SAW: System-Assisted Wear Leveling on the Write Endurance of NAND Flash Devices

Chundong Wang and Weng-Fai Wong
School of Computing
National University of Singapore
Email: {wangc.nus@gmail.com, wongwf@comp.nus.edu.sg}

ABSTRACT

The write endurance of NAND flash memory adversely impacts the lifetime of flash devices. A flash cell is likely to wear out after undergoing excessive *program/erase (P/E) flips*. Wear leveling is hence employed to spread erase operations as evenly as possible. It is traditionally conducted by the *flash translation layer* (FTL), a management firmware residing in flash devices. In this paper, we shall propose a novel wear leveling algorithm involving the operating system (OS). We will show that our *operating **S**ystem-**A**ssisted **W**ear leveling* (SAW) algorithm can significantly improve the wear evenness. SAW takes advantage of OS's knowledge about files at a higher level of abstraction, and provides useful hints to the lower-level FTL to accommodate data. A prototype based on a file system and an FTL has been developed to verify the effectiveness of SAW. Experiments show that wear evenness can be improved by as much as 85.0% compared to the state-of-the-art FTL wear leveling schemes.

1. INTRODUCTION

Today it is economically feasible to utilize NAND flash devices for secondary storage in both embedded systems and general-purpose computing systems. The issue of *write endurance*, however, hinders the further use of NAND flash as the lifespan of a flash device is inherently limited.

Write endurance refers to the limited cycles of *program/erase (P/E) flips* on a flash *page*. A page is the unit for write and read operations in NAND flash. It consists of thousands of cells. A page cannot be directly reprogrammed with updates due to the physical characteristics of flash cells. It has to be erased first. Programming a page is to selectively set some bits to '0'. All bits are reset to '1' by an erase operation. Erase operations must be performed in the unit of a *block* that consists of multiple pages. Excessive P/E flips will wear out a page, and damage its ability to retain data. The limitation of P/E flips for single-level cell (SLC) NAND flash which stores one bit in a cell, is 100,000 cycles. For the denser multi-level cell (MLC) flash, it is about 10,000. A flash block that has a worn-out pages, i.e., a worn-out block, cannot be used any further. Too many worn-out blocks will lead to the complete failure of a flash device [19].

Wear leveling [20, 10, 2, 14, 19] has been devised to target the write endurance of NAND flash. Generally, wear leveling attempts to intelligently put data in suitable blocks to avoid the skewness of erase operations. To do so, it first needs to classify data and blocks, respectively. *Hot* data are ones that are frequently updated, while data that are never or seldom rewritten are deemed to be *cold*. On the other side, the erase count of each flash block is maintained in a *block aging table* [14]. A block that has a smaller erase count is *younger* than one that has a larger erase count. Wear leveling tries to put hot data into younger blocks, and cold data into elder blocks. It is expected that updates of hot data can result in many erasures, while blocks occupied by cold data can avoid being frequently erased. However, the correct identification of hot or cold data, and how to site them in a block of a suitable age are two challenging issues.

Wear leveling, and other modules of flash management, are included in a firmware of flash devices called the *flash translation layer* (FTL) [8, 12, 20, 10, 7]. The FTL is designed to be self-contained. The host operating system (OS) communicates with the flash device through interfaces like USB or SATA, and is generally oblivious to the management of flash memory. The OS sends requests to the FTL, and waits for replies in a client-server way, treating the flash device as a black box. File system-aware FTLs [9, 8, 12] have been proposed, but they focus only on deleted data, metadata or critical data. Our paper, however, will exploit OS's knowledge of files for wear leveling in a collaborative way.

Data from the OS have to be assessed for the purpose of wear leveling. Traditionally, the FTL estimates data by itself, and allocates pages and blocks accordingly [20, 1]. The estimation is achieved either through recording data's update frequencies [5, 17], or with the aid of address translation scheme in use [20]. Nonetheless, the FTL's classification of data within a flash device is arduous and costly on time and space. Worse, at that level, the FTL has no idea of the intended use of data. On the other hand, the OS is aware of the file type data belong to, and what applications are using them. The scheme proposed in this paper, namely *operating **S**ystem-**A**ssisted **W**ear leveling* (SAW), will leverage on the OS knowledge of the files being accessed so that the FTL can do a better job in wear leveling. The key ideas of SAW, also the main contributions of this paper, are as follows:

- Data of files are *quantitatively* analyzed and classified using a measure of the *temperature*. The temperatures

DAC '13, May 29 - June 07 2013, Austin, TX, USA.

of files of various types are periodically updated by the OS, which is based on a succinct and reasonable mathematical model. The computed temperature is sent along with data to the FTL.

- Flash blocks that are available for use, i.e., *free blocks*, are organized into groups using an *exponential division*. The FTL interprets the temperature of data, and performs allocation from a relevant group accordingly.
- A prototype based on the Linux virtual file system (VFS), an open-source file system and its special FTL, i.e., UBIFS and UBI [13], has been developed to verify the effectiveness of SAW.

UBIFS and UBI are specifically designed for raw flash that is found in embedded systems like smartphones. With the above prototype, experiments show that the wear evenness can be significantly improved by as much as 85.0% compared to the state-of-the-art FTL-based wear leveling schemes.

The rest of this paper is organized as follows. Section 2 shows background and related works. Section 3 presents the details of SAW. Section 4 describes the experimental evaluation. Section 5 is our acknowledgement and Section 6 will conclude the paper.

2. BACKGROUND AND RELATED WORKS

The FTL is a firmware that autonomously manages flash device. It processes access requests from upper-level file system by translating logical address to physical address into the form of flash block and page. It also conducts wear leveling, as well as garbage collection for resource reclamation. The latter is used to reclaim pages and blocks that are occupied by obsolete data owing to out-of-place updating.

Wear leveling is an important module of the FTL. It may either be *static* or *dynamic* [10, 14]. Dynamic wear leveling generally selects the youngest free block for new data, while static wear leveling may vacate the block currently occupied by cold data for use. The latest FTL wear leveling schemes include dual-pool algorithm [1], BET [10], lazy wear leveling [2] and OWL [20]. We shall focus on BET and OWL as they represent two different strategies. BET takes the perspective of the flash block. It has a *Block Erasing Table* (BET) to maintain the erase status of each flash block in every fixed interval. If the count of erasures over the number of erased blocks exceeds a predefined threshold, BET will repeatedly pick un-erased blocks of the last interval, and perform data transfers, after which it will erase them until the skewness is smoothed out. OWL, however, emphasizes on data. OWL also has a table, namely the *Block Access Table* (BAT), in which a block is a logical block instead of a flash block. The BAT stores access frequencies of logical blocks that have been recently rewritten. OWL ranks data of logical blocks with the BAT, and allocates flash blocks accordingly. The ranking is used to predict a logical block's access frequency compared to other logical blocks in the near future. Doing so OWL attempts to put data into suitable blocks in a proactive way to avoid wear unevenness.

