Towards Improving Endurance and Performance in Flash Storage Clusters

Mohammed Salman

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Ali R. Butt, Chair
David Raymond
Haibo Zeng

May 8, 2017
Blacksburg, Virginia

Keywords: Flash Storage,Wear Balancing,Flash Endurance,Write Off-loading

Towards Improving Endurance and Write-Performance in a Flash Storage Cluster

(ABSTRACT)

NAND flash-based Solid State Devices (SSDs) provide high performance and energy efficiency and at the same time their capacity continues to grow at an unprecedented rate. As a result, SSDs are increasingly being used in high end computing systems such as supercomputing clusters. However, one of the biggest impediments to large scale deployments is the limited erase cycles in flash devices. The natural skewness in I/O workloads can results in Wear imbalance which has a significant impact on the reliability, performance as well as lifetime of the cluster. Current load balancers for storage systems are designed with a critical goal to optimize performance. Data migration techniques are used to handle wear balancing but they suffer from a huge metadata overhead and extra erasures. To overcome these problems, we propose an endurance-aware write off-loading technique (EWO) for balancing the wear across different flash-based servers with minimal extra cost. Extant wear leveling algorithms are designed for a single flash device. With the use of flash devices in enterprise server storage, the wear leveling algorithms need to take into account the variance of the wear at the cluster level. EWO exploits the out-of-place update feature of flash memory by off-loading the writes across flash servers instead of moving data across flash servers to mitigate extra-wear cost. To evenly distribute erasures to flash servers, EWO off-loads writes from the flash servers with high erase cycles to the ones with low erase cycles by first quantitatively calculating the amount of writes based on the frequency of garbage collection. To reduce metadata overhead caused by write off-loading, EWO employs a hot-slice off-loading policy to explore the trade-offs between extra-wear cost and metadata overhead. Evaluation on a 50 to 200 node SSD cluster shows that EWO outperforms data migration based wear balancing techniques, reducing up to 70% aggregate extra erase cycles while improving the write performance by up to 20% compared to data migration.

Towards Improving Endurance and Performance in Flash Storage Clusters

Mohammed Salman

(GENERAL AUDIENCE ABSTRACT)

Exponential increase of Internet traffic mainly from emerging applications like streaming video, social networking and cloud computing has created the need for more powerful data centers. Datacenters are composed of three main components- compute, network and storage. While there have been rapid advancements in the field of compute and networking, storage technologies have not advanced as much in comparison. Traditionally, storage consists of magnetic disks with magnetic parts which are slow and consume more power. However, Solid State Disks (SSDs) offer both better performance and lower energy. With the price of these SSDs being comparable to magnetic disks, they are increasingly being used in storage clusters. However, one of the biggest drawback of SSDs is the limited program erase (P/E) cycles. There is a need to ensure the uniform wearing of blocks in a SSD. While solutions for this do exist for a single SSD device, usage of these devices in a cluster poses new problems.

This work introduces EWO which is a wear balancing algorithm that balances wear in a flash storage cluster. It carried out load balancing in a flash storage cluster while incorporating the wear characteristics as a cost function. EWO carries out lazy data migration also referred to as write offloading. To alleviate the metadata overhead, the migration is performed at the slice level.

To evaluate EWO, a distributed key value store emulator was built to simulate the behavior of an actual flash storage cluster.

Dedicated to my parents,

for their endless love, encouragement, prayers, and support ...

Acknowledgement

I would like to express my sincere gratitude and appreciation to all the people that helped me along this challenging yet enjoyable path to finishing this research work.

Dr. Ali R. Butt, for giving me the chance to start my academic research at Distributed Systems and Storage Lab, and guiding my research along the path.

I would also like to thank Dr. David Raymond and Randy Marchany, my supervisors at the IT Security Lab, for their continued guidance and support during my time here at Virginia Tech.

Dr. Haibo Zeng, for taking time from his busy schedule and being on my committee.

Dr. Nannan Zhao for helping me with her knowledge and guidance on my thesis.

Mark De Young and Ali Anwar, for showing me the efficient methodology of doing academic research and helping me get through grad school.

Everyone at the Distributed Systems and Storage Lab(DSSL) and IT Security Lab(ITSL), for supporting me and helping me during my troubles.

And finally my family and friends for backing me up all the time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The past several decades have witnessed the success of flash memory as a storage medium for mobile computing devices due to its superiority in terms of high throughput, persistence and lower power consumption. The development of commodity flash devices, e.g., PCIe SSDs, expand the role of flash memory in the enterprise storage servers. Previous work has shown that using SSDs as a per-server disk cache could effectively bridge the gap between RAM and HDD for a distributed storage system, such as HyCache [77], HDStore [32], FlashCache [43], FlashTier [64], and FaCE [42]. Recently much interest has been focused on all-flash or disk-free server storage, such as FlashStore [30] and ChunkStash [31]. Flash-based storage servers that can play a big role in accelerating application performance are then clustered together and managed as a single entity for high reliability as well as availability via distributed storage platforms such as FAWN [6], BlueDBM [39], QuickSAN [18] and CORFU [16].

