Uniformity Improving Page Allocation for Flash Memory File Systems[i]

Seungjae Baek
Division of Information and CS
Dankook University
Seoul, Korea
ibanez1383@dankook.ac.kr

Seongjun Ahn
Software Laboratories
Samsung Electronics
Seoul, Korea
seongjunahn@gmail.com

Jongmoo Choi
Division of Information and CS
Dankook University
Seoul, Korea
choijm@dankook.ac.kr

Donghee Lee
School of Computer Science
University of Seoul
Seoul, Korea
dhl_express@uos.ac.kr

Sam H. Noh
School of Computer and Information Engineering
Hongik University
Seoul, Korea
samhnoh@hongik.ac.kr

ABSTRACT
Flash memory is a storage medium that is becoming more and more popular. Though not yet fully embraced in traditional computing systems, Flash memory is prevalent in embedded systems, materialized as commodity appliances such as the digital camera and the MP3 player that we enjoy in our everyday lives. This paper considers an issue in file systems that use Flash memory as a storage medium and makes the following two contributions. First, we identify the cost of block cleaning as the key performance bottleneck for Flash memory analogous to the seek time in disk storage. We derive and define three performance parameters, namely, utilization, invalidity, and uniformity, from characteristics of Flash memory and present a formula for block cleaning cost based on these parameters. We show that, of these parameters, uniformity most strongly influences the cost of cleaning and that uniformity is a file system controllable parameter. This leads us to our second contribution, designing the **mod**ification-**a**ware (MODA) page allocation scheme and analyzing how enhanced uniformity affects the block cleaning cost with various workloads. Real implementation experiments conducted on an embedded system show that the MODA scheme typically improves 20 to 30% in cleaning time compared to the traditional sequential allocation scheme that is used in YAFFS.

Categories and Subject Descriptors
D.4.2 [Operating System]: Storage Management--Secondary storage; D.4.3 [Operating System]: File Systems Management--File organization; C.4 [Computer Systems Organization]: Performance of Systems--Modeling techniques;

General Terms: Performance, Design, Experimentation, Verification.

Keywords: Flash memory, File system, Modeling, Uniformity, Implementation, Performance Evaluation

1. INTRODUCTION
Characteristics of storage media has been the key driving force behind the development of file systems. The Fast File System's (FFS) introduction of cylinder groups and the rule of thumb of keeping 10% of the disk as a free space reserve for effective layout was, essentially, to reduce seek time, which is the key bottleneck for user perceived disk performance [1]. Likewise, development of the Log-structured File System (LFS) was similarly motivated by the want to make large sequential writes so that the head movement of the disk would be minimized and to fully utilize the limited bandwidth that is available [2]. Various other optimization techniques that take into consideration the mechanical movement of the disk head has been proposed both at the file system level and the device level [3].

Similar developments have occurred for the MEMS-based storage media. Various scheduling algorithms that consider physical characteristics of MEMS devices such as the disparity of seek distances in the *x* and *y* dimensions have been suggested [4,5].

Recent developments in Flash memory technology have brought about numerous products that make use of Flash memory. In this paper, we explore and identify the characteristics of Flash memory and analyze how they influence the latency of data access. We identify the cost of block cleaning as the key characteristic that influences latency. A performance model for analyzing the cost of block cleaning is presented based on three parameters that we derive, namely, utilization, invalidity, and uniformity, which we define clearly later.

The model reveals that the cost of block cleaning is strongly influenced by uniformity just like seek is a strong influence for disk

[i] This work was partly supported by the IT R&D program of MIC/IITA [2006-S-040-01, Development of Flash Memory-based Embedded Multimedia Software] and supported in part by grant No. R01-2004-000-10188-0 from the Basic Research Program of the Korea Science & Engineering Foundation.

EMSOFT'07, September 30-October 3, 2007, Salzburg, Austria.

based storage. Also, we observe that most of algorithms trying to improve block cleaning time in Flash memory are, essentially, trying to maintain high uniformity of Flash memory. Furthermore, the model gives the upper bound of performance gain expected by developing a new algorithm. To validate the model and to analyze our observations in real environments, we design a new **mod**ification-**a**ware (MODA) page allocation scheme that strives to maintain high uniformity by grouping files based on their update frequencies.

We implement the MODA page allocation scheme on an embedded system that has 64MB of NAND Flash memory running the Linux kernel 2.4.19. The NAND Flash memory is managed by YAFFS (Yet Another Flash File System) [7] supported in Linux. We modify the page allocation scheme in YAFFS to MODA and compare its performance with the original scheme. Experimental results show that, by enhancing uniformity, the MODA scheme can reduce block cleaning time up to 47.8 seconds with an average of 9.8 seconds for the benchmarks considered. As the utilization of Flash memory increases, the performance enhancements become even larger. Performance is also compared with the DAC (Dynamic dAta Clustering) scheme that was previously proposed [8]. Results show that both the MODA and the DAC scheme performs better than the sequential allocation scheme used in YAFFS, though there are some delicate differences causing performance gaps between them.

The rest of the paper is organized as follows. In Section 2, we elaborate on the characteristics of Flash memory and explain the need for cleaning, which is the key issue that we deal with. We then review previous works that have dealt with this issue in Section 3. Then, we present a model for analyzing the cost of block cleaning in Section 4. In Section 5, we present the new page allocation scheme, which we refer to as MODA, in detail. The implementation details and the performance evaluation results are discussed in Section 6. We conclude the paper with a summary and directions for future works in Section 7.

2. FLASH MEMORY AND BLOCK CLEANING

Flash memory as a storage medium has characteristics that are different from traditional disk storage. These characteristics can be summarized as follows [9].

- **No seek time**: Access time in Flash memory is location independent similar to RAM. There is no "seek time" involved.
- **Overwrite limitation**: Overwrite is not possible in Flash memory. Flash memory is a form of EEPROM (Electrically Erasable Programmable Read Only Memory), so it needs to be erased before new data can be overwritten.
- **Asymmetric execution time**: Execution time for the basic operations in Flash memory is asymmetric. Traditionally, three basic operations, namely, read, write, and erase, are supported. An erase operation is used to clean used pages so that the page may be written to again. In general, a write operation takes an order of magnitude longer than a read operation, while an erase operation takes another order or more magnitudes longer than a write operation [21].