Note that both BET and BAT are data structures maintained by the FTL inside flash device. There are non-trivial spatial and temporal overheads in doing so.

There are FTLs that were devised to take file system into account. MFTL [8] interposes a filter between file system and the FTL to separate metadata and real data of files. It specifically manages metadata that are small and frequently updated. MFTL was implemented for ext2 and ext3 file systems, and performance improvement was reported. FSAF [9] focuses only on deleted data in FAT32. It is similar to the TRIM command of modern OS [3]. FSAF detects the deletion by utilizing its knowledge about the format of FAT32 in flash devices. Meta-Cure [12] is similar to MFTL. It adds a filter between file system and FTL to enhance the reliability of "critical data" to avoid being damaged. Meta-Cure does not change the file system; it is transparent to the FTL. In all these works, either the OS is unaware of FTL's workings, or vice-versa.

3. SAW

In this section we will present details of how the OS collaborates with the FTL for wear leveling in SAW. The OS manages files for applications. It segments or assembles data of files to satisfy applications' access requests. Thus, the OS knows which file a data segment belongs to, and which application is requesting it. Such information is invisible to the lower-level FTL. Traditionally, data are either coarsely identified to be hot or cold, or arduously classified by the FTL within the limited computation resource of a flash device. It is where SAW is to make a difference. In SAW, the OS is responsible for *quantitatively* classifying files based on a mathematical model. For a file type, the OS detects its files' updates, and generates a *temperature* that is sent along with each data segment to the FTL. When a data segment arrives in the flash device, the FTL extracts the temperature information, and processes the allocation request accordingly.

3.1 File Type Temperatures

Files have attributes, such as filename, extension, access mode, and last modified time. Mesnier et al. [6] revealed that files' properties, like access pattern, can be predicted based on their attributes. Take a text file for example. It is likely to be rewritten more often than a video file. Mesnier et. al. [6] did an offline mining over collected files. Online exploration of the files' attributes, however, is not straightforward. We need a succinct and reasonable mathematical model for doing so.

For simplification, SAW only considers two attributes of a file, namely its filename extension and access mode. Read-only files are hardly rewritten, and will be specially dealt with. A rewritable file will be assigned a temperature degree according to it *type*. Here a file's type refers to its filename extension, although it is conceivable that other attributes can be used too. Files without any extension will be treated separately. Previous qualitative ways to identify data to be hot or cold are somewhat lacking. With the assistance of the OS, we will perform the classification in a quantitative way. The temperature of a file type, as will be derived below, depends on files' update *frequency* and *recency*.

3.1.1 Update Frequency of A File Type

Measuring the update frequency of a file type is the key issue of SAW. FTLs can record the number of writes to a logical block or a logical page [20]. For files, however, it is not so simple. The OS manages a large number of files of the same type. It is neither reasonable nor scalable to keep access information for each file. Moreover, two files with the same type may have completely different update frequencies. Hence, we need an approximation to represent

the access frequency for a **type** of files. Since not all files are accessed at runtime, we will not consider *dormant* files but focus only on *active* ones. This simplifies the online analysis, and also reduces the overhead of resuming SAW at boot-up.

SAW maintains several variables for a file type. Given a file type t, $S_{(t)}$ records the total number of active files of type t. $\varsigma_{(t)}$ is the number of accessed files of type t. This includes the files of type t that have been opened (and possibly then closed) after the current system boot, as well as newly created files. $\delta_{(t)}$ is the number of files of type t that have been deleted (since the last boot). $\omega_{(t)}$ counts the *rewrites* to all t files. $\varsigma_{(t)}$, $\delta_{(t)}$ and $\omega_{(t)}$ are used to compute the update frequencies of file type t. Note that we are interested in rewrites detected in the kernel module of file system, not writes, because the latter is not a good estimate for update frequency. For example, a video file triggers a vast number of writes during its creation. Afterwards its contents are hardly rewritten again. So a video file's update frequency is low. When a text file is reopened, however, it may be inserted, appended or replaced with new data. Thus, its update frequency is much higher due to many rewrites.

Let $\varphi_{(t)}$, the *update frequency* of type t, be defined as

$$\varphi_{(t)} = \frac{\omega_{(t)}}{S_{(t)}}. \tag{1}$$

At the first sight, $\varphi_{(t)}$ seems to be the average rewrite of active files of type t. Nonetheless, as is mentioned, files of the same type may differ significantly in update behavior. Moreover, files are being created and deleted at runtime. So Equation (1) is imprecise. But it is infeasible to keep too much information for each file. We shall place more constraints to enhance the accuracy of Equation (1).

First, the OS will collect the values of ς, δ and ω **periodically**. The interval is defined as I. The total number of active files of type t after the nth I is to be $S^n_{(t)}$. The base case, i.e., at boot-up, is defined as $S^0_{(t)}$, and initialized to be zero. In the nth I interval, $\varsigma^n_{(t)}$ files were newly accessed or created, and $\delta^n_{(t)}$ files were removed. So the number of type t files before the start of the $(n+1)$th I is

$$S^{n+1}_{(t)} = S^n_{(t)} + (\varsigma^n_{(t)} - \delta^n_{(t)}). \tag{2}$$

Hence, the absolute increment of type t files is

$$S^{n+1}_{(t)} - S^n_{(t)} = \varsigma^n_{(t)} - \delta^n_{(t)}. \tag{3}$$

The rate of increase of type t files, $s^n_{(t)}$, of the nth I, is

$$s^n_{(t)} = \frac{\varsigma^n_{(t)} - \delta^n_{(t)}}{S^n_{(t)}}, \tag{4}$$

where $n \geq 1$ because at boot-up $S^0_{(t)} = 0$, and it is in the first I that files are accessed or created.

$s^n_{(t)}$ could be positive or negative, as the number of type t files may increase or decrease. β is a bound such that

$$\text{-}\beta \leq s^n_{(t)} \leq \beta, \tag{5}$$

or put in another way,

$$S^{n+1}_{(t)} = S^n_{(t)} \cdot (1 \pm \beta), \tag{6}$$

and Equation (1) is hence valid for the calculation of temperature. In this paper, β is set to be 10%. We do not expect the number of active files of type t changes sharply. If $|s^{n+1}_{(t)}| > \beta$, we will identify t to be an *outlier*. An outlier deserves special attention since many t files are likely to be created or removed in a short period of time.

3.1.2 Update Recency

After the nth I, Equation (1) can be rewritten as

$$\varphi^n_{(t)} = \frac{\omega^n_{(t)}}{S^n_{(t)}}. \tag{7}$$

Equation (7) gives the rewrite frequency on a file type t, and it estimates the update behavior of type t files in the $(n+1)$th interval. However, $S^n_{(t)}$ accumulates the number of active files during past n intervals. As time goes by, the updates of type t may change a lot due to the context switch of applications. Hence a value from a long time ago may mislead the estimation. Generally, the most recent intervals are more relevant to the coming interval, and this *recency* should be factored into Equation (7).