A promising use case is the flash storage disaggregation [47], where a cluster of flash devices are connected via low-latency network [3], boosting and scaling-out datacenter applications such as FAWN [6], BlueDBM [39], and CORFU [16]. These features are especially helpful in High Performance Computing applications. Flash devices differ from magnetic devices as they write data at the page level but erase data at a larger flash block level. The erase process is time-consuming (relative to transfer speeds) and also has implications for device

endurance, as the number of program/erase cycles of a flash location is limited. SSDs typically provide a flash translation layer (FTL) within the device, which maps logical addresses to physical locations. A host can access individual flash blocks that are usually kilobytes in size, and if some live blocks are physically located in the erasure unit (called victim block) is being recycled, the FTL will invoke garbage collection by copying the live data to a new location and then erasing the previous location to make it available for new writes. Hence, garbage collection process leads to write amplification and has a significant impact on both performance and endurance of flash-based storage.

Despite its superiority, flash memory has two critical technical constraints

1. **No in-place overwrite (Out-place-update)** – an erasure block must be erased before writing any data page in this block.
2. **Limited program/erase (P/E) cycles** – an erasure block can wear out after a certain number of P/E cycles (typically 10,000-100,000). The endurance and lifetime of flash memories decrease as it is repeatedly erased and written over. Therefore, the endurance and lifetime of flash memory strongly depend on the write intensiveness of workload.

The I/O workload is unbalanced in storage clusters, mainly because various objects generate different loads (e.g. have different popularity) and object popularity changes over time. Data placement schemes can evenly distribute data on a cluster of storage servers in an efficient and effective manner [33] [68] [35]. However, load balancing of I/O workloads across servers has remained an open problem [53]. Non-uniform I/O behavior from diverse workloads can result in creation of hotspots and significant imbalance across servers.

Extreme-scale high performance computing (HPC) systems leverage SSD arrays to scale-out the I/O performance. While SSD arrays are in a position to gradually replace spinning disk arrays due to better performance and low power consumption (i.e., Triple-A [41]), the economical factors are not yet practical. However, researchers and practitioners are already

adopting SSD-based burst buffer architecture [36, 4] to cache burst writes for large-scale scientific applications, and asynchronous draining of data to the back-end parallel file systems. The maintenance of SSD devices at scale raises many concerns. For instance, any maintenance required by the storage devices may require taking the entire node offline. This not only incurs administrative cost (e.g., SSD gets replaced when they are worn out) but also performance degradation especially when SSDs are crucial for burst buffer I/O nodes performance (e.g., in the upcoming SUMMIT cluster [4]). Given the skewed characteristic of workloads (especially write intensity), the wear is unbalanced over flash-based servers in storage clusters. The flash memories associated with heavily loaded servers perform more erasure operations, therefore wear more rapidly than others. Moreover, for a heavily loaded server, frequent erasure operations cause higher overall performance degradation since erasure operation consumes more time than read/write operation. Consequently, wear unbalance has a critical effect on the overall lifetime and reliability as well as performance of the whole cluster.

To address the challenge of balancing the wear across flash-based servers (called flash server for short), researchers can take inspiration from data migration. Data migration schemes were originally proposed to dynamically balance load across a large number of spinning hard drives based servers by moving hot (i.e., dynamically or frequently updated) data from heavily loaded servers to the lightly loaded servers [70] [51]. For example, EDM [63] is a data migration-based wear balancing scheme. It moves a certain number of objects from the flash devices with higher erasure cycles to the devices with lower cycles for balancing the wear intensity in the whole cluster.

Although data migration-based wear balancing scheme can reduce the wear variance of the flash-based storage cluster, the writes generated by data migrations are considered as an additional overhead. These additional writes during data migration can incur a considerable write amplification overhead inside flash devices and consequently cause more garbage collection (GC) and significant extra-wear of the whole cluster as seen in section 7

In this thesis, we propose a practical technique called Endurance-aware Write Off-loading

(EWO) for balancing the wear across different flash servers. EWO can be categorized as a wear leveling algorithm. Traditional wear leveling algorithms are designed for individual flash device. With the use of flash devices in enterprise server storage, the wear leveling techniques have to be designed considering wear variance to a greater extent.