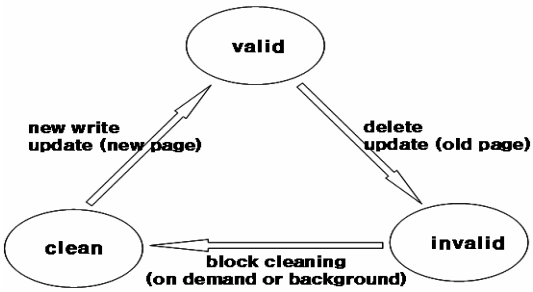

Figure 1. Page state transition diagram

- **Different operation unit**: The unit of operation is also different for the basic operations. While the erase operation is performed in block units, read/write operations are performed in page units.
- **Wear-leveling**: The number of erasures possible on each block is limited, typically, to 100,000 or 1,000,000 times.

These characteristics make designing software for Flash memory challenging and interesting [11].

Now, let us discuss why block cleaning is required and how it affects the performance of Flash memory file systems. Reading data or writing totally new data into Flash memory is simply like reading/writing to disk. A page in Flash is referenced/allocated for the data and data is read/written from/to that particular page. The distinction from a disk is that all reads/writes take a (much shorter) constant amount of time (though writes take longer than reads).

However, for updates of existing data, the story is totally different. As overwriting updated pages is not possible, various mechanisms for non-in-place update have been developed [7,12,13,14]. Though specific details differ, the basic mechanism is to allocate a new page, write the updated data onto the new page, and then, invalidate the original page that holds the (now obsolete) original data. The original page now becomes a dead or invalid page.

Note that, from the above discussions, a page can be in three different states, as shown in Figure 1. That is, a page can be holding legitimate data making it a valid page; we will say that a page is in a valid state if the page is valid. If the page no longer holds valid data, that is, it was invalidated by being deleted or by being updated, then the page is a dead/invalid page, and the page is in an invalid state. Note that a page in this state cannot be written to until the block it resides in is first erased. Finally, if the page has not been written to in the first place or the block in which the page resides has just been erased, then the page is clean. In this case, we will say that the page is in a clean state. Note that only pages that are in a clean state may be written to. Recall that in disks, there is no notion of an invalid state as in-place overwrites to sectors is always possible.

From the tri-state characteristics, we find that the number of clean pages diminishes not only as new data is written, but also as existing data is updated. In order to store more data and even to make updates to existing data, it is imperative that invalid pages be continually cleaned. Since cleaning is done via erase operation, which is done in block units, valid pages in the block to be erased must be copied to a clean block. This exacerbates the already

large overhead incurred by the erase operation needed for cleaning a block.

3. RELATED WORK

The issue of segment/block cleaning is not new and has been dealt with in both the disk and Flash memory realms. In this section, we discuss previous research in this area, especially in relation to the work presented in this paper.

Conceptually, the need for block cleaning in Flash memory is identical to the need of segment cleaning in the Log-structured File System (LFS). LFS writes data to a clean segment and performs segment cleaning to reclaim space occupied by obsolete data just like invalid pages are cleaned in Flash memory [2]. Segment cleaning is a vital issue in the Log-structured File System (LFS) as it greatly affects performance [15,16,17,18]. Blackwell et al. use the terms 'on-demand cleaning' and 'background cleaning' separately and try to reduce user-perceived latency by applying heuristics to remove on-demand cleaning [15]. Matthews et al. propose a scheme that adaptively incorporates hole-plugging into cleaning according to the changes in disk utilization [16]. They also consider how to take advantage of cached data to reduce the cost of cleaning.

Some of the studies on LFS are in line with an aspect of our study, that is, exploiting modification characteristics. Wang and Hu present a scheme that gathers modified data into segment buffers and sorts them according to the modification frequency and writes two segments of data to the disk at one time [17]. This scheme writes hot and cold modified data into different segments. Wang et al. describe a scheme that applies non-in-place update for hot-modified data and in-place updates for cold-modified data [18]. These two schemes, however, are disk-based approaches, and hence, are not appropriate for Flash memory.

In the Flash memory arena, studies for improving block cleaning have been suggested in many studies [8,11,12,19,20]. Kawaguchi et al. propose using two separate segments for cleaning: one for newly written data and the other for data to be copied during cleaning [12]. Wu and Zwaenepoel present a hybrid cleaning scheme that combines the FIFO algorithm for uniform access and locality gathering algorithm for highly skewed access distribution [19]. These approaches differ from ours in that their classification is mainly based on whether data is newly written or copied.

Chiang et al. propose the CAT (Cost Age Time) and DAC (Dynamic dAta Clustering) schemes [8]. CAT chooses blocks to be reclaimed by taking into account the cleaning cost, age of data in blocks, and the number of times the block has been erased. The DAC partitions Flash memory into several regions and place data into different regions according to their update frequencies. Later, we compare the performance of DAC that we implemented with the MODA scheme we propose, and explain what the differences are between the two schemes. One key difference between this work and ours is that we identify the parameters that influence the cost of block cleaning in Flash memory and provide a model for the cost based on these parameters. The model gives us what fundamental we need to focus on and how much gain we can expect, when we develop a new algorithm for Flash memory such as page allocation, block selection for cleaning, and background cleaning scheme.

There are also some noticeable studies regarding Flash memory. Kim et al. describe the difference between block level mapping and page level mapping [21]. They propose a hybrid scheme that not only handles small writes efficiently, but also reduces resources required for mapping information. Gal and Toledo present a recoverable Flash file system for embedded systems [13]. They conjecture that a better allocation policy and a better reclamation policy would improve endurance and performance of Flash memory. The same authors also present a nice survey, where they provide a comprehensive discussion of algorithms and data structures regarding Flash memory [11]. Chang et al. discuss a block cleaning scheme that considers deadlines in real-time environments [20]. Ben-Aroya analyzes wear-leveling problems mathematically and suggests separating the allocation and cleaning policies from the wear-leveling problems [24].