We introduce another variable to improve Equation (7), $f^n_{(t)}$, which is defined to be the predicted value for $\varphi^n_{(t)}$ of the nth interval. Next, we define an *exponentially average* value of $f^{n+1}_{(t)}$ for the $(n+1)$th I as

$$f^{n+1}_{(t)} = \alpha \cdot \varphi^n_{(t)} + (1-\alpha) \cdot f^n_{(t)}, \tag{8}$$

in which $0 \leq \alpha \leq 1$. When $\alpha = 0$, the recent interval will have no effect. With $\alpha = 1$, the past history is assumed to have no influence. Given an α that $0 < \alpha < 1$, we have

$$\begin{aligned} f^{n+1}_{(t)} &= \alpha \cdot \varphi^n_{(t)} + (1-\alpha) \cdot f^n_{(t)} \\ &= \alpha \cdot \varphi^n_{(t)} + (1-\alpha) \cdot \left[\alpha \cdot \varphi^{n-1}_{(t)} + (1-\alpha) \cdot f^{n-1}_{(t)}\right] \\ &= ... \\ &= \alpha \cdot \varphi^n_{(t)} + (1-\alpha) \cdot \alpha \cdot \varphi^{n-1}_{(t)} + ... + (1-\alpha)^i \cdot \alpha \cdot \varphi^{n-i}_{(t)} \\ &\quad + (1-\alpha)^{(i+1)} \cdot \alpha \cdot \varphi^{n-(i+1)}_{(t)} + ... + (1-\alpha)^{n+1} \cdot f^0_{(t)}. \end{aligned}$$

Because

$$\alpha > (1-\alpha) \cdot \alpha > ... > (1-\alpha)^i \cdot \alpha > ... > (1-\alpha)^n \cdot \alpha, \tag{9}$$

we can conclude for $f^{n+1}_{(t)}$, the farther an interval is, the less the effect it has ($f^0_{(t)} = 0$ and $\varphi^n_{(t)} = 0$, so the last $(1-\alpha)^{n+1}$ is ignorant). In other words, $f^{n+1}_{(t)}$ depends the most on $\varphi^n_{(t)}$, and also takes the past history into consideration when $0 < \alpha < 1$. Now we can use $f^{n+1}_{(t)}$ to predict the future update behavior to files of type t in the $(n+1)$th interval.

3.1.3 Temperature of File Types

Now that we have f^n for all file types, we can compute their temperature before each interval. The temperature used in this paper is from 0 to T. T is a predefined constant. A file with the zero degree is very cold, effectively like a read-only file. If a file's temperature is near to T, it is very hot. Given a set of file types, each one with an f^{n+1} for the $(n+1)$th interval, we sort them by their f values in an ascending order. The type t then has a *position number* in the sequence, $P^{n+1}_{(t)}$, where $0 \leq P^{n+1}_{(t)} \leq \Theta - 1$. Θ is the number of active file types. For example, there are five file types (i.e., $\Theta = 5$), and the sorting sequence is

$$f^{n+1}_{(t_0)} \leq f^{n+1}_{(t_1)} \leq f^{n+1}_{(t_2)} \leq f^{n+1}_{(t_3)} \leq f^{n+1}_{(t_4)}.$$

So $P^{n+1}_{(t_0)} = 0$ and $P^{n+1}_{(t_3)} = 3$. Then we can calculate the temperature for type t, $C^{n+1}_{(t)}$, using

$$C^{n+1}_{(t)} = \frac{P^{n+1}_{(t)}}{\Theta} \cdot T. \tag{10}$$

If T is set to be 5, $C^{n+1}_{(t_3)} = 3$ for type t_3. Note that Equation (10) is valid when $n \geq 1$. The temperature of each type t for the first interval, i.e., $C^1_{(t)}$, is initialized to be zero.

It may seem tedious to have to perform a sort over Θ file types after each interval. However, since the access behaviors of the majority of file types are stable, a complete re-sorting is not yet necessary. Instead, SAW scans the previous sequence with updated f values, performing the necessary reordering. This is fairly inexpensive.

According to Equation (10), it is not possible for a file to have a temperature of T, as T is reserved for outlier files.

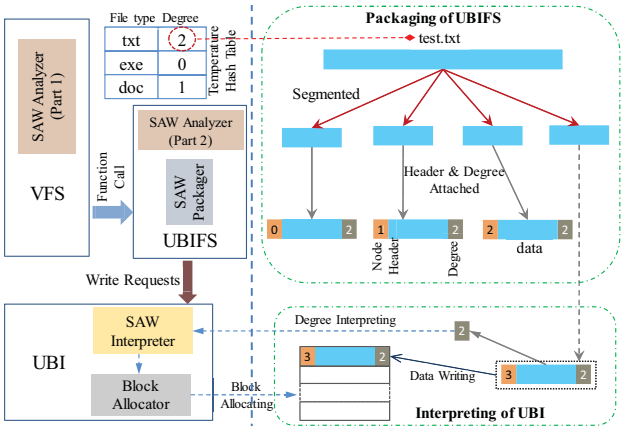

Figure 1: A Sketch of SAW Prototype

3.2 Wear Leveling with Temperature

3.2.1 *Exponential Division of Flash Blocks*

We use the temperature of a file to allocate blocks and pages to its data. First, we need a hash table to maintain the temperatures for file types. The hash key is a file type t that is hashed to its $f^n_{(t)}$ for the nth interval. This table is managed by the OS in main memory, not in flash devices.

The basic idea of wear leveling is to allocate young blocks to hot data, and old blocks to cold data. To make use of the temperature, free blocks in flash device should be well organized. SAW sorts them in an ascending order by their erase counts. As we have T degrees, all free blocks are divided into T groups. The division is not equal but in an *exponential* way. Assuming there are Γ sorted free blocks, the first group has $\Gamma/2$ blocks that have the smallest erase counts. The second group has $\Gamma/2^2$ blocks. By analogy, the gth group has $\Gamma/2^g$ $(0 \leq g \leq T-1)$ blocks. The Tth group, however, is an exceptional one that keeps $\Gamma/2^{(T-1)}$ blocks that are the most worn at that time. This is because

$$\Gamma = (\frac{\Gamma}{2^1} + \frac{\Gamma}{2^2} + \frac{\Gamma}{2^3} + ... + \frac{\Gamma}{2^g} + ... + \frac{\Gamma}{2^{(T-1)}}) + \frac{\Gamma}{2^{(T-1)}}. \quad (11)$$

In SAW, an allocation request with a temperature of d is satisfied by the $(T-d)$th block group. Whether to allocate a page or a block depends on the FTL's allocation policy. As is mentioned, SAW specially treats read-only and outlier files. The former corresponds to the Tth group, and the latter will be handled with pages and blocks from the first group. There are usually not that many read-only files, so $\Gamma/2^{T-1}$ blocks should be sufficient. Outlier files are quite active. They are accommodated into the youngest $\Gamma/2$ blocks.