1.1 Contributions

The contributions of the thesis are as follows-

- We develop a flash-friendly wear balancing technique operating at the server level which mitigates the wear balancing overhead and minimizes erasures imposed on flash devices.
- Wear leveling is ensured by off-loading writes from the flash servers with high erase cycles to the ones with low erase cycles by first quantitatively calculating the amount of writes based on the frequency of garbage collection.
- To reduce meta-data overhead caused by write off-loading, EWO employs a hot-slice off-loading policy to explore the trade-offs between extra-wear cost and the meta-data overhead.
- EWO is integrated with a distributed flash-based Key-Value store application. Results show that EWO outperforms the state-of-the-art data migration based wear balancing techniques by reducing the erasures upto 70%.

1.2 Thesis Organization

Chapter 2 formally defines the concepts of flash endurance, workload imbalance and how it leads to wear imbalance. It also gives an overview of the different levels at which wear-

leveling is performed i.e page-level, block-level and server or cluster level.
Chapter 3 discusses the prior art. The related work can be divided into four categories- works based on flash endurance, wear leveling for a single disk, wear leveling at the cluster level and examples of distributed flash storage systems.

Chapter 5 talks about the EWO algorithm in detail. It introduces the concept of slice and how by utilizing the slice we are able to minimize the meta-data overhead.

Chapter 4 is a short discussion about the Flash simulator and how it models an SSD. The chapter talks about the software and hardware design on the simulator and how it models an SSD. In addition, we also talk about protocol buffers and how a meta-data object is handled.

Chapter 6 talks about the implementation and the experimental setup. This includes our emulator design and modes of communication between the KV store clients and our emulator.
Chapter 7 talks about the evaluation performed and also includes a discussion of the experimental results.
Chapter 8 summarizes the results and discusses some of the future work including specific shortcomings of EWO and how they can be addressed.

Chapter 2

Background

In this chapter, we introduce the issue of wear imbalance in the flash-based storage clusters, and then we discuss different classes of wear-leveling algorithms based on the abstraction level at which they operate i.e. page-level, block-level and server-level.

2.1 Flash endurance, Workload imbalance and Wear imbalance

2.1.1 Introduction of Flash Memory

Flash memory chips store data in a large array of floating gate memory cells (i.e. metal-oxide-semiconductor field-effect transistor (MOSFET)). Typically, flash memory has two types, NOR based and NAND based. The memory cells in NOR based flash are organized in parallel. Each cell can be programmed and erased via a single contact as shown in figure 2.1. NOR based flash exhibits superior random read and write performance as it allows random access to any location of the memory. In contrast, NAND based flash employs serial

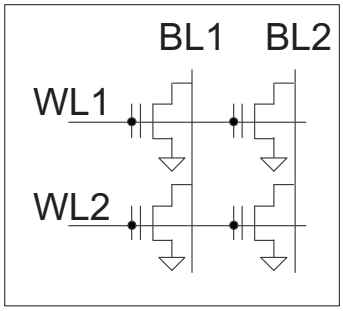


Figure 2.1: NOR memory architecture: Cells in parallel.

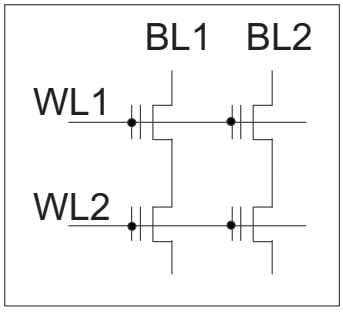


Figure 2.2: NAND memory architecture: Cells in series.

array architecture. Figure 2.2 shows two memory cells that are organized serially. Cells can be written individually but must be erased in blocks. NAND based flash has a much higher storage density and lower cost per bit. Hence, NAND based flash has been the most predominant of the two types. Each flash chip consists of a number of blocks that are basic units of erase operations. Each block further consists of a constant number of pages that are basic units of read and write operations. In addition, most flash memories also provide a spare area for each page to store out-of-band (OOB) data, such as the error correction code, the logical page number, and the state flag for the page.

NAND flash memory cells usually have a control gate, floating gate, source and drain. Electric charge is applied to the floating gate during write operations and removed during erase operations via Fowler-Nordheim (FN) current tunneling or via Hot Carrier Injection (HCI) mechanism [56]. This stored charge causes changes in the threshold voltage of the underlying transistor, which may then be sensed by the read circuitry. Although FN tunneling and HCI

are beneficial for programming and erasing flash memory, they also affect flash endurance and reliability. Over time, the flash memory cells will lose its insulating properties leading to the inability to be programmed or erased. In this case, flash memory cells have a limited lifetime [56].

NAND flash cells fall into two categories: single-level cell (SLC) with a single bit stored on each cell, and multi-level cell (MLC) with 2 to as many as 4 bits for each cell. The benefit of MLC storage is that the capacity can be increased with a cheaper price per bit than SLC memories. However, significant degradation in flash endurance occurs with increasing number of bits per cell. MLC memories are more prone to endurance effects with respect to SLC memories. For example, the endurance of an MLC flash-memory block is only 10,000 (or 5,000) erase cycles while its SLC flash-memory counterpart is 100,000 erase cycles [21].