4. BLOCK CLEANING COST ANAYSIS

4.1. Model for Flash Memory

As mentioned in Section 3, Log-structured File System (LFS) requires segment cleaning to reclaim space occupied by obsolete data [2]. The authors of LFS have presented a formula for estimating the cleaning cost, that is:

$$write\ cost = \frac{2}{1-u} \quad \ldots (1)$$

where
u: $utilization (0 \le u \le 1)$

The above formula provides a reasonable model for LFS, revealing that the cleaning cost increases as the utilization increases. It also serves as a policy-making guideline when choosing the block to be cleaned. However, there are some limitations in adopting this formula to Flash memory file systems. First, the cleaning cost depends not only on the utilization, but also on the distribution of invalid pages. However, Equation (1) does not reflect this in the cost model. Second, due to the overwrite limitation of Flash memory, block cleaning includes the erase operation for invalid pages, which also affects the block cleaning cost. Hence, a model for Flash memory should have the capability to reflect the amount of invalid pages on the cost analysis. Finally, the equation provides a relative performance measure. That is, the write cost in Equation (1) is expressed as a multiple of the time required when there is no cleaning overhead. What we desire in a model, however, is the absolute time required to clean blocks for a given Flash memory.

To derive an appropriate model for Flash memory, we first identify three parameters that affect the cost of block cleaning. They are defined as follows:

- Utilization (u): the fraction of valid pages in Flash memory
- Invalidity (i): the fraction of invalid pages in Flash memory
- Uniformity (p): the fraction of blocks that are uniform in Flash memory, where a uniform block is a block that does *not* contain both valid and invalid blocks simultaneously.

Figure 2 shows three page allocation situations where utilization and invalidity are the same, but uniformity is different. Since there are eight valid pages and eight invalid pages among the 20 pages for all three cases, utilization and invalidity are both 0.4.

However, there is one, three, and five uniform blocks in Figure 2(a), (b), and (c), respectively, hence uniformity is 0.2, 0.6, and 1, respectively. (Another definition of uniformity would be "1 – the fraction of blocks that have both valid and invalid pages.") Note that, for the case of Figure 2(a), 8 page copies and 4 block erases are required to reclaim all invalid pages. For the case of Figure 2(b), 4 page copies and 3 block erases are required, while only 2 block erases are needed for case Figure 2(c).

Utilization determines, on average, the number of valid pages that need to be copied. Invalidity determines the number of blocks that are candidates for erasing. Finally, uniformity determines the actual number of pages and blocks to be copied and erased. From these observations, we can formulate the cost of block cleaning as follows:

$$cleaning\ cost = B*((1-p)+i*p)*e_t + P*(1-p)*\frac{u}{u+i}*(r_t+w_t) \quad \ldots (2)$$

where
u: utilization $(0 \le u \le 1)$
i: invalidity $(0 \le i \le 1-u)$
p: uniformity $(0 \le p \le 1)$
B: number of blocks in Flash memory
P: number of pages in Flash memory
r_t, w_t, e_t : read, write, erase execution time, repectively

In Equation (2), *B*((1- p)+i*p))* represents the number of blocks to be erased. Specifically, *B*(1- p)* denotes the number of non-uniform blocks (containing both valid and invalid pages) and *B*(i*p)* denotes the number of uniform blocks that have invalid pages only. The number of pages to be copied is represented as the term *P*(1- p)*u / (u+i)*. *P*(1- p)*u* refers to the average number of valid pages in non-uniform blocks. However, when there are clean blocks in Flash memory, *P*(1- p)*u* has a tendency to underestimate the number of valid pages by uniformly distributing *u* evenly among the clean blocks. To handle this special case, we divide *P*(1- p)*u* by *(u+i)*. Validation of this model is presented in Section 6.3.2.

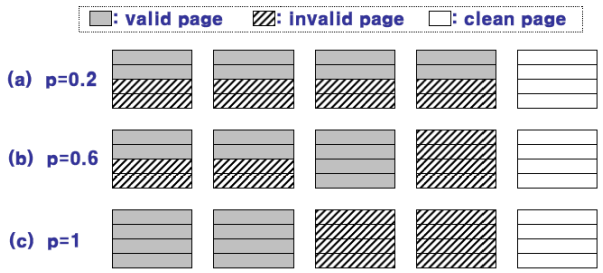

Figure 2. Situation where utilization (*u*=0.4) and invalidity (*i*=0.4) remains unchanged, while uniformity (*p*) changes (a) *p* = 0.2 (b) *p* = 0.6 (c) *p* = 1

4.2. Implication of the Parameters on Block Cleaning Costs

Figure 3 shows the analytic results of the cost of block cleaning based on the derived model. In the figure, the *x*-axis is utilization, the *y*-axis is uniformity, and the *z*-axis is the cost of block cleaning. For this graph, we set invalidity as 0.1 and use the raw data of a small block 64MB NAND Flash memory [10]. The execution times for read, write, and erase operations are 15us, 200us, and 2000us, respectively. Each block has 32 pages where the size of each page is 0.5KB.

The main observation from Figure 3 is that the cost of cleaning increases dramatically when utilization is high and uniformity is low. We also conduct analysis with different values of invalidity and with the raw data of a large block 1GB NAND Flash memory[10], which shows similar trends observed in Figure 3.

Figure 4 depicts how each parameter affects the cost of block cleaning. In each figure, the initial values of the three parameters are all set to 0.5. Then, we decrease utilization in Figure 4(a), decrease invalidity in Figure 4(b), and increase uniformity in

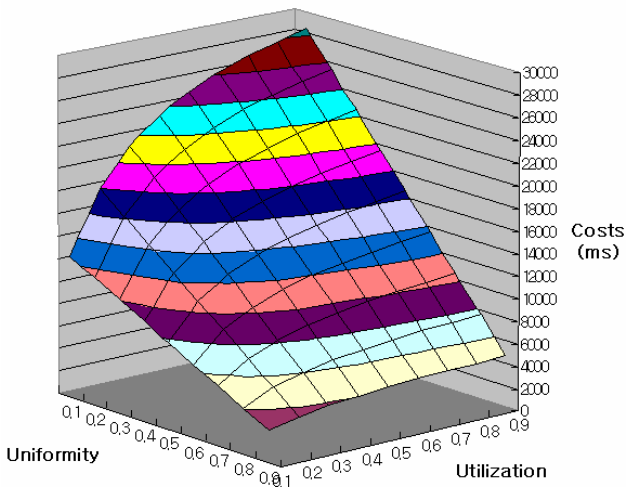

Figure 3. Block cleaning costs

Figure 4(c). From these figures, we find that the impact of utilization and uniformity on block cleaning cost is higher than that of invalidity. Since utilization is almost uncontrollable at the file system level, this implies that to keep cleaning cost down keeping uniformity high may be a better approach than trying to keep invalidity low through frequent cleaning.

