2012 IEEE Seventh International Conference on Networking, Architecture, and Storage

# Making Garbage Collection Wear Conscious for Flash SSD

Jonathan Tjioe, Andrés Blanco, Tao Xie
Department of Computer Science
San Diego State University, San Diego, USA
jontjioe@gmail.com; blancore@rohan.sdsu.edu;
xie@cs.sdsu.edu

Yiming Ouyang
School of Computer & Information
Hefei University of Technology
Hefei, Anhui, P. R. China
oyymbox@163.com

***Abstract*—Since NAND flash memory has certain inherent limitations like out-of-place updates and a coarse granularity of erase unit, a NAND flash memory based solid state disk needs a garbage collector to erase and recycle previously used blocks from time to time. Garbage collection, however, can significantly decrease flash SSD performance as it incurs high overhead. Various techniques have been proposed to reduce the cost of garbage collection. Unfortunately, most of them only take performance into consideration while ignoring block wear status when choosing victim blocks. As a result, some blocks could be frequently erased and recycled due to garbage collection, and thus, prematurely fail because of a high concentration of erasure cycles. In this research, we propose a wear conscious garbage collection mechanism named WECO (*we*ar *co*nscious). To demonstrate the effectiveness of WECO, we extend a well-known SSD simulator FlashSim so that WECO can be integrated into typical FTLs (flash translation layers) such as DFTL (demand-based FTL) and PM (page mapping). Experimental results show that WECO-DFTL and WECO-PM substantially improve wear-leveling while maintaining a performance similar to the two original FTLs.**

***Keywords-flash memory; solid state disk; garbage collection; wear-leveling; flash translation layer***

## I. INTRODUCTION

NAND flash memory (hereafter, flash memory) has become the standard storage medium for consumer devices such as MP3 players, smart phones, laptops, and digital cameras [2][13]. The cost of flash memory continues to decrease while its capacity keeps increasing [9]. Flash memory has many benefits such as small form factor, low energy consumption, and faster access times. As a result, flash memory based solid state disk (hereafter, flash SSD) is now considered a replacement for hard disk drive (HDD) not only in PCs but also in server domains [9].

However, there are certain limitations of flash SSDs that are not present in HDDs. First, flash SSD has the erase-before-write problem [6][18], which stems from the intrinsic physical nature of flash memory. Erase-before-write requires that an occupied data block (typically 64 or 128 of 4KB memory cells called pages) must be erased before the new data can be written to that block. As a result, to update a piece of data on a page, an out-of-place update method must be employed: first, the new data is written to an erased page, next, the page that contains the old data is invalidated, finally, the virtual-to-physical address mapping table is modified to reflect this change [14]. The out-of-place updates necessitate the need for flash SSD to utilize a garbage collection (GC) mechanism, which reclaims invalid pages within a block by first relocating valid pages in the block to new destinations and then erasing the entire block. The second limitation is the coarser granularity of erase operations. While write and read operations are conducted at the page level, erasure must be performed at the block level. Compared with reading or writing a page, erasing a block takes a much longer time [1]. Lastly, the lifetime of a block is limited to a finite number of erasures, after which the reliability of the block can no longer be guaranteed. Nowadays, flash memory device write-erase-cycles normally range from 100K to 1 million [6]. Since many server-class I/O intensive workloads have heavy localities [23], some blocks of flash memory can prematurely fail due to a high concentration of write cycles. The consequence is that the entire flash memory becomes unreliable after it runs out of its spare blocks. The write-erase-cycle limitation brings about the need for a wear-leveling scheme [7][16], which ensures that all blocks in a flash SSD are worn out evenly in order to prolong the life and reliability of the flash SSD.

To overcome these limitations, flash SSD employs a software component called the flash translation layer (FTL), which serves as a middleware between the operating system and the flash memory [13]. FTL helps flash SSD to emulate a standard block device by exposing only read/write operations to the upper software layers [14]. It performs the virtual-to-physical address translations and hides the erase-before-write characteristics of flash memory [14][18]. The address mapping table is usually stored in a small piece of SRAM. FTL also provides garbage collection and wear-leveling capabilities that are vital to the performance and reliability of flash SSD [13]. Garbage collection and wear-leveling are two separate modules in an FTL. Garbage collection mechanisms consolidate existing occupied data blocks and erase the freed blocks, whereas wear-leveling techniques distribute writes across the full array of memory cells in order to avoid premature cell failures.

Since garbage collection involves time-consuming erase operations plus numerous internal reads and writes, an ongoing GC process stalls incoming user requests until it completes [21]. As a consequence, the performance of flash SSD could be significantly degraded by 20% due to the queuing delay [8]. A variety of GC mechanisms [15][21] have been proposed to minimize the garbage collection overhead. In particular, they have put great efforts on reducing the total amount of copied data from the victim

978-0-7695-4722-0/12 $26.00 © 2012 IEEE
DOI 10.1109/NAS.2012.20

blocks because moving valid data from victim blocks to new blocks takes a large portion of the total execution time of a GC process [15]. The most common way to achieve this goal is to separate data based on their update frequency so that the number of dead blocks (i.e., blocks that have no valid data) and almost-dead blocks (i.e., blocks that have few valid data) can be increased [15]. Recycling dead or almost-dead blocks can substantially reduce overhead.

Despite the major concern of garbage collection is its negative impacts on performance, it could also unfavorably influence wear-leveling. Considerable research has shown that real-world enterprise workloads exhibit temporal locality [12][23], which implies that a group of related blocks may be frequently accessed. The consequence is that these hot blocks repeatedly become dead or almost-dead blocks, and thus, are recycled very frequently by garbage collection. Eventually, these blocks will reach their write-erase-cycle limitation much faster than others, which leads to an unreliable flash SSD. Current GC mechanisms normally ignore wear-awareness when they select victim blocks and leave the job of making wear evenly distributed to wear-leveling schemes. Unfortunately, wear-leveling techniques cannot solve this problem as they are not triggered by GC. We argue that a new GC mechanism that is conscious of wear is much needed for server-class flash SSDs.

In this research, we propose a novel garbage collection mechanism named WECO (*we*ar *co*nscious) to escalate the reliability of flash SSD by optimizing wear-leveling while improving the performance. To illustrate the validity of WECO, we incorporate it into two mainstream FTLs, PM and DFTL, and rename the resulting FTLs to be WECO-PM and WECO-DFTL, respectively. Next, we extend FlashSim [20] to implement the two WECO-powered FTLs into the well-known flash SSD simulator. Finally, we conduct extensive simulations on the extended simulator against four real-world enterprise-scale traces. Comprehensive experimental results convincingly show that WECO-PM and WECO-DFTL dramatically improve wear-leveling while maintaining, and sometimes slightly exceeding, the performance of the original PM and DFTL.

The remainder of this paper is organized as follows. In the next section we discuss the related work and motivation. In Section III, we describe the design and implementation of WECO. Simulation environment will be presented in Section IV. In Section V, we evaluate WECO using four real-world traces. Section VI concludes the paper.