2.1.2 Workload imbalance

Load imbalance is an inherent characteristic of workloads [8, 7, 9, 13]. First, there is a large variation among objects in their access frequency [12] (i.e. Popularity). For example, several researchers have observed that the frequency with which file are requested generally follows a *Zipf-like* distribution [65]: the relative probability of a request for the i^{th} most popular file is proportional to $\frac{1}{i^{\alpha}}$ with $0 \leq \alpha \leq 1$. Further, I/O requests are skewed and request skew widely exists in workloads. Facebooks distributed key-value (KV) store workload analysis [15] reports high access skew and time varying workload patterns, implying existence of imbalance in datacenter-scale production deployments. Second, object popularity changes over time. An object might either become hot or cold depending upon its transient access pattern. Some objects exhibit a ascent or decline in popularity while other objects exhibit a relatively stable popularity across days [51].

In most storage clusters, hash-based data placement is commonly used for evenly distributing data to servers, such as DHT and CHT [35]. Objects are *hashed* on to a hash space, which is partitioned among storage servers. The size of partitions each server owns is proportional

to its storage capacity. If all objects generate similar load (e.g. have the same popularity), the load of each server is proportional to its capacity and server workloads are balanced in storage clusters. However, as mentioned above, the workloads are non-uniform. Thus, the data distribution algorithms cannot guarantee load balance across different servers and server I/O workloads are imbalanced in storage clusters.

2.1.3 Wear imbalance

Imbalanced workload across different flash-based storage servers results in wear imbalance. Write requests have significant impact on flash memories since writes influence garbage collection (GC) frequency of flash memories. The more data written into a flash server, the more erasure cycles it has. Consequently, erasure cycles are skewed over flash servers and the flash memory associated with heavily loaded servers wear out faster than the flash memory with lighted loaded servers.

Wear imbalance causes the performance as well as the reliability of the whole storage cluster degradation. First, the heavily loaded SSD servers serve more writes and perform more erasure operations. Erasure operation requires more time than a read or write operation (about 1.5 ms) [55] to reset the memory cells. For a heavily loaded SSD server, frequent erasure operations over a fixed period results in higher overall performance degradation. Thus, wear imbalance makes heavily loaded servers become high-utilized (e.g. the performance bottleneck of the system) while others are under-utilized. Second, since SSD servers are not evenly utilized, the heavily loaded servers wear out faster than the lightly loaded SSD servers. Therefore, wear imbalance has a critical effect on the system's lifetime (or the mean-time-to-failure), and the reliability as well as performance of the whole cluster degrades consequently.

To evaluate the impact of skewed workload on erasures, we built a distributed flash-based key value store that maps data to a 50-node cluster using consistent hashing. The detailed design is explained in Section 4.