5. PAGE ALLOCATION SCHEME THAT STRIVES FOR UNIFORMITY

When pages are requested in file systems, in general, pages are allocated sequentially [7,12]. Flash file systems tend to follow this approach and simply allocate the next available clean page when a page is requested, not taking into account any of the characteristics of the storage media.

We propose an allocation scheme that takes into account the file modification characteristics such that uniformity may be maximized. The allocation scheme is **mod**ification-**a**ware (MODA) as it distinguishes data that are hot-modified, that is, modified frequently and data that are cold-modified, that is, modified infrequently. Allocation of pages for the distinct type of data is done from distinct blocks.

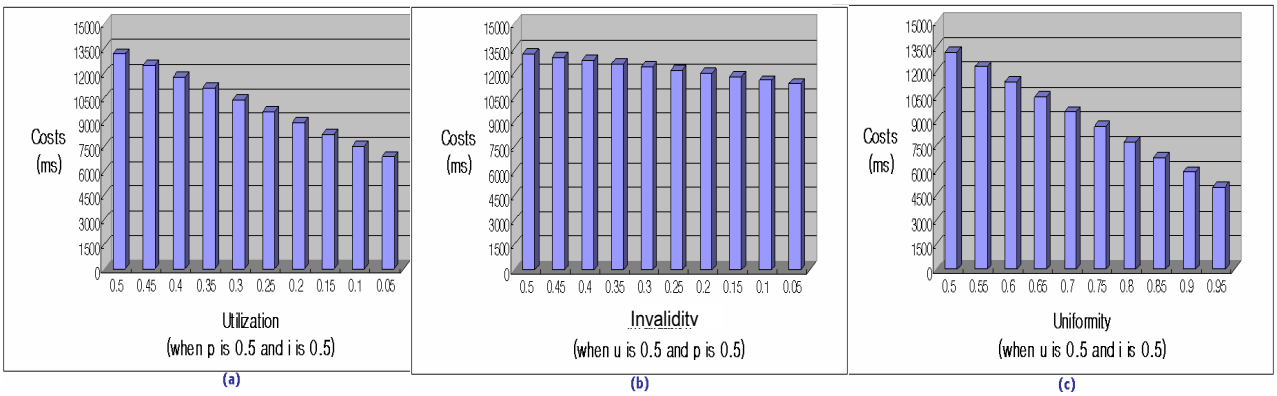

Figure 4. How block cleaning cost is affected by (a) utilization, (b) invalidity, and (c) uniformity as the other two parameters are kept constant at 0.5

The motivation behind this scheme is that by classifying hot-modified pages and allocating them to the same block, we will eventually turn the block into a uniform block filled with invalid pages. Likewise, by classifying cold-modified pages and allocating them together, we will turn this block into a uniform block filled with valid pages. Pages that are neither hot-modified nor cold-modified are sequestered so that they may not corrupt the uniformity of blocks that hold hot and cold modified pages.

The natural question, then, is how to classify hot/cold-modified data. Our solution is to use two levels of modification-aware classifications as shown in Figure 5. At the first level, we make use of static properties, and dynamic properties are used at the second level. This classification is based on the skewness in page modification distribution [18, 20], exploited many other previous research [8, 11, 17, 18, 25].

As static property, we distinguish system data and user date as the modification characteristics of the two are quite different. The superblock and inodes are examples of system data, while data written by users are examples of user data. We know that inodes are modified more intensively than user data, since any change to the user data in a file causes changes to its associated inode.

User data is further classified at the second level, where its dynamic property is used. In particular, we make use of the modification count. Keeping the modification count for each page, however, may incur considerable overhead. Therefore, we choose to monitor at a much larger granularity and keep a modification count for each file which is updated when the modification time is updated.

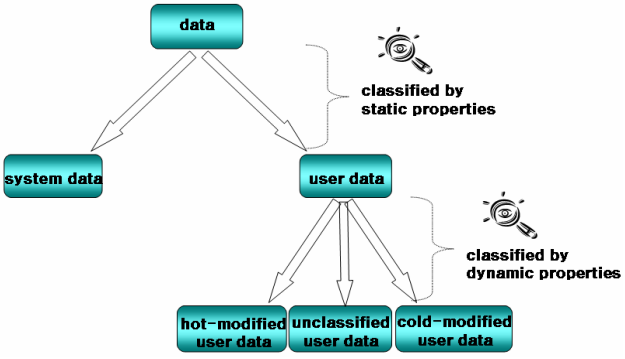

Figure 5. Two level (static and dynamic property) classification used in MODA scheme

For classification with the modification count, we adopt the MQ (Multi Queue) algorithm [25]. Specifically, it uses multiple LRU queues numbered Q_0, Q_1,…, Q_{m-1}. Each file stays in a queue for a given *lifetime*. When a file is first written (created), it is inserted into Q_0. If a file is modified within its *lifetime*, it is promoted from Q_i to Q_{i+1}.On the other hand, if a file is not modified within its *lifetime*, it is demoted from Q_i to Q_{i-1}. Then, we classify a file promoted from Q_{m-1} as hot-modified data, while a file demoted from Q_0 as cold-modified data. Files within queues are defined as unclassified data. In our experiments, we set m as 2 and lifetime as 100 time-ticks (time is virtual that ticks at each modification request). In other words, a file modified more than 2 times is classified as hot, while a file in Q_0 that has not been modified within the recent 100 modification requests is classified as cold. We find that MODA with different values of m = 3 and/or lifetime = 500 shows similar behavior.

6. PERFORMANCE EVALUATION

6.1. Platform and Implementation

We have implemented the MODA scheme on an embedded system. Hardware components of the system include a 400MHz XScale PXA CPU, 64MB SDRAM, 64MB NAND Flash memory, 0.5MB NOR Flash memory, and embedded controllers such as LCD, UART and JTAG [22]. The same NAND Flash memory that was used to analyze the cost of block cleaning in Figures 3 and 4 is used here.