## II. Related Work And Motivation

### A. Flash SSD Background

Flash SSD uses non-volatile NAND flash memory, which enables it to retain data when the power is off [1]. Major components of a flash SSD include flash controller, internal cache, and flash memory as shown in Fig. 1. The flash controller manages the entire flash SSD including error correction, interface with flash memory, and servicing host requests [1]. The internal cache improves performance by utilizing fast-access volatile data storage for read-write buffers and device-specific management data. The flash memory part of a flash SSD consists of one or more packages and each package is composed of one or multiple dies (also called chips). A die contains multiple planes. Each plane has one register that serves as a buffer for I/O operations (see Fig. 1). One plane consists of many blocks and each block is composed of multiple pages. For example, a Samsung 4GB flash memory package has two dies and each die contains four planes (see Fig. 1) [1]. Each plane consists of 2,048 blocks and one 4 KB data register. Each block has 64 pages and each page is 4 KB.

FTL translates the logical page numbers (LPNs) provided by the operating system to the physical page numbers (PPNs) on the flash memory. The address translation typically requires a page-mapping table to be stored in an internal SRAM of a flash SSD. This table contains the LPN-to-PPN mappings. Considerable research has been conducted on developing various FTLs [14][18][22]. Depending on address translation granularity, three major types of FTL schemes have been developed: (1) page-level mapping FTL; (2) block-level mapping FTL; and (3) hybrid FTL [14]. In a page-level FTL (page-mapping or PM), each logical page can be mapped to any physical page in a flash SSD. Although it can efficiently utilize blocks in a flash SSD and achieve the highest performance among all existing FTLs, the size of its mapping table could be too large to be stored in a SRAM-based cache, which is always very expensive [14]. A block-level FTL scheme translates each LPN into a physical flash block number, which results in a much smaller mapping table. However, block-level FTL requires extra operations to serve a request, and thus, degrades the performance of flash SSD [18]. To avoid the drawbacks of the above two extreme mapping schemes, hybrid FTL schemes [22] logically divide all physical flash blocks into two groups: *data blocks* and *log blocks*. The vast majority of physical flash blocks are tagged as data blocks, which are

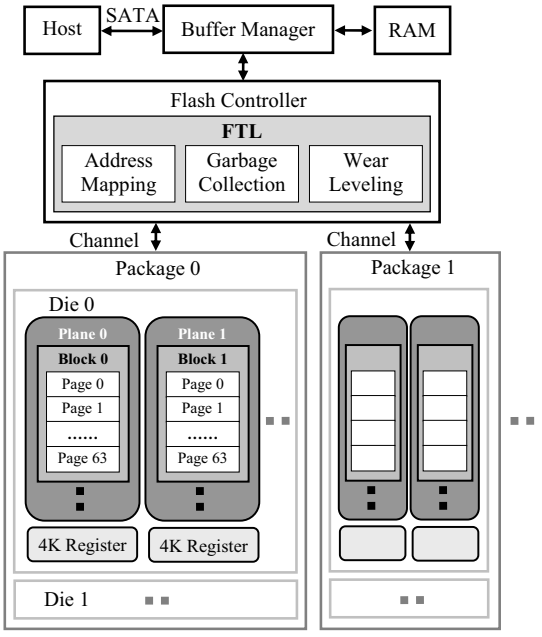


Figure 1. Internal structure of a flash SSD with multiple packages.

administered by a block-mapping scheme. All remaining physical blocks are designated as log blocks, which are page-mapped and invisible to users [18].

Typical hybrid FTL schemes like FAST [22] cannot achieve a performance level comparable to that of page-mapping FTLs due to their inherent log-buffer based mechanisms [14]. Very recently, DFTL [14] and HAT [18], two state-of-the-art FTLs, have been proposed to address the problems associated with hybrid FTLs. Deriving from PM, Aayush *et al.* proposed a demand-based FTL (DFTL) as a page-level FTL that exploits the locality of enterprise workloads by selectively caching page-level address mappings [14]. The performance of DFTL approaches that of PM while leading to a much smaller page-mapping table.

### B. Existing Garbage Collection Techniques

Considerable research has focused on developing efficient GC strategies [2][15][21]. Each of them has to make the following three decisions: (1) when should GC occur? (2) which block should be selected as a victim block? (3) how should GC take place?

A GC algorithm might be triggered once a certain threshold is met. For example, when the percentage of free/erased pages in a flash SSD is less than 5%, a GC process is triggered until the SSD reclaims a certain amount of space [15]. Other triggers could be based on parameters such as the erase count, update count, or page status. If supported by the FTL, when the SSD is idle and not serving any requests, GC can be triggered in the background transparent to the end user. This is known as *passive* GC, and is another method of when GC could occur [13]. One newer GC scheme is able to preempt on-going GC processes to service a pending I/O request, and then resumes the GC process at a later time [21].

Once a GC process is launched, it first needs to select a victim block that will be recycled. The Greedy method performs GC by selecting the block with the most invalid pages. This method offers very high performance as a smaller number of valid pages needs to be copied to new blocks. Nevertheless, it cannot provide good wear-leveling for it does not consider the block erase count. On the other hand, a wear-leveling driven GC algorithm chooses the least worn out blocks as victim blocks, and thus, obviously degrades performance. This is because these least-worn-out blocks normally store valid pages, which implies that the GC overhead could be very high. Apparently, pursuing wear-leveling and improving performance are two conflicting goals for a GC algorithm. To maintain a balance between these two often-conflicting goals, Kim and Lee use a score, which is a weighted sum of block utilization value and wear-leveling index [19]. However, their victim block selection mechanism is specific to flash file systems (FFS) [19]. We extend it to block devices as equation (1) in Section III.

Once a victim block is chosen, any data from valid pages in the victim block has to be copied to new blocks. Instead of merely writing this data to any new block, some GC strategies choose to organize/group this data according to some characteristic such as the data's popularity [2][13][19] so that valid page copying can be performed efficiently.

### C. Motivation

Most existing GC techniques [2][15][21] focus only on performance with little consideration on wear-leveling, which is critical to flash SSD reliability. Besides, wear-leveling algorithms are normally implemented as independent modules inside FTLs, and thus, cannot solve the wear-out-uneven problem caused by GC. Little research has focused on obtaining a balance between performance and wear-leveling in GC. Although touched upon, the method proposed in [19] is dedicated for flash memory based log-structured file systems. How to achieve a good trade-off between performance and wear-leveling in GC for enterprise-class flash SSDs remains an open question.

To make flash SSD GC techniques both performance-centric and wear-conscious, in this research we propose a new GC mechanism called WECO. Two main principles of WECO are: (1) select victim blocks so that wear-leveling can be improved while maintaining performance; and (2) group cold and hot data in order to improve performance.

## III. Design and Implementation

In this section, we provide the design and implementation details of WECO, which can escalate reliability by optimizing wear-leveling and maintain good performance by improving the efficiency of GC. WECO has two major components: (1) a wear-conscious victim block selection policy; (2) an autonomous data separation method.