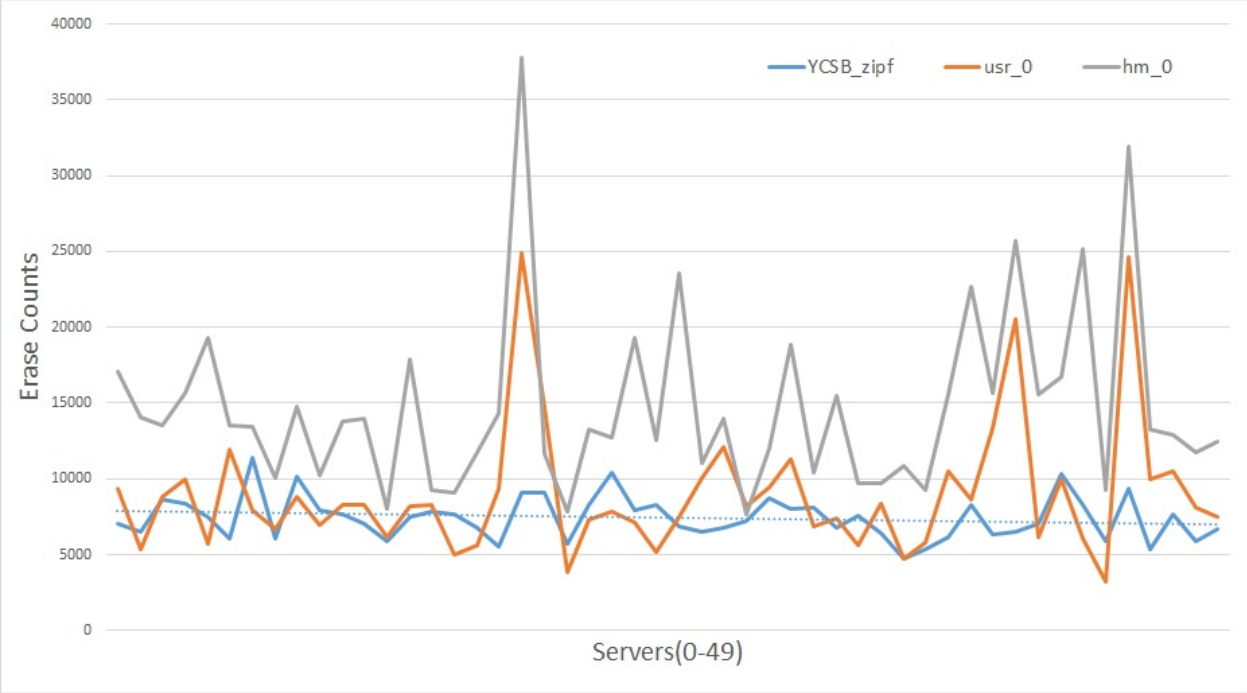


Figure 2.3: Erasure Count Distribution for MSR and YCSB traces.

We measured the number of erasures and write performance under three scenarios -

- Data Center KV Store workloads where we use YCSB KV store workload generator [28].
- Enterprise Storage workload which uses two traces from the MSR-Cambridge repository- usr_0 and hm_0 [14].
- HPC workloads which are generated using the IOR MPI benchmark [52].

Figure 2.3 shows the skewed erasure distribution for the MSR and YCSB traces while figure 2.4 shows the erase count distribution for an HPC workload without EWO. The results showed that the erase count is severely imbalanced. This would inevitably result in an increase in the administrative costs in datacenters with frequently replaced flash devices.

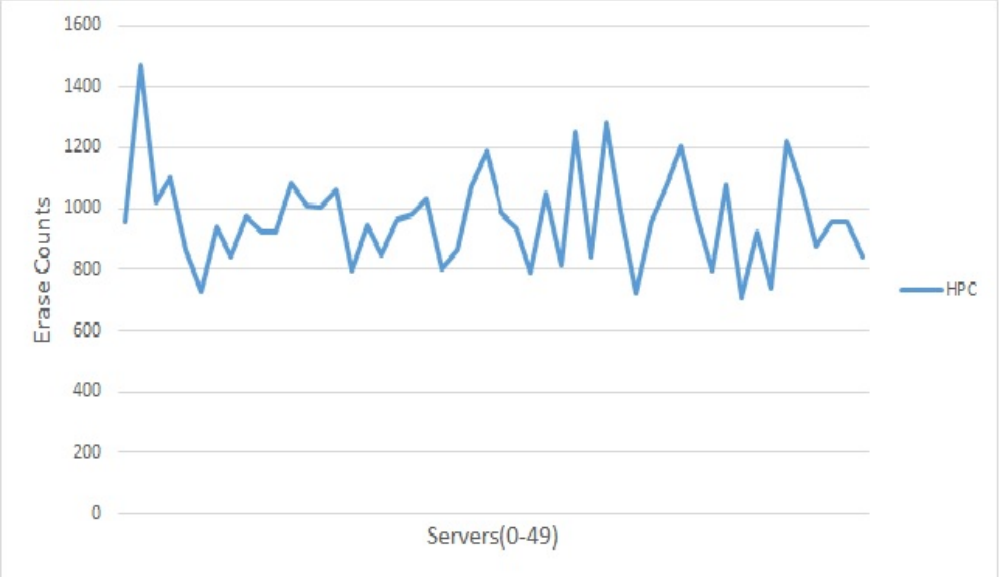


Figure 2.4: Erasure Count Distribution for HPC Trace

2.1.4 Write Amplification

Write Amplification is an undesirable process in which the amount of data written physically to the device is much larger than the amount of data that is intended to be written. Since flash devices erase data at a coarser granularity compared to the writes, the erase process involves moving both the meta-data and the data more than once. As a result, update of data involves reading a portion of flash updating it and then writing it to a new location, along with erasing the new location if it was previously used. As a result, much larger portions of flash must be erased and rewritten than actually required. This increased overhead increases the number of writes to a SSD location reducing its lifetime. The increased writes also consume CPU time which adversely affects the random write performance to the SSD. Figure 2.5 explains the process of write amplification.