The Linux kernel 2.4.19 was ported on the hardware platform and YAFFS is used to manage the NAND Flash memory [7]. We modify the page allocation scheme in YAFFS and compare the performance with the native YAFFS. We will omit a detailed discussion regarding YAFFS, but only describe the relevant parts below. Interested readers are directed to [6,7] for details of YAFFS.

The default page allocation scheme in YAFFS is the sequential allocation scheme. We implemented the MODA scheme in YAFFS and will refer to this version of YAFFS, the MODA-YAFFS or simply MODA. In MODA-YAFFS, we modified functions such as *yaffs_WriteChunkDataToObject(), yaffs_FlushFile(), yaffs_UpdateObjectHeader()* in yaffs_guts.c and *init_yaffs_fs(), exit_yaffs_fs()* in yaffs_fs.c.

The block cleaning scheme in YAFFS is invoked at each write request. There are two modes of cleaning: normal and aggressive. In normal mode, YAFFS chooses the block that has the largest number of invalid pages among the predefined number of blocks (default setting is 200). If the chosen block has less than 3 valid pages, it reclaims the block. Otherwise, it gives up on the reclaiming. If the number of clean blocks is lower than a predefined number (default setting is 6), the mode is converted to aggressive. In aggressive mode, YAFFS chooses a block that has invalid pages and reclaims the block without checking the number of valid pages in it. The block cleaning scheme in MODA-YAFFS is exactly the same. We do add a new interface for block cleaning that may be invoked at the user level for ease of measurement.

6.2. The Workload

To gather sufficient workloads for evaluating our scheme, a comprehensive survey of Flash memory related papers was done. The result of this survey is the following 7 benchmarks used in our implementation experiments.

- **Camera** benchmark : This benchmark executes a number of transactions repeatedly, where each transaction consists of three steps; creating, reading and deleting some of these files randomly. Such actions of taking, browsing, and erasing pictures are common behaviors of digital camera users, as observed in [21, 26].
- **Movie player** benchmark : This benchmark simulates the workload of a Portable Media Players [26].
- **Phone** benchmark : This benchmark simulates the behavior of a cellular phone [13].
- **Recorder** benchmark : This benchmark models the behavior of an event recorder such as an automotive black box and remote sensors [13].
- **Fax machine** benchmark : This benchmark initially creates two files. Then, it creates four new files and updates a history file (200 bytes) when it receives a fax. This behavior can be observed in fax machines, answering machines, and music players [13].
- **Postmark** benchmark: This benchmark creates a large number of randomly sized files. It then executes read, write, delete, and append transactions on these files [23].
- **Andrew** benchmark : This benchmark was originally developed for testing disk based file systems, but many Flash memory studies have used it for performance evaluation [12, 21]. The Andrew benchmark consists of 5 phases, but in our study, we only execute the first two phases.

These benchmarks can be roughly grouped into three categories: sequential read/write intensive workloads, update intensive workloads, and multiple files intensive workloads. The first group includes the Camera and Movie benchmarks that access large files sequentially. The Phone and Recorder benchmarks are typical examples of update intensive workloads that manipulate a small number of files and update them intensively. The Fax, Postmark and Andrew benchmarks access multiple files with different access probabilities and can be group into the third category.

6.3. Performance Evaluation

6.3.1 Performance results

Table 1 shows performance results both measured by executing benchmarks and estimated by the model. Before each execution the utilization of Flash memory is set to 0, that is, the Flash memory is reset completely. Then, we execute each benchmark on YAFFS and MODA-YAFFS and measure its execution time reported in the 'Benchmark Running Time' column. Note that the only difference between YAFFS and MODA-YAFFS is the page

Table 1. Performance comparison of YAFFS and MODA-YAFFS for the benchmarks when utilization at start of execution is 0 (The unit of time measurement is in seconds)

Benchmark	Scheme	Benchmark Running Time	Performance Parameters			Estimated Results			Measured Results		
			U	I	P	# of Erase	# of Copy	Cleaning Time	# of Erase	# of Copy	Cleaning Time
Camera	YAFFS	38	0.3	0.002	0.98	76	2192	2.83	69	1516	9.60
	MODA	37	0.3	0.002	0.99	17	317	0.42	10	62	7.56
Movie	YAFFS	481	0.99	0.0001	0.99	10	319	0.41	10	7	24.56
	MODA	481	0.99	0.0001	0.99	1	31	0.04	1	3	24.54
Phone	YAFFS	90	0.05	0.32	0.62	2151	6047	11.18	1398	6047	12.08
	MODA	90	0.05	0.22	0.72	1606	6052	10.24	1011	6063	10.80
Recorder	YAFFS	32	0.005	0.16	0.83	1128	692	2.81	626	699	2.00
	MODA	32	0.005	0.08	0.90	636	692	1.95	344	690	1.76
Fax machine	YAFFS	100	0.86	0.0087	0.73	1024	31710	40.74	1001	30996	60.99
	MODA	99	0.86	0.0087	0.97	76	2407	3.14	76	1383	23.19
Postmark	YAFFS	17	0.08	0.0107	0.90	393	10158	13.17	357	10147	16.18
	MODA	17	0.08	0.0107	0.93	285	7057	9.17	248	6652	11.39
Andrew	YAFFS	33	0.09	0.0008	0.90	372	10174	13.16	345	10060	16.32
	MODA	32	0.09	0.0008	0.98	92	1828	2.40	62	1004	3.64

allocation scheme. Also, after executing the benchmark, we measure the performance parameters, namely utilization, invalidity, and uniformity of Flash memory denoted as 'U', 'I', 'P' in the Table 1

Using the measured performance parameters and the model proposed in Section 4.1, we can estimate the number of erase and copy operations required to reclaim all the invalid pages. Also, the model gives us the expected cleaning time. These estimated results are reported in the 'Estimated Results' column. Finally, we actually measure the number of erase and copy operations and cleaning times to reclaim all invalid pages, which are reported in

the 'Measured Results' column. The measured results reported are averages of three executions for each case unless otherwise stated.

6.3.2 Model Validation

Table 1 shows that the number of erase and copy operations estimated by the model are similar to those measured by real implementaion, though the model tends to overestimate the erase operations when invalidity is high. These similarities imply that the model is fairly effective to predict how many operations are required under given status of Flash memory.