The exponential division is due to our intention to make the best use of young blocks that are the least worn. SAW maintains more blocks to the higher temperatures. Ones with the smallest erase counts are given more chance to be utilized, while elder block can avoid being frequently picked.

3.2.2 *Temperature Adjustment*

The temperature is re-calculated in every interval. Hence, cold data would lag behind with outdated temperature since they are infrequently updated. Their temperatures should be adjusted. To look for such cold data is not easy. SAW will not do it by itself. As is mentioned, there is a module called garbage collection in flash management to clean up obsolete data that are generated due to out-of-place updating. Cold data are left with them. SAW works alongside when garbage collection are being conducted. At this time, SAW checks data to be moved and changes their temperature. They are written back by garbage collection with updated temperature then. In this way, the overhead is minimized.

3.3 A Prototype of SAW

We have developed a prototype of SAW based on UBIFS and UBI [13]. Generally, there are two types of flash device. One is found in solid-state drives (SSDs), SD cards and USB thumb drives. On equipment such as smartphones, raw flash may be used. UBIFS is designed for the latter, and UBI can be viewed as its special FTL. We chose UBIFS and UBI to implement SAW because they are open-source.

UBIFS is a log-structure file system. UBI serves UBIFS to access data and performs functionalities of flash management. Several features of UBI and UBIFS facilitate the implementation of SAW. First, UBIFS roughly classifies data to be LONGTERM, SHORTTERM and UNKNOWN. For example, all files' data are hot, i.e., SHORTTERM. Second, data are encapsulated by UBIFS in a *node* with information like the inode number that they belong to [13]. Note that the coarse identification of data is not embedded into nodes. Though, the node structure makes it possible to add our temperature degree into each node. Third, their original wear leveling and garbage collection are not complicated and can be easily replaced or enhanced.

The prototype of SAW has three components, as is shown in Figure 1. The *SAW analyzer* is implemented in the Linux VFS and UBIFS. It maintains the hash table and performs SAW calculations. The *SAW packager* is in UBIFS. It packages data along with relevant temperature into a node. The *SAW interpreter* of UBI supports block allocations using the temperature. Figure 1 also gives a sample on text file "test.txt". The temperature degree of "txt" is 2. The file is segmented into four parts, each packed with the temperature. When a node arrives in UBI, SAW interpreter will suggest to the allocator what would be a suitable age for the block to be allocated. The temperate would be written to flash along with data. Note that in real implementation the temperate is in the header. Here we separate it out for ease of discussion. For the same reason, the writing sequence of the nodes does not adhere strictly to their header numbers.

4. EXPERIMENTAL EVALUATION

The evaluation of SAW was done in two ways. The first is within the above prototype. We compiled the Linux kernel 3.1.6 in Ubuntu 12.04.1. A flash device of 1GB was simulated using the nandsim simulator of Linux kernel. BET was implemented for comparison. The second way we evaluated SAW was with the FlashSim simulator [11], in which we implemented OWL, BET, lazy wear leveling and SAW. The simulated flash was also 1GB. We went on further to enhance BET and lazy wear leveling with the basic idea of SAW on block allocation.

The reason why we did experiments in two ways is that OWL and lazy wear leveling work within hybrid address mapping [20, 2], so they cannot be implemented in UBI. BET does not have such a limitation [10, 20]. The NAND flash in the simulation was configured according to a recent datasheet [15]. The wear evenness is measured using the average erase count and its standard deviation over all flash blocks [20, 19]. For similar average erase counts, the smaller the standard deviation is, the better the wear evenness is.

We did not find any file system benchmarks that target the write endurance of flash memory. What we want are ones that operate on a large number of files and generate sufficient write requests. We examined the analysis of Traeger et al. [4] on various benchmarks, and selected two macro-benchmarks: `postmark` [16] and `filebench` [18]. `Postmark` is single-thread, while `filebench` can be multi-thread. However, they both name file in sequential numbers without any extension. We modified them in order to append a suffix to each file in the form of ".ϵ". ϵ is a lower-case English letter from 'a' to 'z' randomly picked for a file.

The parameters of SAW are set as follows. $T = 10$ and $\alpha = 0.5$. I is relatively measured in terms of write requests. Its default length is 10,000 write requests.

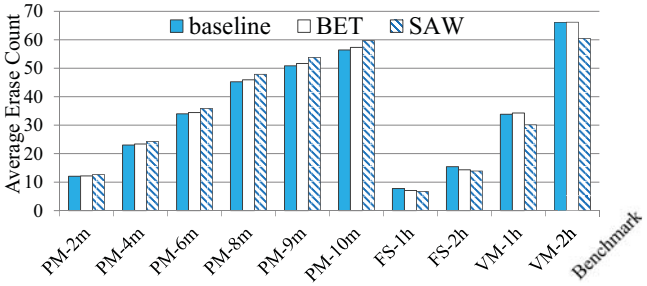

Figure 2: Average Erase Count with Prototype

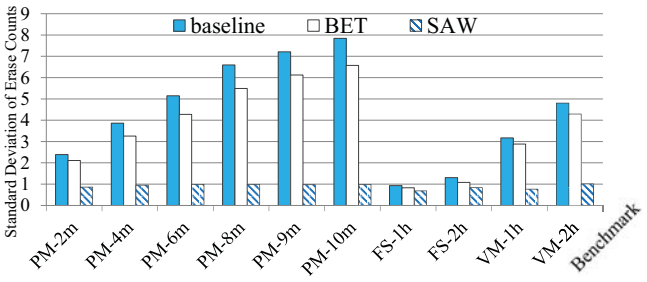

Figure 3: Standard Deviation of Erase Counts with Prototype

4.1 The Effectiveness of SAW

Figure 2 and 3 show the average erase count and standard deviation of `baseline`, `BET` and `SAW` with the prototype. `baseline` has the original wear leveling of UBIFS and UBI. `BET` and `SAW` refer to implementations of BET and SAW, respectively. We ran `postmark` with ten settings, from 1 million to 10 million transactions. The number of simultaneous files was 50,000. Because of space limitation, we present the results of 2, 4, 6, 8, 9 and 10 million transactions, and they are referred to as PM-2m, PM-4m and so on. We ran `filebench` with two public workloads: `fileserver` and `varmail`. For each workload we ran for an hour and two hours, respectively. They are referred to as FS-1h, FS-2h, VM-1h and VM-2h. The number of files was also set to be 50,000. As both `postmark` and `filebench` have random behaviors at runtime [16, 4], we ran our experiments with each setting thrice, and the results shown in Figure 2 and 3 are the mean values. Full results are presented in the Appendix.