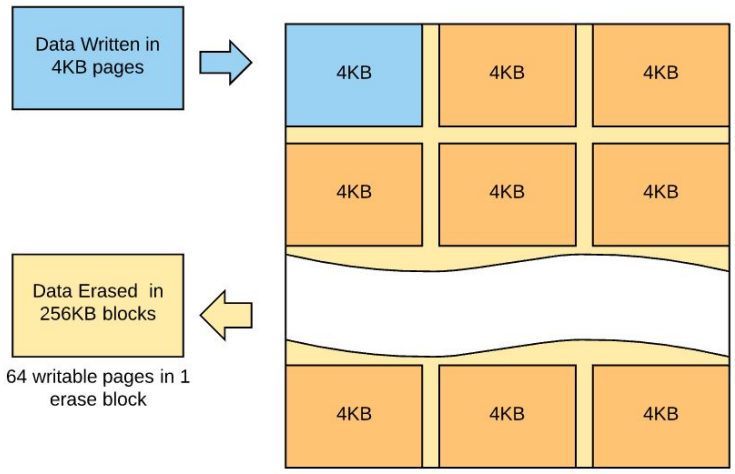


Figure 2.5: Write Amplification Explained

2.2 Wear Leveling Algorithms

Typically, flash memory cells wear out after a limited number of erasure cycles. As high density solid state disk becomes more popular in both commodity and commercial systems, the degradation of devices becomes a considerable concern. Various wear leveling algorithms have been proposed to extend the lifetime of flash memory. Most of the wear leveling algorithms try to improve the overall lifetime of flash devices by evenly wearing all blocks on block-level, or by evenly wearing all pages within a block on page-level, or by evenly distributing erasure cycles to servers on server-level.

2.2.1 Block-Level Wear Leveling Algorithms

The existing block-level wear leveling algorithms fall into two categories – Dynamic wear leveling and Static wear leveling. All of these block-level wear leveling algorithms have two goals:

- Evenly distribute block erases over all flash devices

- Reduce extra overhead.

Dynamic Wear Leveling algorithms achieve wear leveling by recycling blocks of hot data with higher erasure cycles and reusing blocks of cold data with lesser erase cycles. L. Chang et al. proposed a lazy wear leveling algorithm to adaptively reach a good balance between wear evenness and overhead with low cost [19]. To select a good threshold for a good balance between wear evenness and overhead, it adopts an analytical model. D. Jung et al. proposed a memory-efficient group-based wear-leveling algorithm to reduce memory cost [40].

Static Wear Leveling algorithms move cold data to the blocks with higher erasure counts thereby improving the even spread of wear. Y. Chang et al. proactively move static (i.e. infrequently updated) blocks to distribute wear over the entire flash space. It uses an adjustable housekeeping data structure to reduce meta-data overhead and memory-space requirements [21]. Rejuvenator is a static wear leveling algorithm designed for large-scale NAND flash memory [59]. It first clusters the blocks into different groups based on their erasure counts. Then, it places frequently accessed data in lesser erased clusters and rarely accessed data in higher erased clusters.

2.2.2 Page-Level Wear Leveling Algorithms

Based on the observation that various pages degrade at different speed within a block, X. Jimenez et al. proposed wear unleveling technique to extend the device lifetime by relieving the weakest pages and putting more stress on the strongest ones [38]. Phoenix is proposed to keep on using worn-out MLC blocks as SLC blocks. In this case, the useless unreliable blocks are protected by storing only a single bit per cell. Consequently, its lifetime is significantly extended [37].

2.2.3 Server-Level Wear Leveling Algorithms

EDM [63] is an endurance-aware data migration scheme for wear balancing in SSD storage clusters. EDM balances the wear among different flash-based servers through data migration. It moves a certain number of objects from the servers with higher erasure cycles to the servers with lower cycles for balancing the wear speeds in the whole cluster.

Most of the algorithms mentioned above work on a single SSD device and operate either at the block-level or the page-level with the exception of EDM. In the next section, we look at approaches for wear leveling/balancing at the cluster or server level.

2.3 Protocol Buffers

Protocol Buffers[71] are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data. They are similar to XML but comparatively just faster and simpler. Protocol buffers allow us to specify the structure of the data or message we want to transmit which includes optional and required fields in a proto file which can then be compiled using the protoc compiler which can then be compiled into specially generated source code to easily write and read your structured data to and from a variety of data streams. Further, we can even update your data structure without breaking deployed programs that are compiled against the "old" format. Below is the representation of the data structure we are using for the data objects beings transferred between the flashmanager and the flashserver.

Metadata Object Structure

```
message Message {
        required uint32          obj_no = 1;
        required uint32          offset = 2;
        required uint32          length = 3;
        enum trace_operator_t {
                operator_read = 1;
                operator_write = 2;
                operator_trim = 3;
        }
        required trace_operator_t operator_t = 4;
        required float             timestamp = 5;
        required int32          ec_index = 6;
        enum request_type_t {
                need_flash_info = 1;
                not_need_flash_info = 2;
                shut_cluster = 3;
        }
        required request_type_t rq_type = 7;
        required float             flash_utilization = 8;
        required float             flash_victim_utilization = 9;
        required float             flash_full_blk_utilization = 10;
        required uint32            node_nr_erases = 11;
        required float             local_log_utilization = 12;
        required uint64            request_number = 13;
        optional int32             response_time = 14;
}
message Response {
        required string rsp = 1;
        required float  rsp_time = 2;
}
```