### A. Wear-Conscious Victim Block Selection Policy

To make a good trade-off between performance and wear-leveling, WECO extends the victim block implementation proposed in [19] to flash SSDs. WECO's wear-conscious victim block selection policy is expressed by equation (1) as below.

$$victimBlockScore = (1-\lambda)\left(\frac{valid(j)}{valid(j)+invalid(j)}\right)+\lambda\left(\frac{erasures(j)}{1+\varepsilon_{\max}}\right), \quad (1)$$

$$where\ \lambda = \frac{2}{1+e^{\frac{k_\varepsilon}{\Delta_\varepsilon}}}\ if \Delta_\varepsilon \neq 0; \lambda = 0 if \Delta_\varepsilon = 0 and \Delta_\varepsilon = \varepsilon_{\max} - \varepsilon_{\min}$$

The physical block that has the lowest score will be selected as the victim block. We can see that (1) has two main terms and two preceding weighting factors. The first term shows the utilization of a block $j$ since it has the number of valid pages of a block $j$ divided by the total number of pages in block $j$. Upon closer investigation, this term is merely the Greedy algorithm, and it emphasizes performance as the only consideration when performing GC. The second term has the erase count of the block $j$ divided by the quantity $(1 + \varepsilon_{max})$ and it emphasizes wear-leveling as the only concern. The preceding coefficients for the first and second terms are $(1-\lambda)$ and $\lambda$, respectively. $\lambda$ is defined as the measurement of wear in the flash and $k_\varepsilon$ is a constant used to define the responsiveness of (1). In other words, if the wear starts increasing, $k_\varepsilon$ will determine how fast it will be corrected. From empirical results, we have set 10 as the default value of $k_\varepsilon$ to provide a moderate "steepness" of response. Since $k_\varepsilon$ decides the value of $\lambda$, which in turn determines the trade-offs made by WECO between

performance and wear-leveling, we investigated the impacts of $k_\varepsilon$ on WECO in Section V. $\Delta_\varepsilon$ is the erasure count difference between the block with the most erasures, $\varepsilon_{max}$, and the block with the least erasures, $\varepsilon_{min}$.

Thus, for (1) when wear in the flash SSD is high, the second term in the equation, which is wear-conscious, weighs more heavily for victim block selection than the first term does. On the other hand, if the wear $\lambda$ is low, then the first term, which is performance-centric, has more weights than the second term does. This allows the WECO strategy to dynamically balance a GC process between performance and wear-leveling depending on the current wear condition of the flash SSD. Clearly, equation (1) makes WECO intelligently adjust around a "sweet spot" between performance and wear-leveling. Unlike existing GC algorithms [13], WECO's victim block selection policy does not require a certain threshold to be met. Instead, during each GC operation, it judiciously adjusts the perfect balance of performance and wear-leveling, resulting in a smoother, less jagged curve of the actual wear condition of the flash SSD.

### *B. Autonomous Data Separation Method*

The second major component of WECO is an autonomous data separation method that groups cold and hot data separately in order to reduce GC overhead and thus to improve GC performance. Many previous studies have shown that real-world enterprise-level workloads exhibit temporal locality [12][23]. Oftentimes, locality is described as the popularity of a file. Since a file is simply data stored on pages, locality can be extended to the finer granularity of the page. Thus, in the context of disk storage systems, locality has to do with a page or a group of related pages being frequently accessed. This is sometimes referred to as the *temperature, heat,* or *popularity* of a page. We define *temporal locality* to be the times that a page or a group of pages is accessed during a period of time [12].

Although grouping hot or cold data together in order to improve performance has also been used by some existing GC algorithms [2][8][17][19], they either need to manually adjust certain tunable parameters [19] or require the knowledge of workload characteristics a priori [8], which largely limits their application in real-world environments. WECO's autonomous data separation method, however, employs the average heat of all current hot pages as the "hot temperature" threshold. In other words, if the heat of a page exceeds this threshold, it will be categorized as a hot page. Otherwise, it is taken as a cold page. In this way WECO can automatically tune the "hot temperature" threshold based on changing workload conditions without any user intervention or any prior knowledge of workload characteristics.

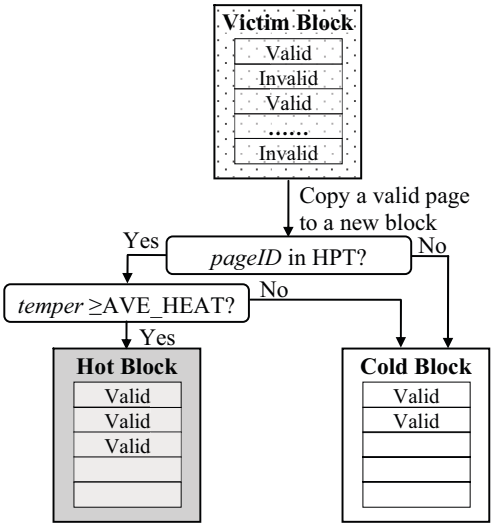


Figure 2. Autonomous data separation method.

WECO takes advantage of the temporal locality of enterprise workloads by grouping hot data and cold data into separate blocks when performing garbage collection. When GC occurs, instead of merely just copying the valid data from the victim block into a new data block, WECO first checks the temperature of the LPN that the PPN's data corresponds to. If the page is determined to be "hot", then WECO moves it to a hot block. Otherwise, the page is determined to be "cold" and it will be moved to a cold block. The general premise is that blocks that contain hot, or frequently updated, data will be invalidated faster than those blocks that contain cold data. Grouping hot pages together will cause the physical blocks that they reside on to be invalidated faster, prompting those blocks to have a greater chance of being selected as a victim block in the future. Once a hot block is selected as a victim block, it will have mostly invalid pages. As a result, less valid data will have to be copied, resulting in a higher efficiency in GC operations. On the other hand, cold data will be grouped together into cold blocks. Data in these blocks will require little to no updating, thus reducing the need to migrate valid cold data to new blocks, thereby reducing the number of GC operations. This will in turn result in an increase in performance. Fig. 2 illustrates the basic idea of WECO's autonomous data separation method.

In order to measure and keep track of the temperature of LPNs, WECO uses a data structure called HPT (hot page table), which has four fields: LPN of the hot page (pageID), update counter (uc), last time stamp (lts), and temperature (temper). Table I shows each field with its size.

TABLE I. THE HPT TABLE

| Fields | Function | Type | Size |
|---|---|---|---|
| pageID | LPN (logical page number) | ulong | 4 bytes |
| uc | Update counter | ulong | 4 bytes |
| lts | Last time stamp | ulong | 4 bytes |
| temper | LPN temperature | ulong | 4 bytes |

