Proactive reclamation, on the other hand, reduces the overall overhead on the system. Since proactive reclamation executes at a higher efficiency than a normal garbage

collection operation and also eliminates or delays regular garbage collections, effectively system overhead is significantly reduced.

CHAPTER 7

CONCLUSION AND FURTHER WORK

This work proposed a new FTL-based framework for improving application response times and overall Flash management that is consistent with the current system interfaces. The first method, SLAC breaks up lengthy GC operations into chunks and carries out the same efficiently, so as to achieve significant improvements in application response times, average write access times and the number of erasures. The second approach is a novel method to impart the awareness of file system operations at the FTL level, without changing any existing file system architectures. By being able to detect and treat dead data efficiently at the FTL level, this method achieves significant improvements in application response times, average write access times as well as erasures.

This approach can be carried over to Multi-Level Cell (MLC) NAND Flash based applications. By storing two bits per NAND cell, MLC Flashes offer higher densities, but perform poorer compared to basic Single-Level Cell (SLC) counterparts, owing to higher Flash management overheads in garbage collection, wear leveling and also error detection and correction. Also, the former have lesser life times. MLC Flashes can be targeted for broader range of applications by extending the proposed approach to improve their performance and life time.

REFERENCES

[1] A. Ban. Flash file system. United States Patent, no.5404485, April 1995.

[2] A. Ban. Wear leveling of static areas in Flash memory. US Patent 6,732,221. M-systems, May 2004.

[3] Elaine Potter, "NAND Flash End-Market Will More Than triple From 2004 to 2009", http://www.instat.com/press.asp?ID=1292&sku=IN0502461SI

[4] Golding, Richard; Bosch, Peter; Wilkes, John, "Idleness is not sloth". USENIX Conf, Jan. 1995

[5] Hyojun Kim Youjip Won , "MNFS: mobile multimedia file system for NAND Flash based storage device", Consumer Communications and Networking Conference, 2006. CCNC 2006. 3rd IEEE

[6] Hanjoon Kim, Sanggoo Lee, S. G., "A new Flash memory management for Flash storage system," COMPSAC 1999.

[7] Intel Corporation. "Understanding the Flash translation layer (ftl) specification". http://developer.intel.com/.

[8] J.W. Hsieh, L.-P. Chang, and T.-W. Kuo. Efficient On-Line Identification of Hot Data for Flash-Memory Management. In Proceedings of the 2005 ACM symposium on Applied computing, pages 838.842, Mar 2005.

[9] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho. "A space-efficient Flash translation layer for compact Flash systems". IEEE Transactions on Consumer Electronics, May 2002.

[10] J. C. Sheng-Jie Syu. An Active Space Recycling Mechanism for Flash Storage Systems in Real-Time Application Environment. 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Application (RTCSA'05), pages 53.59, 2005.

[11] Kawaguchi, A., Nishioka, S., and Motoda, H., "A Flash-memory Based File System", USENIX 1995.

[12] Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo, “Real-Time Garbage collection for Flash-Memory Storage Systems of Real-Time Embedded Systems”, ACM Transactions on Embedded Computing Systems, November 2004

[13] L.-P. Chang and T.-W. Kuo. An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems. In IEEE Real-Time and Embedded Technology and Applications Symposium, pages 187.196, 2002.

[14] Malik, V. 2001a.” JFFS—A Practical Guide”, http://www.embeddedlinuxworks.com/articles/jffs guide.html.

[15] Mei-Ling Chiang, Paul C. H. Lee, Ruei-Chuan Chang, “Cleaning policies in mobile computers using flash memory,” Journal of Systems and Software, Vol. 48, 1999.

[16] M.-L. Chiang, P. C. H. Lee, and R.-C. Chang. Using data clustering to improve cleaning performance for Flash memory. Software: Practice and Experience, 29-3:267.290, May 1999.

[17] Microsoft, “Description of the FAT32 File System”, http://support.microsoft.com/kb/154997

[18] Ohoon Kwon and Kern Koh, “Swap-Aware Garbage collection for NAND Flash Memory Based Embedded Systems”, Proceedings of the 7th IEEE CIT2007.

[19] Rosenblum, M., Ousterhout, J. K., “The Design and Implementation of a Log-Structured FileSystem,” ACM Transactions on Computer Systems, Vol. 10, No. 1, 1992.

[20] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S.-W. Park, and H.-J. Songe. “FAST: A log-buffer based ftl scheme with fully associative sector translation”. The UKC, August 2005.

[21] Toshiba 128 MBIT CMOS NAND EEPROM TC58DVM72A1FT00, http://www.toshiba.com, 2006.

[22] Wu, M., Zwaenepoel, W., “eNVy: A Non-Volatile, Main Memory Storage System”, ASPLOS 1994.

[23] Yuan-Hao Chang Jen-Wei Hsieh Tei-Wei Kuo, “Endurance Enhancement of Flash-Memory Storage, Systems: An Efficient Static Wear Leveling Design”, DAC’07

[24] Zaitcev, “The usbmon: USB monitoring framework”, http://people.redhat.com/zaitcev/linux/OLS05_zaitcev.pdf