Chapter 3

Related Work

The existing work can be divided into four categories since EWO leverages existing work in all these categories. Load balancing techniques [48, 49, 11] have been used to better exploit the datacentre heterogenity. Such techniques can also improve the lifetime of storage devices but they include using HDD. However, most of these methods cannot be applied to flash based devices because of the asymmetric write nature of SSD devices. In case of HDD, both reads and writes affect the performance in a similar manner. On the other hand, in case of flash based devices due to the limited number of P/E cycles. Individual flash devices contain an FTL layer which ensures the wear is balanced on a single device. EWO looks at wear balancing at the cluster level and in many ways is complementary to these methods. Interdisk wear leveling techniques include wear leveling algorithms explored at the cluster level and serve as the baseline which we compare EWO against.

3.1 Flash endurance

A large body of work has examined flash endurance. [17] examines the write endurance of USB flash drives using a range of approaches: chip-level measurements, reverse engineer-

ing, timing analysis, whole-device endurance testing, and simulation. They found that the measure erase counts are very similar to the ones observed by reverse engineering the FTL parameters. They also provide a numerical bound on the endurance achievable by an online algorithm under arbitrary or malicious access patterns which can be used as the baseline for EWO. CAFTL[22] is a Content-Aware Flash Translation layer which removes unnecessary unnecessary duplicate writes thereby effectively reducing flash write traffic. In addition, it also coalesces redundant data in SSDs which further improves the efficiency of garbage collection and wear-leveling. Further, they also design acceleration techniques to reduce the runtime overhead and minimize the performance impact caused by extra computational cost.

Gokul Soundararajan et al [58] model the physical processes that affect endurance, which include both stresses to the memory cells as well as a recovery process model using which they are able to demonstrate higher number of erases and counts compared to the one mentioned in the datasheets.

Youyou Lu et al [54] analyzes the effect of fileystem mechanisms on wear leveling. Methods such as journaling, metadata synchronization, and page-aligned update, can induce extra write operations resulting in write amplification. It proposes an object based flash translation layer (OFTL) which leverages page metadata due to which there is a lazy persistence of index metadata and elimination of journaling [66]. Coarse-grained block state maintenance reduces persistent free space management overhead. With byte-unit access interfaces, OFTL is able to compact and co-locate the small updates with metadata to further reduce writes.

In addition, there have been other techniques such as log-structured caching [67], inclusion of phase change memory into SSDs [69], and combining multiple bad blocks into virtual healthy blocks [72, 38] which have been explored to improve the lifetime of flash devices. These works are orthogonal and complementary to EWO.

3.2 Intradisk Wear Leveling

Existing wear leveling techniques are essentially single disk data distribution schemes as they distribute erases and writes across the flash medium within a single SSD. Chae et al[40] proposes a memory-efficient group-based wear-leveling algorithm. The group-based algorithm achieves a small memory footprint by grouping several logically sequential blocks and managing only the summary information for each group. They propose an effective group summary structure and a method to reduce unnecessary wear leveling operations in order to enhance the wear-leveling performance. LazyFTL[55] maintains a global mapping table which means it reserves two partitions in the flash memory and delays the modifications of the GMT caused by write requests or valid page movements. By maintaining the update buffer which is a very small overhead, [55] is able to avoid the excess duplicate writes and the write overhead involved with each of them.

Chang et al[19] introduces a dual pooling algorithm which realizes two key ideas: To cease the wearing of blocks by storing cold data, and to smartly leave alone blocks until wear leveling takes effect. It requires no complicated tuning, and it resists changes of spatial locality in workloads.

Rejuvenator[59] is a static wear leveling algorithm which carries out lazy migration of cold data. The data migration is carried out only on the next update request thereby reducing the write overhead. It clusters the blocks into different groups based on their current erase counts. Hot data is placed in blocks in lower numbered clusters and cold data in blocks in the higher numbered clusters. The range of the clusters is restricted within a threshold value. This threshold value is adapted according to the erase counts of the blocks. As a result, it minimizes the number of stale cold data migrations and also spreads out the wear evenly.

Unleveling[38] attempts to identify the weakest cells in the flash storage and introduces a wear unbalancing technique that let the strongest cells relieve the weak ones in order to lengthen the overall lifetime of the device. As a result, [38] periodically skips or relieves the

weakest pages whenever a flash block is programmed.

Phoenix[37] attempts to address the problem of balancing erase counts on a multi level cell. In case a flash block becomes unreliable to store multiple bits per cell , it can be revived by storing only a single bit per cell. Although the capacity is halved, the lifetime of the system is increased without jeopardizing the stored data.