The *pageID* is simply a unique number used to keep track of a page. We use the LPN of the page for this field. The *uc* is the number of updates that the corresponding LPN has received. The *lts* is the arrival time of the most recent request corresponding to the *pageID*. The *temper* is the current temperature of the LPN, which is the update count of the LPN. The size of the HPT table in terms of the number of entries is designated by HPT_SIZE. The average temperature

of all the hot pages in HPT is calculated by equation (2)

$$AVE\_HEAT = \frac{\sum_{i=1}^{HPT\_SIZE} uc(i)}{HPT\_SIZE} \quad (2)$$

AVE_HEAT is the average temperature of all of the hot pages (all LPN entries in HPT) and it is used as the threshold for determining whether a page is hot or not. In this way, WECO is able to adapt to changing workloads. For example, if the percentage of writes of a certain workload is moderate, WECO will automatically adjust AVE_HEAT to a lower value, prompting pages with temperatures greater than this moderate value to be moved to a hot block. On the other hand, if a workload exhibits an extremely high percentage of writes, AVE_HEAT will increase, thus prompting only very hot LPNs to be moved to a hot block. For many other FTLs, in order to perform well under different workloads, certain tunable parameters must be adjusted manually and a prior knowledge of workload characteristics is required. However, for WECO, using the average heat from all hot pages in HPT allows this tuning to happen automatically without any user intervention or any knowledge of workload characteristics. In addition, since AVE_HEAT is dynamically calculated, if a workload changes its access pattern over time, e.g., its write request percentage decreases, WECO dynamically adjusts AVE_HEAT with the changing workload. When a valid page is about to be moved from the victim block, WECO first checks if its LPN is contained in HPT (see Fig. 2). If not, it will be copied to a cold block because its LPN has not been written to recently, and thus, does not have the chance to be hot. On the other hand, if its LPN is in HPT and its *temper* is no less than AVE_HEAT, the page is viewed to be "hot" and will be copied to a hot block (Fig. 2).

Fig. 3 explains how WECO populates and maintains the HPT table dynamically. Note that a LPN entry was added to HPT because of an incoming write request to that LPN. Read requests are skipped as they do not generate any update. The HPT table is checked only during GC operations although its contents are populated and updated during the servicing of incoming requests. If the request is a read, nothing needs to be done in HPT and the request is serviced normally. On the other hand, if the request is a write, an entry needs to be either added or updated in HPT. Upon arrival of a write request, WECO checks HPT to see if there is a corresponding entry for the LPN in question. If there is an existing entry, then the arrival time of the request is written to the latest time stamp (*lts*) field and the update counter (*uc*) is incremented by one. If an entry does not exist for the LPN, and there are empty rows in HPT, a new entry will be written in the first empty row. If an entry does not exist, but there is no empty row, then the entry with the oldest value in its *lts* field will be replaced with the entry of current write request. Thus, HPT employs a LRU (least recently used) eviction policy. Once the new entry has been inserted into HPT, its *lts* and *uc* fields will be updated. Finally, the write request will be serviced normally (see Fig. 3).

### C. Overhead of WECO

As with all FTLs, it is important to consider the overhead associated with new capabilities added. With FAST, for example, 3% of the physical flash is dedicated for log blocks [22]. With DFTL [14], 0.2% of the physical flash is reserved for translation pages to contain the complete LPN to PPN mappings. Therefore, for a 64GB flash SSD, DFTL would require about 132MB of the flash memory. For WECO, no extra space on the physical flash needs to be reserved. However, WECO does need to store its HPT table in SRAM. Fortunately, the size of HPT is very small as it is set to 400 records in our implementation. Each record consists of 4 fields, each of which only takes 4 bytes each, yielding a size of 16 Bytes per record. Thus, the total size of SRAM required by WECO is only 6.25KB. Unlike DFTL and FAST, which require a certain percentage of memory, the overhead associated with WECO is fixed and will not increase as the capacity of the flash SSD increases.

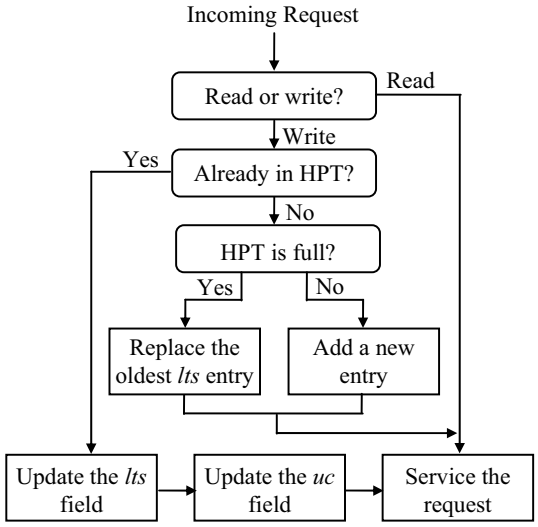


Figure 3. Population of the HPT.

The incoming request monitoring is the main contributor to the extra CPU time consumed by WECO (see Fig. 3). Nevertheless, its runtime overhead is trivial because each of the steps within an update process on the HPT table (see Fig. 3) only takes $O(1)$ to complete. Besides, the size of HPT is very small. Assume that the aggregate write request arrival rate is 100/second. In other words, in every second there are 100 write requests arrive. With a modern 3 GHz processor with 5 cycles per instruction, updating the HPT table 100 times only takes less than 1 millisecond.

## IV. Simulator and Other Contributions

In this section, we present the simulation environment setup and other contributions of this research.

### A. Simulation Environment

We used DiskSim 3.0 [4], a validated simulator for disk storage systems, and a well-known flash SSD simulator FlashSim [20]. DiskSim 3.0 is an efficient and highly configurable hard disk system simulator designed to evaluate various aspects of storage subsystem architecture [4]. It contains modules that allow customizing and simulating of storage architecture components such as

HDD, controllers, buses, etc. However, when DiskSim3.0 was originally created, it did not support the simulation of flash SSDs. As a result, FlashSim [20], a flash memory based SSD simulator, was developed to supplement DiskSim3.0 so that researchers could evaluate the performance of flash SSDs by using the combination of DiskSim3.0 [4] and FlashSim [20]. In this research, we used the DiskSim3.0/FlashSim environment described in [14] as a starting point for our simulations. We installed DiskSim3.0/FlashSim in a virtual machine (VM) running the Ubuntu OS where all our simulations took place.

### B. Other Contributions

In addition to the research contributions provided by this research, we also spent considerable time to write compilation instruction manuals for both DiskSim3.0 [4] and FlashSim [20] as well as trace analysis scripts to help other researchers in flash SSD community. When using the DiskSim 3.0 environment described in [14], we found that the available documentation to install, configure, and run the simulations was inadequate. It took us a couple of weeks to get the environment running. To save other researchers' time, we wrote a step-by-step compilation instruction manual on how to get the simulator working. This document is now published by the DiskSim website [26]. Additionally, existing documentation to get FlashSim running was also found to be inadequate. We authored detailed compilation procedures for FlashSim as well [27].

Using the freely available VirtualBox software provided by Oracle [24], we walk the user through installing a VM with the Ubuntu OS and also how to install and configure both DiskSim 3.0 and FlashSim. Finally, we have provided the virtual machine in the Open Virtualization Format (.ovf), which can be easily imported into any popular virtualization software. We take these efforts that we devoted as other contributions to research community.

### C. Simulator Extension

We found that there were many items that were hard coded in the DiskSim/FlashSim environment. One such example arose with the logical block address (LBA). We modified DiskSim3.0 so that traces that have LBAs out of the boundary of the flash SSD size being simulated could be remapped onto the existing SSD size. This increases the scalability of the simulation environment, while still preserving the original characteristics of the real world trace. We also found that the page size was hard coded to 4 sectors per page. Again, several modifications were made to accommodate varying size of a page in DiskSim3.0 [4].