The effectiveness of SAW is evident. From Figure 2 we can see that in each case `SAW` performed a similar number of erasures compared to `baseline` and `BET`. However, in Figure 3, `SAW`'s standard deviation of erase counts significantly decreases compared to `BET`, as much as 85.0% with PM-10m. Even with FS-1h and FS-2h that are read-dominant workloads, the reductions can reach 17.3% and 22.8% compared to `BET`, respectively. Hence we conclude that SAW effectively avoids wear skewness with the cooperation of the OS.

Measuring the performance overheads is not straightforward with the involvement of the OS. Moreover, the changing behaviors of `postmark` and `filebench` during each run make direct comparison difficult. We have recorded counts of write, read and erase operations for each case as indicators of the performance. They can be found in the Appendix.

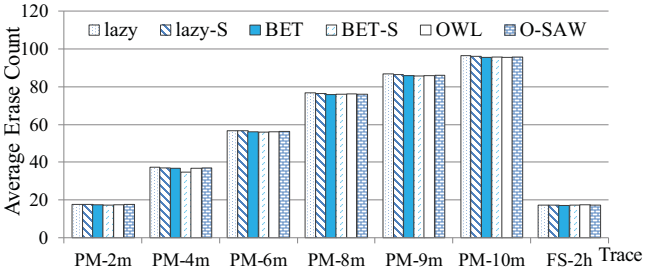

Figure 4: Average Erase Count with FlashSim

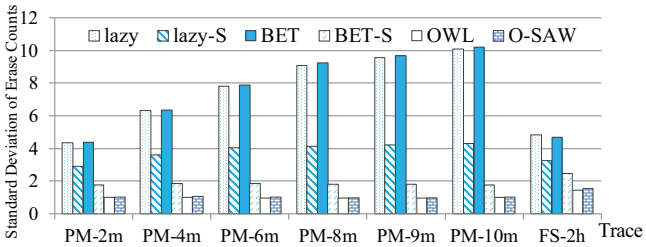

Figure 5: Standard Deviation of Erase Counts with FlashSim

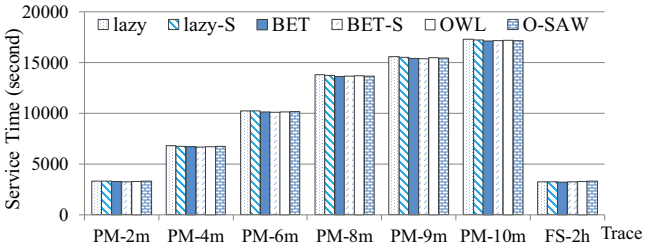

Figure 6: Service time with FlashSim

OWL, lazy wear leveling, and BET were implemented in FlashSim. The latter two were enhanced with SAW's idea in their block allocation. Their implementation are referred to as `OWL`, `lazy`, `BET`, `lazy-S` and `BET-S`, respectively. OWL has already considered block allocation. We did not enhance it. Instead, we replaced OWL's block allocation with SAW's. This implementation is referred to as `O-SAW`.

Note that FlashSim is a trace-driven simulator. Previous experiments on FlashSim utilized traces collected from various machines. However, since write requests of those traces have no temperature information, they are not suitable. Instead, we recorded access request in UBI. There, each request does have a temperature. These traces were then fed

to FlashSim. Experimental results are partially shown in Figure 4, 5 and 6 due to the space limitation.

Figure 4 and 5 show that the average erase counts on a trace for each scheme is similar, but the standard deviation has decreased significantly for `lazy-S` and `BET-S`, by as much as 55.9% and 82.6%, respectively. Thus, wear evenness was highly improved in the presence of SAW. On the other side, `O-SAW` has comparable wear evenness to `OWL`, and the standard deviation of the former is at most 7.0% more. But `OWL` allocates blocks according to its own calculation utilizing the lower computation capability of a flash device, while `O-SAW` just needs to use the temperature of each incoming request. Hence, `O-SAW` has a much lower computation and resource overhead, while achieving a similar level of wear evenness.

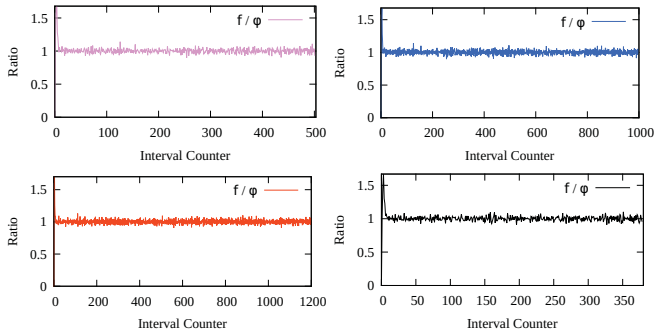

Figure 7: Fluctuation of f/φ (Clockwise: PM-5m, PM-10m, FS-2h, VM-2h)

The performance overhead can be measured using trace driven simulation because it is entirely deterministic. The time needed to service all requests of a trace is a good indicator of the performance overhead incurred by wear leveling [20, 19]. The more the service time, the greater the performance degradation. Figure 6 shows the service time for traces with each scheme. It is obvious that the addition of SAW has little performance impact.

4.2 The Accuracy of f for φ

f is used to predict φ for the next interval using Equation (8), which is the basis of the temperature calculation. We ran experiments to verify the accuracy of f for φ. Without loss of generality, we selected the file type whose filename extension is "*.c*". We collected f and φ in every interval with PM-5m, PM-10m, FS-2h and VM-2h, and calculated f/φ, as is shown in Figure 7. We can see after system boot-up, f/φ fluctuates within tight bounds around 1.0. Hence, we conclude that the prediction of f for φ is accurate.

4.3 Impact of Interval Length

I is an important parameter of SAW. Its default length is 10,000 write requests. We also experimented with lengths of 5,000, 15,000, 20,000 and 25,000. They are referred to as `5k`, `10k`, `15k`, `20k` and `25k`, respectively. Because of space limitation, we could only show their standard deviation in each case in Table 1. There are the mean values over five intervals, as well as the absolute mean differences between the value of each I and the mean. From Table 1 we can see the fluctuation caused by changes of I is insignificant.

5. ACKNOWLEDGEMENT

This paper is supported by the Ministry of Education of Singapore under the grant MOE2010-T2-1-075. We also thank Sudipta Chattopadhyay of NUS for his valuable help.

6. CONCLUSION

In this paper, we revisit the write endurance issue of flash device, and propose a novel scheme named *operating **S**ystem-**A**ssisted **W**ear leveling* (SAW). In SAW, the OS participates in the process of wear leveling by exploiting its higher level view of files. Using our proposed model, the OS quantitatively estimates the temperature of files, and sends this information to the lower-level FTL that in turn uses it for block allocation. We have developed a prototype based on an open-source flash file system and FTL. Experiments show that the collaboration between the OS and the FTL improves the wear evenness by as much as 85.0% compared to the latest FTL-based wear leveling algorithms.