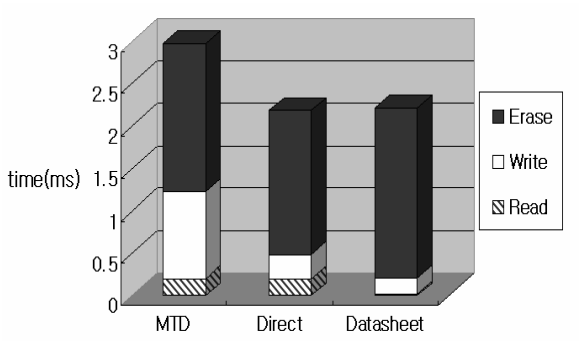

Figure 6. Execution time at each level

However, there are noticable differences between the measured and estimated block cleaning times. Through sensitive analysis, we find two main reasons behind these differences. The first reason is that the model requires the read, write, and erase times to estimate the block cleaning time. The simplest way to determine these values is by using the data sheet provided by the Flash memory chip vendor. However, through experiments we observe that the values reported in the datasheet and the actual time seen at various levels of the system differ considerably. Figure 6 shows these results. The results shows that while the datasheet reports read, write, and erase times of 0.01ms, 0.2ms, and 2ms, respectively, for the Flash memory used in our experiments, the times observed for directly accessing Flash memory at the device driver level is 0.19ms, 0.3ms, and 1.7ms, respectively. Furthermore, when observed just above the MTD(Memory Technology Device) layer, the read, write, and erase times are 0.2ms, 1.03ms, and 1.74ms, respectively, with large variances. These variances influence the accuracy of the model drastically. Since the YAFFS runs on the basis of the MTD layer, the estimated results reported in Table 1 use the times observed above the MTD layer.

The second reason is that block cleaning not only causes copy and erase overhead, but it also incurs software manipulating overhead. Specifically, YAFFS does not manage block and page status information in main memory in order to minimize the RAM footprint. Hence, it needs to read Flash memory to detect the blocks to be cleaned and how many valid pages the blocks have. Due to this overhead, there are differences between the measured and estimated cleaning times. The overhead also explains why the difference increases as utilization increases. However, the cleaning time difference between YAFFS and MODA for the estimates derived from the model and the measurements are quite similar, which implies that the model is a good indicator of the performance characteristics of Flash memory.

6.3.3 Effects of Uniformity

By comparing the results of YAFFS and those of MODA, we make the following observations.

- The benchmark execution time is the same for YAFFS and MODA. This implies that there is minimal overhead for the additional computation that may be incurred for data classification.
- The MODA scheme maintains high uniformity, which leads to block cleaning time reductions of up to 47.8 seconds (for Fax machine benchmark) with an average of 9.8 seconds for the benchmarks considered.
- The performance gains of MODA for Movie and Camera benchmarks are minimal. Our model reveals that the original sequential page allocation scheme used in YAFFS also keeps high uniformity making it difficult to obtain considerable gains.
- The gains of MODA for Phone and Recorder benchmark are also trivial, even though there is room for enhancing uniformity. Careful analysis reveals these benchmarks access only a small number of files; six files for Phone and two files for Recorder. Since the MODA classifies hot/cold data on the file-level granularity, the classification turns out to be ineffective. These experiments disclose the limitation of the MODA scheme and suggest that page-level classification may be more effective for some benchmarks.

We also experiment with combinations of two or more benchmarks such as 'Camera + Phone' simulating activities of recent cellular phone that have digital camera capabilities and 'Movie + Recorder + Postmark' simulating a PMP player that uses an embedded database to maintain movie titles, actor libraries and digital rights. Experiments show that the trends for multiple benchmark executions are similar to those of the Postmark results reported in Table 1. For example, for the combination of 'Movie + Recorder + Postmark', the block cleaning time of YAFFS and MODA are 34.82 and 22.58 seconds, while uniformity are 0.73 and 0.84, respectively. We also find that the interferences among benchmarks drive uniformity of Flash memory low, even for large sequential multimedia files.

6.3.4 Effects of Utilization

In real life, utilization of Flash memory will rarely be 0. Hence, we conduct similar measurements as was done for Table 1, but varying the initial utilization value. Figure 7 shows the results of executing Postmark under the various initial utilization values.

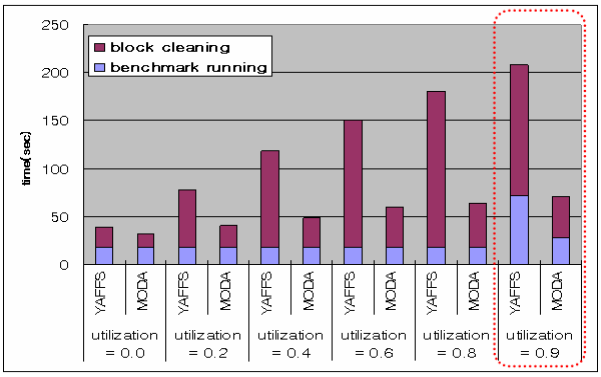

Figure 7. Results reported under various initial utilization values

Utilization was artificially increased by executing the Andrew benchmark before each of the measurements. Exact same experiments were conducted with utilization adjusted by pre-executing the Postmark benchmark, but the result trend is similar, hence we only report one set of these results.

For the moment, ignore the results reported when utilization is 0.9, which shows somewhat different behavior. We come back to discuss these results shortly. The results in Figure 7 show that block cleaning time increases as utilization increases confirming what we had observed in Figure 4(a). In Figure 3, our model shows that under high utilization, enhancement of uniformity leads to greater reduction in cleaning time. Figure 7 validates this expectation by showing that the difference in block cleaning time between YAFFS and MODA-YAFFS increases as utilization increases.

Let us now discuss results reported when the initial utilization is 0.9. Observe that the results are different from results with lower utilization values, in that the benchmark running time is much higher, more so for YAFFS. This is because YAFFS is confronted with a lack of clean blocks during execution, and hence, turns the block cleaning mode to aggressive. As a result, on-demand block cleaning occurs frequently increasing the benchmark running time to 72 seconds, four times the running time compared to when utilization is lower. Note that in MODA-YAFFS, the running time does increase, but not as significantly. This is because the MODA allocation scheme allows for more blocks to be kept uniform, and hence aggressive on-demand block cleaning is invoked less.