In order to evaluate the effectiveness of WECO, we extended the DiskSim3.0/FlashSim simulation code so that two WECO-powered FTLs, i.e., WECO-PM and WECO-DFTL, could be implemented. In particular, we revised the simulation code to have several toggles that allow us to easily change the settings for victim block selection, hot/cold blocks, and wear leveling.

## V. PERFORMANCE EVALUATION

### A. Experiment Setup

We use real-world traces to compare the performance of DFTL and PM with WECO-DFTL and WECO-PM, respectively. The four real-world traces used are: Build [5], Exchange [11], Financial1 [25], and Financial2 [25]. These traces were chosen because they are write-intensive. The more erasures or "wear" that a flash SSD experiences, the more contributions of wear-leveling to the reliability of the flash SSD. The Build trace is from a collection of Microsoft Build Server production traces and it spans a period of 25 hours. The Exchange trace is from a collection of production traces collected over a period of 24 hours at Microsoft using the event tracing for Windows framework. The Financial1 and Financial2 traces are I/O traces from OLTP applications running at two large financial institutions [25]. Table II shows the statistics of the real-world traces.

TABLE II. REAL-WORLD TRACE STATISTICS

| Trace Name | Write Percentage | Ave. Size (sector) | Access Rate (reqs/sec.) | Duration (hour) |
|---|---|---|---|---|
| Build | 52.90 | 21 | 319.3 | 0.25 |
| Exchange | 90.80 | 23 | 172.8 | 0.25 |
| Financial1 | 76.83 | 6 | 122 | 12.14 |
| Financial2 | 17.65 | 4 | 90.2 | 11.39 |

Table III illustrates the experimental parameters used. The three metrics that we measured during simulations are:

- *Mean Response Time:* average response time of all requests submitted to a flash SSD. This is the performance metric used to evaluate the performance of the four FTLs.
- *Std. Dev. of Number of Erasures:* the standard deviation of number of erasures that each flash block receives during a simulation experiment. This is the primary metric for measuring the four FTLs' performance in wear-leveling.
- *Delta Epsilon:* the difference in number of erasures between the block with the most number of erasures and the block with the least number of erasures in a flash SSD (see (1)). This is the secondary metric for measuring the four FTLs' performance in terms of wear-leveling.

TABLE III. SIMULATION PARAMETERS

| Parameter | Value (Fixed) – (Varied) |
|---|---|
| SSD Capacity (GB) | (2) – 2, 4, 8 |
| Page Size (KB) | (4) – 4, 8, 16, 32 |
| $k_\varepsilon$ | (10) – 0, 4, 6, 8, 10, 20,20, 50, 100, 1000 |
| HPT_SIZE | 400 |

### B. Overall Comparisons

In this section, we compare the overall performance of the four FTLs. Also, we evaluate the scalability of the four FTLs by investigating the impacts of flash SSD capacity on performance and wear-leveling. The maximal capacity of a flash SSD is set to 8 GB due to the limited footprint of the four real-world traces. The results from Fig. 4 show that all four FTLs exhibit a good scalability when the flash SSD capacity increases from 2 GB to 8 GB. We found that the two PM-based FTLs (PM and WECO-PM) consistently

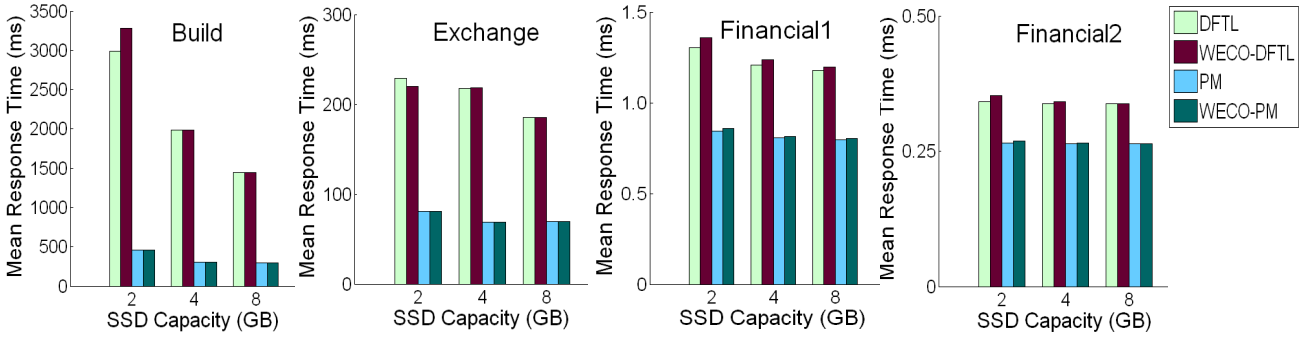


Figure 4. The impacts of flash SSD capacity on performance.

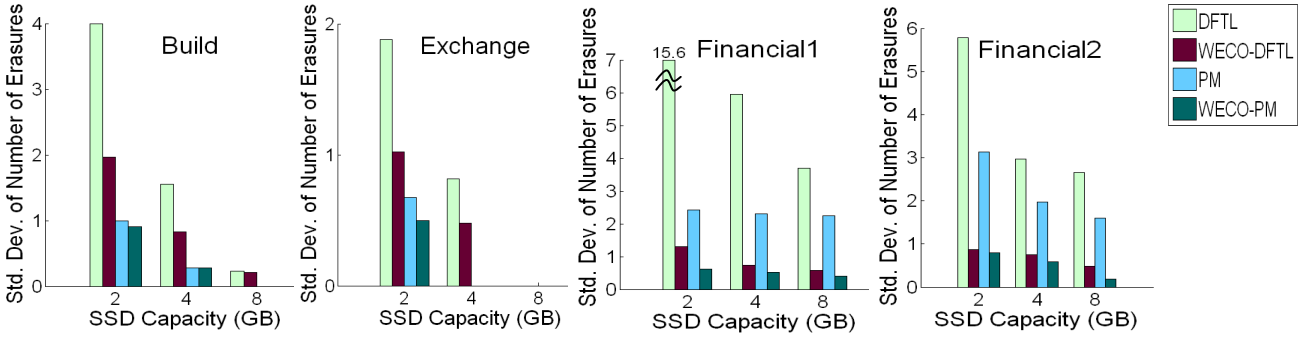


Figure 5. The impacts of flash SSD capacity on wear-levling.