Table 1: Mean Difference of Standard Deviation with Five Intervals (I)

Benchmark	Mean	Absolute Mean Difference				
		5k	10k	15k	20k	25k
PM_2m	0.865	0.009	0.001	0.001	0.002	0.013
PM_4m	0.960	0.018	0.004	0.014	0.007	0.007
PM_6m	0.981	0.019	0.002	0.004	0.017	0.004
PM_8m	0.986	0.007	0.025	0.002	0.002	0.018
PM_9m	0.982	0.015	0.012	0.002	0.003	0.022
PM_10m	0.985	0.007	0.011	0.023	0.014	0.009
FS-1h	0.692	0.017	0.001	0.024	0.004	0.007
FS-2h	0.861	0.047	0.009	0.016	0.029	0.007
VM-1h	0.789	0.012	0.004	0.011	0.002	0.003
VM-2h	1.036	0.004	0.021	0.017	0.026	0.026

7. REFERENCES

[1] L.-P. Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *SAC '07*.
[2] L.-P. Chang and L.-C. Huang. A low-cost wear-leveling algorithm for block-mapping solid-state disks. In *LCTES '11*.
[3] Intel Corporation. What are the advantages of TRIM and how can I use it with my SSD? http://www.intel.com/support/ssdc/hpssd/sb/CS-031846.htm.
[4] A. Traeger et al. A nine year study of file system and storage benchmarking. *Trans. Storage*, 4(2):5:1–5:56, May 2008.
[5] J.-W. Hsieh et al. Efficient identification of hot data for flash memory storage systems. *Trans. Storage*, 2(1):22–40, Feb. 2006.
[6] M. Mesnier et al. File classification in self-* storage systems. In *ICAC '04*.
[7] P.-C. Huang et al. Joint management of RAM and flash memory with access pattern considerations. In *DAC '12*.
[8] P.-L. Wu et al. A file-system-aware FTL design for flash-memory storage systems. In *DATE '09*.
[9] S. K. Mylavarapu et al. FSAF: file system aware flash translation layer for NAND flash memories. In *DATE '09*.
[10] Y.-H. Chang et al. Endurance enhancement of flash-memory storage systems: an efficient static wear leveling design. In *DAC '07*.
[11] Y. Kim et al. FlashSim: A simulator for NAND flash-based solid-state drives. In *SIMUL '09*.
[12] Y. Wang et al. Meta-Cure: a reliability enhancement strategy for metadata in NAND flash memory storage systems. In *DAC '12*.
[13] A. Hunter. A brief introduction to the design of UBIFS, 2008.
[14] Micron Technology Inc. TN-26-61: Wear-leveling in Micron® NAND flash memory. Technical report, Oct 2011.
[15] Micron Technology, Inc. NAND flash memory datasheet (MT29F16G08AJADAWP), Feburary 2012.
[16] J. Katcher. Postmark: A new file system benchmark. Technical Report TR3022, Oct. 1997.
[17] D. Park and D. H.-C. Du. Hot data identification for flash-based storage systems using multiple bloom filters. In *MSST '11*.
[18] File system and Storage Lab. Filebench benchmark, 2011. http://sourceforge.net/projects/filebench/.
[19] C. Wang and W.-F. Wong. Extending the lifetime of NAND flash memory by salvaging bad blocks. In *DATE '12*.
[20] C. Wang and W.-F. Wong. Observational wear leveling: an efficient algorithm for flash memory management. In *DAC '12*.

APPENDIX

A. MORE EXPERIMENTAL RESULTS

A.1 The Impact of β

It was mentioned in Section 3.1 that we use β as a bound for identifying whether a file type is an outlier or not. Without loss of generality, we collected the s value in every interval for the file type "*.c*" using the experimental configurations PM-5m, PM-10m, FS-2h and VM-2h. The results are presented in Figure A-1. At boot-up, many files are accessed, so s is somewhat large. After the system has warmed up, s fluctuates marginally around 0. Note that β was set to be 10% by default. In summary, experiments show that s is typically much less than the selected threshold.

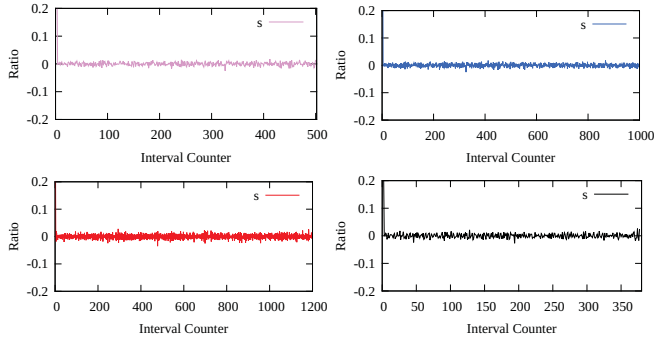

Figure A-1: s and β at runtime (Clockwise: PM-5m, PM-10m, FS-2h, VM-2h)

A.2 Experimental Results with the Prototype

As mentioned in the paper, due to the essentially nondeterministic nature of the operating system, there can be differences between each run of the experiments. Here, we present the full experimental results of `baseline`, `BET` and `SAW` in terms of the average erase count, standard deviation, the counts of write and read operations in Table A-1, A-2 and A-3. Table A-1, A-2 and A-3 show results recorded at each time, respectively. The count of erase operations is not separately listed because it can be computed using the average erase count in each table.

The detailed experimental results of five settings on the interval I are presented in Table A-4, including the average erase count and standard deviation.

A.3 Experimental Results with FlashSim

The average erase count, standard deviation and service time of `lazy`, `lazy-S`, `BET` and `BET-S`, `OWL` and `O-SAW` are shown in three tables for readability. They are Table A-5, A-6 and A-7. The traces of VM-1h and VM-2h were not fed to FlashSim because they are too short. Note that the computation time of `OWL`'s sorts is not included in service time due to FlashSim's limitation. Instead, we recorded the count of sorting for each trace. It is also shown in Table A-7.

The result of the prototype and that of the trace-driven simulation under the same setting may be different, or even vary significantly. Take the average erase count for example. The value of PM-5m using `BET` in Table A-1 is 28.855, but in Table A-6 the average erase count is 46.355. We ascribe this to the different simulators (nandsim vs Flashsim) used. nandsim is within a full-system simulator while Flashsim performs standalone trace-driven simulation.