6.3.5 Effects of Periodic block cleaning

In YAFFS, block cleaning is invoked at each write request and attempts to reclaim at most one block at each trial. In other file system, block cleaning is invoked when free space becomes smaller than a predefined lower threshold and attempts to reclaim blocks until it becomes larger than an upper threshold [2, 12]. On the contrary, block cleaning can happen when the system is idle [15]. Ideally, if this can happen, then all block cleaning costs may be hidden from the user. Whether this is possible or not will depend on many factors including the burstiness of request arrival, idle state detection mechanism, and so on.

To see how periodic block cleaning affects performance we conduct the following sequence of executions. Starting from utilization zero, that is, a clean Flash memory state, we repeatedly execute Postmark until the benchmark can no longer complete as Flash memory completely fills up. During this iteration, two different actions are taken. For Figures 8(a) and (c), nothing is done between each execution. That is, no form of explicit cleaning is performed. For Figures 8(b) and (d), block cleaning is performed between benchmark executions. This represents a case where block cleaning is occurring periodically. Figures 8(a) and (b) are the measurement results for YAFFS, while Figures 8(c) and (d) are results for MODA. The utilization values reported on the *x*-axis is the value before each benchmark execution.

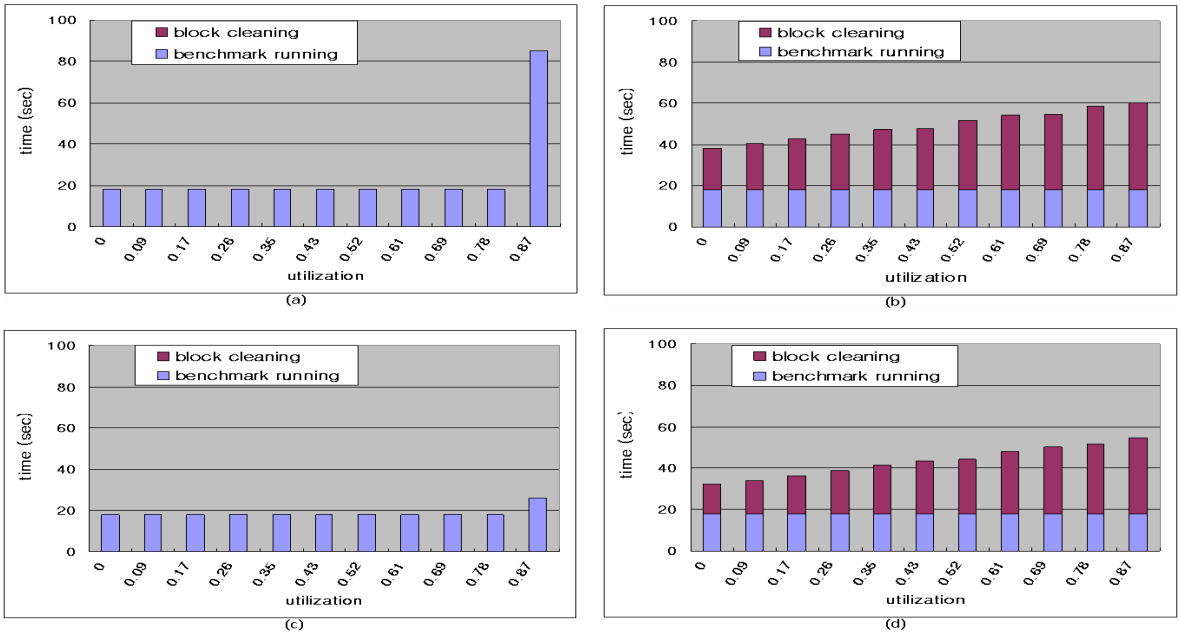

Figure 8. Execution time for Postmark for YAFFS (a) without periodic cleaning, (b) with periodic cleaning and for MODA (c) without periodic cleaning, (d) with periodic cleaning

The results here verify what we would expect based on observations from Section 6.3.4. As utilization is kept under some value, YAFFS and MODA perform the same (with or without periodic cleaning). Without periodic cleaning, once past that threshold, 0.87 in Postmark, the benchmark execution time abruptly increases for YAFFS, while for MODA, it increases only slightly. This is because enough clean blocks are being maintained in MODA. When block cleaning is invoked periodically, the benchmark execution time may be maintained at a minimum. However, the problem with this approach is that periodic cleaning itself incurs overhead, and the issue becomes whether this overhead can be hidden from the user or not. As this issue is beyond the scope of this paper, we leave this matter for future work. Note that the periodic cleaning times are smaller in MODA than those in YAFFS, implying the periodic cleaning also get benefits from keeping uniformity high.

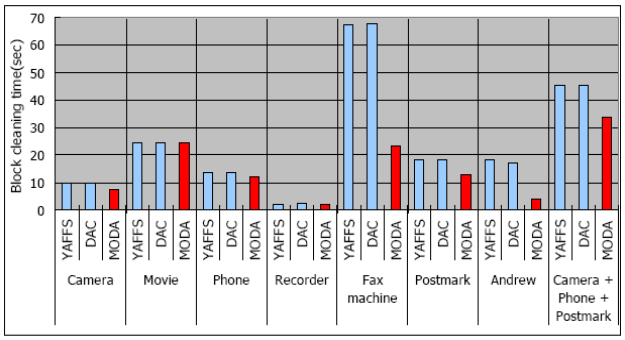

Figure 9. Performance Comparison between MODA and DAC

6.4. MODA vs DAC

In this subsection, we compare MODA with the DAC scheme proposed by Chiang et al. [8]. In the DAC scheme, Flash memory is partitioned into several regions and data are moved toward top/bottom region if their update frequencies increase/decrease. Both MODA and DAC try to cluster data not only at block cleaning time, but also at data updating time. We implement the DAC schemes into YAFFS and set the number of regions as 4 as this is reported to have shown the best performance [8].

Figure 9 shows the results between for the MODA and DAC schemes. (The results for YAFFS are shown for comparisons sake.) We execute each benchmark under the same conditions described in Table 1. The results show that the MODA scheme performs better than the DAC scheme.

Detailed examinations reveal that the performance gap between the MODA and the DAC schemes comes from two sources. One is that the DAC scheme clusters data into the cold region if their update frequency is low. This may cause the real cold-modified data and newly written data (which may actually be hot-modified data) to coexist on the same block, which reduces uniformity. However, the MODA scheme groups newly written data into separate blocks managed by the unclassified manager shown in Figure 5, segregating them from the cold-modified data.