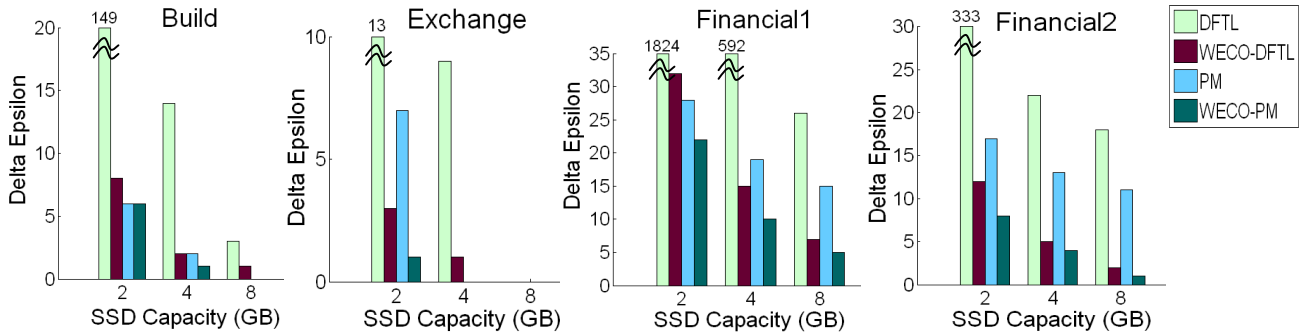


Figure 6. The impacts of flash SSD capacity on delta epsilon.

outperform the two DFTL-oriented algorithms (DFTL and WECO-DFTL) in all cases. As we explained before, PM can provide the highest performance although its cost for maintaining a large mapping table could be prohibitive. Next, we discovered that the performance of WECO-DFTL and WECO-PM are very close to that of DFTL and PM, respectively. In the worse case, WECO-DFTL increases mean response time by 9.9% compared with DFTL when flash SSD capacity is 2 GB in Build trace (see Fig. 4). In most scenarios, the performance gap between the two original FTLs and the two WECO-powered FTLs is almost unnoticeable. Compared with the original DFTL algorithm WECO-DFTL on average only degrades performance by 2.7% and 1.5% in Financial1 and Financial 2, respectively. Similarly, compared with the existing PM scheme WECO-PM on average merely lowers down performance by 1.03% and 0.45% in Financial1 and Financial 2, respectively. In fact, in occasional cases, the WECO-powered FTLs are slightly faster than the originals. For example, WECO-DFTL slightly performs better than DFTL by 3.9% for the Exchange trace with a 2GB SSD.

Fig. 5 demonstrates the four FTLs' performance in terms of wear-leveling because a lower standard deviation of number of erasures among all blocks in a flash SSD indicates a more even distribution of wear, which leads to a higher level of reliability. The two WECO-powered FTL schemes are obviously superior to the two original FTLs in wear-leveling in all cases. On average, WECO-DFTL performs better than DFTL by 50.4% in wear-leveling. Similarly, WECO-PM is on average 41.88% better than PM in wear-leveling. We argue that on average improving wear-leveling by more than 40% at the cost of performance degradation no more than 2.4% is worthwhile as it can largely improve flash reliability. Fig. 6 shows the impacts of flash SSD capacity on delta epsilon ($\Delta_e$, see (1)), which is another metric to show the degree of wear-leveling. A lower value of delta epsilon implies a better scenario for wear-leveling as the distribution of erasures among all blocks in an SSD becomes more even.

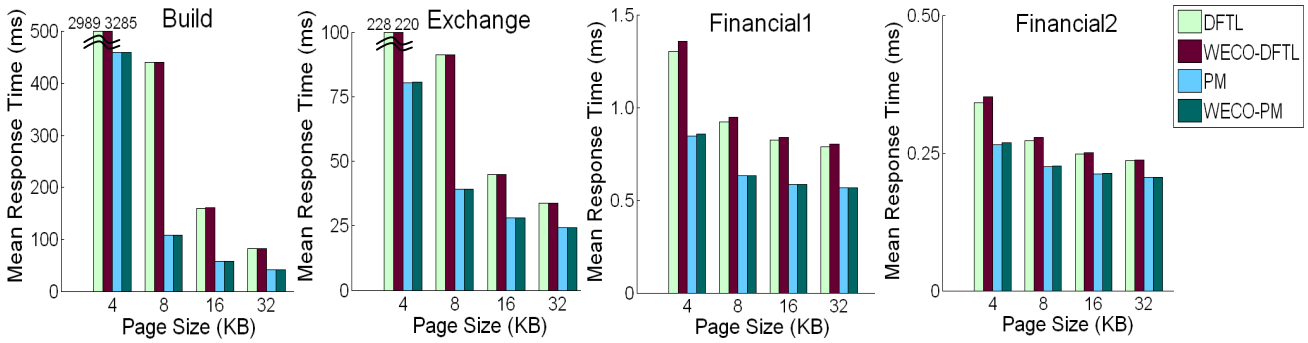


Figure 7. The impacts of flash page size on performance.

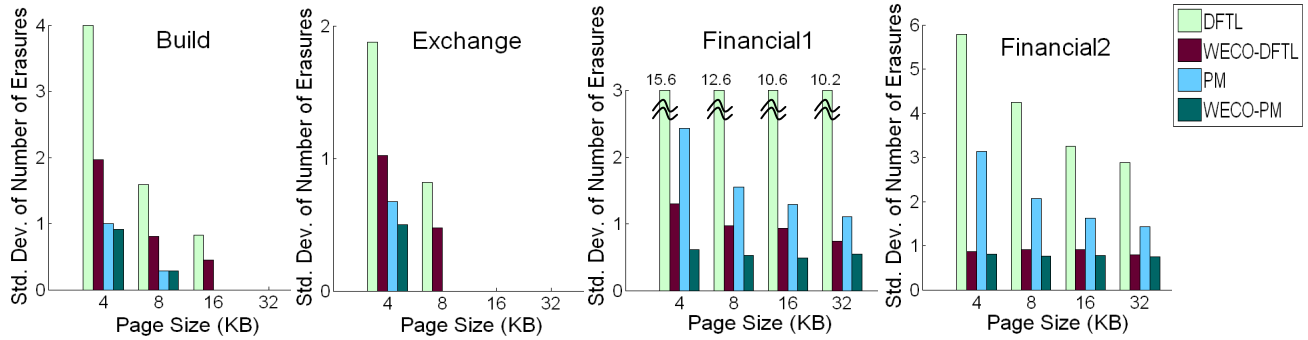


Figure 8. The impacts of flash page size on wear-leveling.

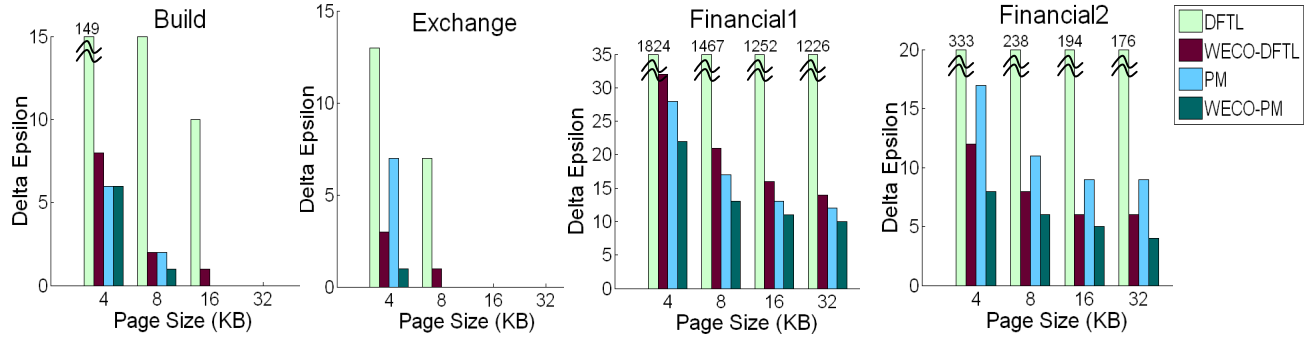


Figure 9. The impacts of flash page size on delta epsilon.