Table A-1: Average Erase Count, Standard Deviation, the Counts of Write and Read Operations of `baseline`, `BET` and `SAW` (1st Time)

Benchmark		Average Erase Count			Standard Deviation			Write Operations			Read Operations		
		`baseline`	`BET`	`SAW`	`baseline`	`BET`	`SAW`	`baseline`	`BET`	`SAW`	`baseline`	`BET`	`SAW`
Postmark	PM-1m	6.572	6.624	6.842	1.471	1.350	0.791	2,972,904	3,000,917	3,119,305	5,949,162	5,634,356	5,049,137
	PM-2m	12.043	12.196	12.643	2.387	2.115	0.860	5,877,722	5,950,703	6,196,399	8,443,477	9,615,839	9,516,198
	PM-3m	17.512	17.771	18.464	3.167	2.731	0.912	8,779,117	8,896,513	9,281,541	12,810,709	13,366,750	14,236,254
	PM-4m	22.985	23.430	24.276	3.864	3.271	0.957	11,681,921	11,836,146	12,363,103	16,897,790	17,397,386	20,548,770
	PM-5m	28.446	28.855	30.075	4.497	3.798	0.952	14,578,317	14,767,427	15,437,939	21,119,642	22,706,330	23,939,011
	PM-6m	33.956	34.461	35.929	5.186	4.319	0.992	17,500,788	17,737,843	18,544,059	26,133,797	25,403,395	32,996,678
	PM-7m	39.590	40.135	41.868	5.828	4.894	0.976	20,485,814	20,746,099	21,692,520	30,624,022	29,920,760	49,397,386
	PM-8m	45.249	45.916	47.881	6.606	5.509	0.975	23,484,623	23,796,587	24,881,391	35,741,196	34,436,772	55,940,048
	PM-9m	50.798	51.627	53.847	7.207	6.098	0.979	26,429,906	26,812,825	28,042,582	40,370,526	40,111,468	47,798,743
	PM-10m	56.377	57.412	59.750	7.817	6.525	0.991	29,384,145	29,829,525	31,170,566	45,729,523	47,253,426	48,540,694
Filebench	FS-1h	7.628	7.224	6.655	0.915	0.825	0.678	4,035,325	3,788,685	3,518,173	253,787,028	233,511,118	213,650,267
	FS-2h	15.463	14.195	12.767	1.285	1.068	0.817	8,192,460	7,440,788	6,761,944	520,335,165	465,269,296	417,412,195
	VM-1h	34.619	34.574	27.985	3.219	2.892	0.738	18,354,325	18,180,748	14,829,527	161,193,400	164,616,315	135,871,795
	VM-2h	65.281	68.581	55.862	4.747	4.346	0.976	34,619,953	36,225,506	29,611,400	364,102,368	326,448,894	270,803,909

Table A-2: Average Erase Count, Standard Deviation, the Counts of Write and Read Operations of `baseline`, `BET` and `SAW` (2nd Time)

Benchmark		Average Erase Count			Standard Deviation			Write Operations			Read Operations		
		`baseline`	`BET`	`SAW`	`baseline`	`BET`	`SAW`	`baseline`	`BET`	`SAW`	`baseline`	`BET`	`SAW`
Postmark	PM-1m	6.572	6.621	6.839	1.472	1.328	0.777	2,977,457	2,996,229	3,117,289	5,603,129	4,346,563	5,790,084
	PM-2m	12.060	12.173	12.667	2.398	2.094	0.864	5,886,879	5,935,576	6,208,200	8,700,539	8,570,968	9,520,877
	PM-3m	17.476	17.743	18.469	3.126	2.693	0.917	8,791,218	8,880,335	9,284,858	12,817,376	13,057,314	14,646,823
	PM-4m	23.035	23.379	24.270	3.877	3.255	0.942	11,708,459	11,834,954	12,361,087	17,244,826	19,895,452	19,788,998
	PM-5m	28.509	28.853	30.085	4.561	3.752	0.958	14,610,057	14,766,016	15,442,771	21,478,833	26,090,482	23,782,072
	PM-6m	34.014	34.431	35.895	5.097	4.242	1.004	17,528,201	17,721,035	18,525,806	25,659,251	28,238,393	33,947,237
	PM-7m	39.626	40.155	41.861	5.818	4.811	0.961	20,503,432	20,752,174	21,688,147	30,191,413	32,778,028	45,098,977
	PM-8m	45.270	45.907	47.865	6.602	5.470	0.987	23,498,782	23,801,510	24,869,822	34,692,895	36,817,108	59,203,647
	PM-9m	50.900	51.613	53.799	7.272	6.098	0.966	26,478,332	26,802,807	28,014,754	39,780,798	41,270,430	71,700,908
	PM-10m	56.457	57.172	59.724	7.861	6.566	0.989	29,427,218	29,770,773	31,155,659	44,148,945	48,942,244	47,564,490
Filebench	FS-1h	7.731	6.754	6.695	0.941	0.816	0.684	4,091,164	3,553,758	3,538,685	256,187,246	218,886,483	215,527,014
	FS-2h	15.284	14.553	14.135	1.296	1.086	0.837	8,097,963	7,673,349	7,486,036	508,522,787	481,981,880	462,880,636
	VM-1h	33.276	34.123	29.758	3.130	2.833	0.761	17,642,494	18,003,351	15,769,747	156,901,333	160,309,605	147,363,881
	VM-2h	66.426	63.870	61.740	4.873	4.222	1.003	35,227,359	33,746,903	32,729,150	316,867,995	385,058,182	277,678,522

Table A-3: Average Erase Count, Standard Deviation, the Counts of Write and Read Operations of `baseline`, `BET` and `SAW` (3rd Time)

Benchmark		Average Erase Count			Standard Deviation			Write Operations			Read Operations		
		`baseline`	`BET`	`SAW`	`baseline`	`BET`	`SAW`	`baseline`	`BET`	`SAW`	`baseline`	`BET`	`SAW`
Postmark	PM-1m	6.568	6.635	6.828	1.466	1.342	0.782	2,975,544	3,005,347	3,112,347	5,269,322	5,423,570	7,082,480
	PM-2m	12.031	12.201	12.644	2.382	2.115	0.866	5,871,655	5,951,349	6,194,713	8,804,954	10,184,471	9,911,950
	PM-3m	17.504	17.769	18.445	3.165	2.718	0.918	8,773,932	8,890,051	9,272,150	13,180,203	16,262,391	14,982,829
	PM-4m	23.012	23.407	24.278	3.837	3.245	0.964	11,692,882	11,826,157	12,363,732	17,205,703	17,782,500	19,032,648
	PM-5m	28.446	28.845	30.067	4.475	3.776	0.961	14,577,788	14,760,804	15,433,961	21,460,584	21,652,666	23,525,050
	PM-6m	33.976	34.449	35.902	5.163	4.259	0.979	17,509,942	17,729,383	18,528,649	26,179,472	27,685,902	37,342,566
	PM-7m	39.569	40.159	41.872	5.806	4.831	0.975	20,471,142	20,758,460	21,692,750	30,716,852	32,483,543	50,465,286
	PM-8m	45.233	45.946	47.891	6.581	5.484	1.012	23,476,279	23,806,545	24,878,595	36,004,233	40,954,853	61,744,099
	PM-9m	50.820	51.693	53.808	7.152	6.176	0.969	26,437,793	26,828,372	28,014,460	39,524,900	48,308,253	68,480,672
	PM-10m	56.366	57.392	59.686	7.849	6.633	0.974	29,380,836	29,811,485	31,137,180	44,703,237	44,835,663	47,361,632
Filebench	FS-1h	7.935	7.251	6.893	0.942	0.842	0.691	4,200,841	3,825,111	3,645,013	261,152,625	238,413,134	221,023,278
	FS-2h	15.579	14.295	14.859	1.334	1.093	0.852	8,253,135	7,544,699	7,870,924	526,400,710	475,705,393	484,400,972
	VM-1h	33.592	34.077	32.638	3.154	2.920	0.793	17,810,247	17,984,757	17,296,913	159,574,537	160,924,500	141,305,567
	VM-2h	66.478	66.029	63.885	4.794	4.295	1.057	35,254,799	34,859,154	33,866,482	327,086,848	322,980,052	296,867,320