The second source is that the MODA scheme uses a two-level classification scheme distinguishing system data and user data (static property) at the first level, then further classifying user data based on their modification counts (dynamic property) at the second level. But, by only considering the dynamic property of the data, the DAC scheme is not able to gather enough information to make a timely distinction between the two types. When we apply static property based classification into the DAC scheme, its performance comes close to MODA.

7. CONCLUSION

Two contributions are made in this paper. First, we identify the cost of block cleaning as the key performance bottleneck for Flash memory analogous to the seek time in disk storage. We derive three performance parameters from features of Flash memory and present a formula for block cleaning cost based on these parameters. We show that, of these parameters, uniformity is the key controllable parameter that has a strong influence on the cost. This leads us to our second contribution, which is a new **mod**ification-**a**ware (MODA) page allocation scheme that strives to maintain high uniformity. Using the MODA scheme, we validate our model and evaluate performance characteristics with the views of uniformity, unitization and periodic cleaning.

We are considering two research directions for future work. One direction is enhancing the proposed MODA scheme that can keep uniformity high for benchmarks manipulating small number of files. Another direction is in the development of an efficient block cleaning scheme. Uniformity is influenced by not only the page allocation scheme, but also by the block cleaning scheme. We need to investigate issues such as defining and finding idle time to initiate block cleaning and deciding which blocks and how many of these blocks should be reclaimed once reclaiming is initiated.

ACKNOWLEDGEMENT

We would like to thank to Prof. Sang Lyul Min, Seoul National University. Without his contribution, this paper would not be possible. We would also like to thank the anonymous reviewers for their constructive suggestions and comments.

8. REFERENCES

[1] M. McKusick, W. Joy, S. Leffler, and R. Fabry, "A Fast File System for UNIX", ACM Transactions on Computer Systems, 2(3), pp. 181-197, Aug., 1984.

[2] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system", ACM Transactions on Computer Systems, vol. 10, no. 1, pp. 26-52, 1992.

[3] William Stalling, "Operating Systems: Internals and Design Principles", 5th Edition, Pearson Prentice Hall, 2004.

[4] H. Yu, D. Agrawal, and A. E. Abbadi, “Towards optimal I/O scheduling for MEMS-based storage,” Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS’03), 2003.

[5] S. W. Schlosser and G. R. Ganger, “MEMS-based storage devices and standard disk interfaces: A square peg in a round hole?” Proceedings of 3rd USENIX Conference on File and Storage Technologies (FAST’04), 2004.

[6] S. Lim, K. Park, “An Efficient NAND Flash File System for Flash Memory Storage”, IEEE Transactions on Computers, Vol. 55, No. 7, July, 2006.

[7] Aleph One, "YAFFS: Yet another Flash file system", www.aleph1.co.uk/yaffs/.

[8] 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, vol. 29, no. 3, pp. 267-290, 1999.
[9] Ashok K. Sharma, "Advanced Semiconductor Memories: Architectures, Designs, and Applications", WILEY Interscience, 2003.
[10] Samsung Electronics, “NAND Flash Data Sheet”, www.samsung.com/Products/Semiconductor/NANDFlash.
[11] E. Gal and S. Toledo, "Algorithms and Data Structures for Flash Memories", ACM Computing Surveys, vol. 37, no. 2, pp 138-163, 2005.
[12] A. Kawaguchi, S. Nishioka and H. Motoda, "A Flash-memory based file system", Proceedings of the 1995 USENIX Annual Technical Conference, pp. 155-164, 1995.
[13] E. Gal and S. Toledo, "A transactions Flash file system for microcontrollers", Proceedings of the 2005 USENIX Annual Technical Conference, pp. 89-104, 2005.
[14] D. Woodhouse, "JFFS: The journaling Flash file system", Ottawa Linux Symposium, 2001, http://source.redhat.com/jffs2/jffs2.pdf.
[15] T. Blackwell, J. Harris, and M. Seltzer, "Heuristic cleaning algorithms in log-structured file systems", Proceedings of the 1995 Annual Technical Conference, pp. 277-288, 1993.
[16] J. Matthews, D. Roselli, A. Costello, R. Wang, and T. Anderson, "Improving the performance of log-structured file system with adaptive methods", ACM Symposiums on Operating System Principles (SOSP), pp. 238-251, 1997.
[17] J. Wang and Y. Hu, "WOLF - a novel reordering write buffer to boost the performance of log-structured file system", Proceedings of the USENIX Conference on File and Storage Technologies (FAST), pp. 46-60, 2002.
[18] W. Wang, Y. Zhao, and R. Bunt, "HyLog: A High Performance Approach to Managing Disk Layout", Proceedings of the USENIX Conference on File and Storage Technologies (FAST), pp. 145-158, 2004.
[19] M. Wu and W. Zwaenepoel, "eNVy: a non-volatile, main memory storage system", Proceeding of the 6th International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS), pp. 86-97, 1994.
[20] L. P. Chang, T. W. Kuo and S. W. Lo, "Real-time garbage collection for Flash memory storage systems of real time embedded systems", ACM Transactions on Embedded Computing Systems, Vol. 3, No. 4, pp. 837-863, 2004.
[21] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, Y. Cho, "A space-efficient Flash translation layer for CompactFlash systems", IEEE Transactions on Consumer Electronics, vol. 48, no. 2, pp. 366-375, 2002.
[22] EZ-X5, www.falinux.com/zproducts.
[23] J. Katcher, "PostMark: A New File System Benchmark", Technical Report TR3022, Network Appliance Inc., 1997.
[24] A. Ben-Aroya, “Competitive analysis of flash-memory algorithms”, M.Sc. paper, Apr. 2006, www.cs.tau.ac.il/~stoledo/Pubs/flash-abrhambe-msc.pdf.
[25] Y. Zhou, P. M. Chen, and K. Li, “The Multi-Queue Replacement Algorithm for Second-Level Buffer Caches, Proceeding of the 2001 USENIX Annual Technical Conference, June, 2001.
[26] H. Jo, J. Kang, S. Park, J. Kim, and J. Lee, ”FAB: Flash-Aware Buffer Management Policy for Portable Media Players”, IEEE Transactions on Consumer Electronics, Vol. 52, No. 2, May, 2006.