## C. *The Impacts of Page Size*

This experiment is intended to investigate the impact of flash page size on the WECO strategy. We vary the size of a flash page from 4 KB to 32 KB. Fig. 7 plots the performance of the four FTLs as functions of page size. Fig. 7, similar to Fig. 4, shows comparable mean response times between the FTLs incorporating the WECO strategy and the original FTLs. Still, several important observations can be drawn from Fig. 7. First of all, the performance of all four FTLs increases when the page size enlarges from 4 KB to 32 KB. This is because the space utilization is improved when larger pages are used, thereby decreasing the mean response time. Consider the example of a 32KB file. If the page size is 4 KB, then a 32KB file would take 8 pages. Updates to any of the 8 pages could result in a block erasure later. If the page size was increased to 8 KB, then the 32KB file would only consist of 4 pages, which obviously reduces the possibility of a future block erasure. From this example, it is easy to see that by increasing the page size the block utilization (see (1)) can be improved. More importantly, by using larger pages the number of erasures can be reduced as the number of pages decreases in the first place, which can improve overall performance of the flash SSD. Second, when page size increases from 4 KB to 8 KB, all four FTLs greatly improve their performance (see Fig. 7). This is especially true for Build and Exchange. However, when page size is further enlarged from 8 KB to 16 KB, the four FTLs did not gain too much performance improvement in Financial1 and Financial2 traces. On the contrary, the performance improvement is still substantial for the four algorithms in Build and Exchange traces. For example, in Build and Exchange traces WECO-DFTL further reduces its mean response time by 63.6% and 50.5%, respectively. Note that the average request size for Build is 10.5 KB and it is 11.5 KB for Exchange (see Table II). Thus, the majority

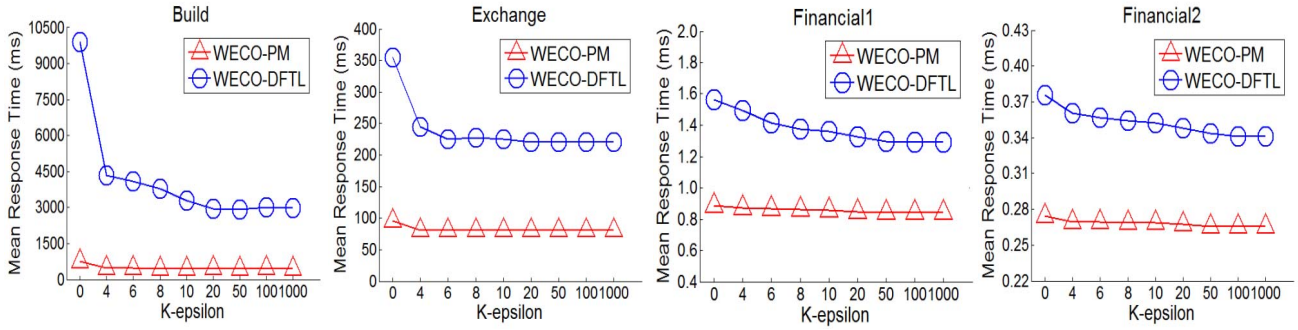


Figure 10. The impacts of k-epsilon on performance.

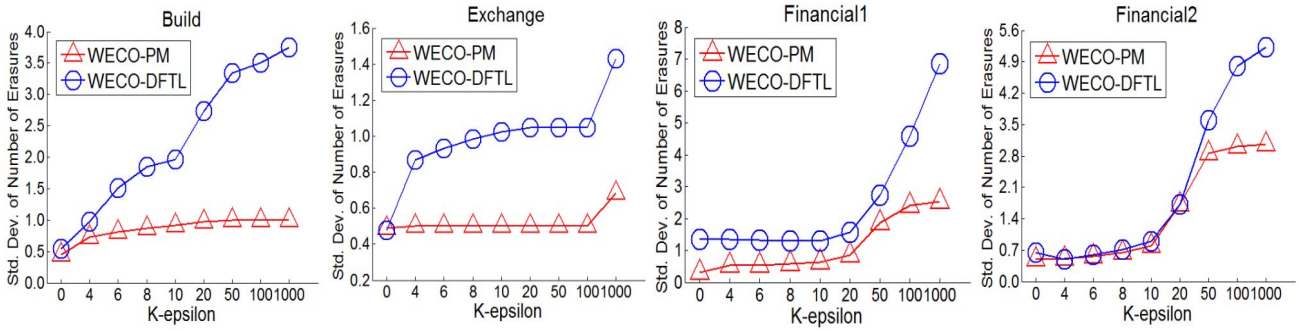


Figure 11. The impacts of k-epsilon on wear-leveling.

of requests can be accommodated by a single page if page size 16 KB is used. Otherwise, they need to involve two pages when 8 KB page size is employed. This explains why we still see substantial performance improvement when page size is further enlarged to 16 KB.

The general trend shown in Fig. 8 is that wear-leveling can be improved when page size increases. For Financial1 trace, in terms of standard deviation of number of erasures, WECO-DFTL on average is only 8% of that of DFTL. Note that in both Build and Exchange traces there are some bars missing in figures Fig. 5, Fig. 6, Fig. 8, and Fig.9. This is because there are no longer any erasures required during the entire course of the simulations.

### *D. Trade-offs between Performance and Wear-Leveling*

In this section we investigate the impacts of $k_\varepsilon$ (i.e., k-epsilon, see (1)) on the two WECO-powered FTLs. As for DFTL and PM, we do not include them in this group of experiments because they are irrelevant to $k_\varepsilon$. We tune the value of k-epsilon from 0 to 1000 to test its impacts.

Fig. 10 demonstrates that the two WECO-powered FTLs improve their performance while the value of k-epsilon increases. As we can see from (1), the value of $\lambda$ decreases when the value of k-epsilon increases, which in turn increases the weight of the utilization term in equation (1). Thus, the two FTLs prefer to select a block with lower utilization to be the victim whenever GC occurs. In other words, in this case the two FTLs are performance-driven while largely ignoring wear-leveling. This speculation is proved by experimental results shown in Fig. 10 to Fig. 11.

Fig. 10 clearly shows that the mean response time of both WECO-DFTL and WECO-PM decreases when k-epsilon increases. However, the impact of k-epsilon on WECO-PM is much lighter than that of on WECO-DFTL. As we explained, PM-based FTLs can achieve highest performance due to their fine granularity page-level address mapping. Therefore, the performance impact caused by GC is limited. When k-epsilon is equal to 0, $\lambda$ becomes one. In other words, WECO turns out to be pure wear-leveling-centric while completely overlooking performance during GC operations (see (1)). In this extreme case, both WECO-PM and WECO-DFTL exhibit the worst performance (see Fig. 10). When the value of k-epsilon is enlarged, the two FTLs noticeably improve their performance. Meanwhile, however, their performance in terms of wear-leveling decreases (see Fig. 11). Based on equation (1), this observation is understandable as with an increasing k-epsilon WECO starts to make a good trade-off between performance and wear-leveling by paying more consideration on performance when selecting victim blocks. Based on the results shown in Fig.10-11, the "sweet spot" for k-epsilon lies in the range between 10 and 20. Experimental results presented in this section verify the effectiveness of WECO in its ability to achieve a good balance between performance and wear-leveling during GC operations.