Table A-4: Average Erase Count and Standard Deviation of 5k, 10k, 15k, 20k and 25k

Benchmark		Average Erase Count					Standard Deviation				
		5k	10k	15k	20k	25k	5k	10k	15k	20k	25k
Postmark	PM-1m	6.822	6.828	6.831	6.828	6.832	0.782	0.782	0.772	0.792	0.786
	PM-2m	12.642	12.644	12.644	12.656	12.631	0.873	0.866	0.866	0.867	0.852
	PM-3m	18.448	18.445	18.433	18.442	18.452	0.913	0.918	0.913	0.900	0.921
	PM-4m	24.249	24.278	24.239	24.260	24.237	0.942	0.964	0.973	0.967	0.952
	PM-5m	30.037	30.067	30.058	30.046	30.041	0.989	0.961	0.966	0.963	0.953
	PM-6m	35.877	35.902	35.888	35.891	35.884	1.000	0.979	0.992	0.964	0.978
	PM-7m	41.848	41.872	41.843	41.865	41.823	0.966	0.975	0.992	0.959	0.970
	PM-8m	47.877	47.891	47.884	47.896	47.858	0.980	1.011	0.985	0.988	0.968
	PM-9m	53.797	53.808	53.832	53.825	53.806	0.966	0.969	0.984	0.985	1.004
	PM-10m	59.722	59.686	59.702	59.635	59.687	0.978	0.974	1.009	0.971	0.995
Filebench	FS-1h	7.664	6.893	7.016	7.152	7.378	0.708	0.691	0.668	0.692	0.698
	FS-2h	15.172	14.859	14.173	14.234	14.749	0.908	0.852	0.845	0.832	0.868
	VM-1h	33.422	32.638	32.741	31.086	32.857	0.800	0.793	0.778	0.786	0.785
	VM-2h	63.635	63.885	62.587	63.971	63.940	1.032	1.057	1.019	1.010	1.062

Table A-5: Average Erase Count, Standard Deviation and Service Time of lazy and lazy-S

Trace	Average Erase Count		Standard Deviation		Service Time (second)	
	lazy	lazy-S	lazy	lazy-S	lazy	lazy-S
PM-1m	7.887	7.876	2.952	2.184	1,556.628	1,554.672
PM-2m	17.670	17.622	4.352	2.888	3,294.608	3,286.003
PM-3m	27.436	27.395	4.481	3.364	5,029.318	5,022.122
PM-4m	37.263	37.067	6.310	3.627	6,775.204	6,740.659
PM-5m	47.056	46.739	7.159	3.484	8,513.718	8,457.832
PM-6m	56.904	56.667	7.833	4.027	10,263.168	10,221.351
PM-7m	66.756	66.518	8.458	4.127	12,014.693	11,972.629
PM-8m	76.791	76.521	9.103	4.126	13,795.465	13,747.889
PM-9m	86.744	86.437	9.586	4.229	15,564.870	15,510.796
PM-10m	96.480	96.183	10.089	4.329	17,295.008	17,242.518
FS-1h	8.25	8.236	3.256	2.431	1,619.643	1,616.433
FS-2h	17.415	17.312	4.833	3.258	3,246.716	3,228.453

Table A-6: Average Erase Count, Standard Deviation and Service Time of BET and BET-S

Trace	Average Erase Count		Standard Deviation		Service Time (second)	
	BET	BET-S	BET	BET-S	BET	BET-S
PM-1m	7.759	7.759	2.932	1.519	1,534.515	1,534.153
PM-2m	17.400	17.400	4.379	1.785	3,246.938	3,246.937
PM-3m	27.048	27.048	5.488	13833	4,960.660	4,960.621
PM-4m	36.732	34.732	6.360	1.875	6,679.744	6,679.732
PM-5m	46.355	46.355	7.255	1.873	8,389.552	8,389.545
PM-6m	56.096	56.096	7.901	1.867	10,119.220	10,119.206
PM-7m	65.948	65.948	8.436	1.839	11,870.850	11,870.873
PM-8m	75.953	75.953	9.256	1.825	13,646.346	13,646.327
PM-9m	85.873	85.873	9.672	1.815	15,409.453	15,458.531
PM-10m	95.618	95.618	10.201	1.772	17,141.411	17,141.443
FS-1h	8.139	8.139	3.214	1.908	1,597.788	1,597.665
FS-2h	17.190	17.190	4.687	2.484	3,202.400	3,202.490

Table A-7: Average Erase Count, Standard Deviation and Service Time of OWL and O-SAW

Trace	Average Erase Count		Standard Deviation		Service Time (second)		The Count of
	OWL	O-SAW	OWL	O-SAW	OWL	O-SAW	OWL's Sorts
PM-1m	7.926	7.929	1.017	0.977	1,584.611	1,585.733	34,547
PM-2m	17.569	17.569	0.987	1.026	3,297.739	3,297.769	76,558
PM-3m	27.211	27.217	1.000	1.054	5,010.163	5,012.342	118,522
PM-4m	36.883	36.888	1.014	1.045	6,730.632	6,732.437	160,669
PM-5m	46.511	46.513	1.002	1.053	8,438.900	8,439.666	202,563
PM-6m	56.248	56.252	0.972	1.011	10,168.723	10,169.981	244,955
PM-7m	66.098	66.102	0.942	0.987	11,919.311	11,920.698	287,988
PM-8m	76.103	16.104	0.934	0.980	13,696.460	13,697.167	331,575
PM-9m	86.021	86.022	0.944	0.970	15,458.531	15,458.991	374,842
PM-10m	95.763	95.762	0.993	1.028	17,189.121	17,188.892	417,425
FS-1h	8.331	8.338	1.049	1.148	1,665.058	1,667.493	38,522
FS-2h	17.381	17.390	1.440	1.541	3,276.431	3,279.699	80,686