## VI. CONCLUSIONS

Flash memory has become ubiquitous in the commercial market, but it has still not become the de facto standard in enterprise-scale environments due to its inherent limitations such as out-of-place updates and coarse granularity of erase unit [1][6][18]. In particular, garbage collection can not only degrade performance due to its high overhead but also negatively affect flash SSD reliability due to an uneven

distribution of erasures it may cause. Unfortunately, current wear-leveling schemes [6][7][16] cannot solve the wear-out-uneven problem as garbage collection mechanism and wear-leveling scheme are two independent modules implemented in one FTL. Therefore, a wear-conscious garbage collection mechanism is much needed. In this research, we design and implement a new garbage collection mechanism called WECO, which can dynamically make good trade-offs between performance and wear-leveling during garbage collection depending on current wear conditions.

In order to implement WECO into two mainstream FTLs (i.e., DFTL and PM), we extended the DiskSim3.0/FlashSim simulation environment considerably. We provided a portable virtual environment as well as a scripting framework for analysis of traces to assist other flash SSD researchers. Our comprehensive experimental results show that WECO-DFTL and WECO-PM far exceed the two original FTLs in terms of wear-leveling while maintaining a similar performance. The future work of this research is to transport WECO into a hardware prototype based on a development board.

## ACKNOWLEDGMENT

We would like to thank Abdul Rahman for helping us setup experiment environment. This work was supported in part by the U.S. National Science Foundation under grants CNS (CAREER)-0845105 and CNS-0834466.

## REFERENCES

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," Proc. USENIX Annual Technical Conference, pp. 57-70, 2008.

[2] S. Baek, S. Ahn, and J. Choi, "Uniformity Improving Page Allocation for Flash Memory File Systems," Proc. 7th ACM/IEEE Int'l Conf. Embedded Software (EMSOFT 07), Sept. 2007, pp. 154-163.

[3] S. Boboila and P. Desnoyers, "Write Endurance in Flash Drives: Measurements and Analysis," Proc. 8th USENIX Conference on File and Storage Technologies (FAST), 2010.

[4] J.S. Bucy and G.R. Ganger, "The DiskSim Simulation Environment Version 3.0 Reference Manual," Pittsburgh, PA, Carnegie Mellon University, 2003.

[5] Build Server Trace, SNIA IOTTA Repository, http://iotta.snia.org/traces/158, Accessed 2010-04-20.

[6] Y.H. Chang, J.W. Hsieh, and T.W. Kuo, "Endurance Enhancement of Flash-Memory Storage Systems: An Efficient Static Wear Leveling Design," Proc. ACM 44th Annual Design Automation Conference (DAC 07), ACM Press, June 2007, pp. 212-217.

[7] L.P. Chang and C.D. Du, "Design and Implementation of an Efficient Wear-Leveling Algorithm for Solid-State-Disk Microcontrollers," ACM Transactions on Design Automation of Electronic Systems. 15, 1, Article 6, 2009.

[8] L. Chang and T. Kuo, "An adaptive striping architecture for flash memory storage systems of embedded systems," Proc. IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 187–196, 2002.

[9] F. Chen, D.A. Koufaty, and X. Zhang, "Understanding Intrinsic Characteristics and System Implications Of Flash Memory based Solid State Drives," Proc. 11th Int'l Joint Conf. Measurement and Modeling of Computer Systems (SIGMETRICS), pp. 181-192, 2009.

[10] C. Dirik and B. Jacob, "The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization," Proc. 36th Int'l Symp. Computer Architecture (ISCA), pp. 279-289, 2009.

[11] Exchange Trace, SNIA IOTTA Repository, http://iotta.snia.org/traces/130, Accessed 2010-04-20.

[12] C. Fox, D. Lojpur, and A. Wang, "Quantifying Temporal and Spatial Localities in Storage Workloads and Transformations by Data Path Components," IEEE Int'l Symp. Modeling, Analysis and Simulation of Computers and Telecommunication Systems (MASCOTS), 2008.

[13] E. Galand,S. Toledo, "Algorithms and data structures for flash memories," ACM Computing Surveys, 37, pp. 138-163, 2005.

[14] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings," Proc. 14th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 229-240, 2009.

[15] K. Ha and J. Kim, "A Program Context-Aware Data Separation Technique for Reducing Garbage Collection Overhead in NAND Flash Memory," Proc. 7th IEEE SNAPI, May 2011.

[16] P.G. Harrison and S. Zertal, "Investigating Flash memory wear levelling and execution modes," Proc. Int'l Symp. Performance Evaluation of Computer & Telecommunication Systems, pp. 81-88, 2009.

[17] J. Hsieh, L. Chang, and T. Kuo, "Efficient on-line identification of hot data for flash-memory management," Proc. ACM Symp. Applied Computing, pp. 838–842, 2005.

[18] Y. Hu, H. Jiang, D. Feng, L. Tian, S. Zhang, J. Liu, W. Tong, Y. Qin, and L. Wang, "Achieving page-mapping FTL performance at block-mapping FTL cost by hiding address translation," Proc MSST, pp. 1-12, 2010.

[19] H. Kim and S. Lee, "An Effective Flash Memory Manager for Reliable Flash Memory Space Management," IEICE TRANS. INF. & SYST., E85-D(6), June 2002.

[20] Y. Kim, B. Taurus, A. Gupta, and B. Urgaonkar, "FlashSim: A Simulator for NAND Flash-based Solid-State Drives," Proc Int'l Conf. Advances System Simulation, Sept. 2009.

[21] J. Lee, Y. Kim, G. Shipman, S. Oral, F. Yang, and J. Kim, "A Semi-Preemptive Garbage Collector for Solid State Drives," IEEE Int'l Symp. Performance Analysis of Systems and Software (ISPASS 11), April 2011, pp. 12-21.

[22] S.W. Lee, D.J. Park, T.S. Chung, D.H. Lee, S. Park, and H.J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," ACM Trans. Embedded Computing Systems (TECS), Vol. 6, Issue 3, July 2007.

[23] A. Leung, S. Pasupathy, G. Goodson, and E.L. Miller, "Measurement and Analysis of Large-Scale Network File SystemWorkloads," Proc. USENIX Annual Technical Conf., Boston, MA, June 2008.

[24] Oracle, VirtualBox Virtualizaton Software, http://www.virtualbox.org/, Accessed on 2010-10-25.

[25] SPC, "Storage Performance Council I/O traces," http://www.storageperformance.org/.

[26] J. Tjioe, Compiling_DiskSim3.0_v1.0, http://www.pdl.cmu.edu/DiskSim/Compiling_DiskSim3.0_v1.0.pdf, 2010, Accessed 2010-10-25.

[27] J. Tjioe, Compiling_FlashSim_v1.0, 2010.