3.3 Interdisk Wear Leveling

Application of SSD arrays in enterprise data-intensive applications is growing. As the cost of SSD devices goes on decreasing, this trend is going to increase. In such an environment, significant variances in number of writes and merge operations that are sent to individual SSDs have been observed . Greenan et al[34] manages EC stripes to increase reliability and operational lifetime of such flash memory-based storage systems, and uses a log-structured approach that does not need explicit wear balancing as data is appended and not updated in place. In contrast, EDM [63] also targets SSD arrays but use data migration to achieve wear balance across the SSDs in the array.It moves a certain number of objects from the servers with higher erasure cycles to the servers with lower cycles for balancing the wear speeds in the whole cluster. Similarly, SWANS [73] dynamically monitors the variance of write intensity across the array and redistributes writes based only on the number of writes that an SSD has received to prolong the SSD arrays service life. These methods share with EWO the goal of wear leveling across an SSD array, however unlike EWO they do not consider the effect of the meta-data overhead and the extra writes at various points in storage hierarchies and their impact on overall wear balancing.

3.4 Distributed flash storage systems

FAWN [6] uses small amounts of local flash storage across a number of low-power resource-constrained nodes to enable a consistent and replicated key-value storage system. It also uses a log-structured approach with in each of its individual drives to improve wear leveling. CORFU [16] extends the local log-structured design by organizing the entire cluster of SSDs as a global shared log. Both of these systems utilize homogeneous nodes and replication for high availability. In contrast, EWO focuses on EC storage solutions, which offer higher storage efficiency and exploits the interactions between various storage hierarchy to improve overall flash lifetime in flash-based cluster. Cloud storage has long been a hot topic and attracted great attention from both industry and academia. To improve the efficiency and to bridge the gap between applications and storage, MBal [24] adopts a resource-partitioning paradigm to enable flexible and efficient load balancing at the cloud memory cache layer. In order to maximize the application performance and minimize the incurred monetary cost, CAST [25, 26, 76] intelligently places data at the most appropriate storage service tier in public cloud. Furthermore, [23] proposes a simple offline flash caching heuristic which can be used to serve as a practical best base baseline for evaluating any online flash caching policies with the goal of maximizing flash device endurance or lifespan without sacrificing performance.

Chapter 4

Endurance-aware Write Offloading

The key challenges in balancing wear in flash-based storage clusters are

1. Ensuring Wear Balance across different flash servers
2. Minimizing extra-cost during wear balancing, such as extra-wear and meta-data overhead.

This section first details how migrating data will add additional *wear* (i.e. extra-wear) to the cluster, then describe how our EWO design leverages write off-loading to minimize extra-cost during wear balancing.

4.1 Data Migration and Write Off-loading

As mentioned in Section 2, unbalanced workload across different flash servers incurs wear imbalance. In this case, researchers can take advantage of workload balancing techniques designed for HDD storage cluster for doing wear balancing in flash-based storage cluster, such as data migration, which has been widely used for load balancing in storage clusters [70].

EDM [63] is an endurance-aware data migration scheme for balancing the wear in SSD storage clusters by moving a certain amount of data across SSDs.

While the data migration technique guarantees the workload balance of the HDD storage clusters, it is inefficient for flash-based storage clusters due to excessive write amplification during data migration, as shown in Figure 4.1. We first consider that data migration is constructed on the HDD storage servers. As shown, objects, *Obj1, Obj2, and Obj3*, are first moved from source server A to destination server B to ensure workload balance between A and B (see arrow 1). Then, the writes to these three objects are remapping to server B (see arrow 2) and in-place updates are performed in HDD. In other words, as shown in Figure 4.1, the updates are overwritten to the original moved objects, *Obj1, Obj2, and Obj3*.

When data migration is set up on flash servers, the situation is different. As shown in Figure 4.2, after being migrated, the updates to the migrated objects, *Obj1, Obj2, and Obj3* are written out of place in the original moved objects by address mapping of the on-device layer FTL (see arrow 3). In this case, once the updates to the original moved objects are committed on flash memory, the flash pages that store original moved objects are marked as invalid and will be reclaimed and erased during garbage collection process. Therefore, the write operations of data migration from source server to destination server (see arrow 1) are *wasted* in the sense that the flash memory cannot perform in-place update. These *wasted* writes incur a considerable write amplification overhead inside flash devices and consequently causes more garbage collection and rapid wear of flash memory.

For example, M_{obj} objects are designed to move from A to B. Assume that each object is divided into n_{pages} flash pages. During data migration, the total number of wasted writes $W_{migration}$ is:

$$W_{migration} = M_{obj} \times n_{pages} \tag{4.1}$$

We define the erasure cost as $\frac{\mu}{1-\mu}$ according to [75] [44] [20], where μ is the utilization of a victim block that need to be cleaned during GC process. That is, the erasure cost is the amount of valid pages μ that need to be rewritten per victim block of new space claimed $(1-\mu)$. Let B_p be the number of pages per block. Each GC operation produces $B_p \times (1-\mu)$

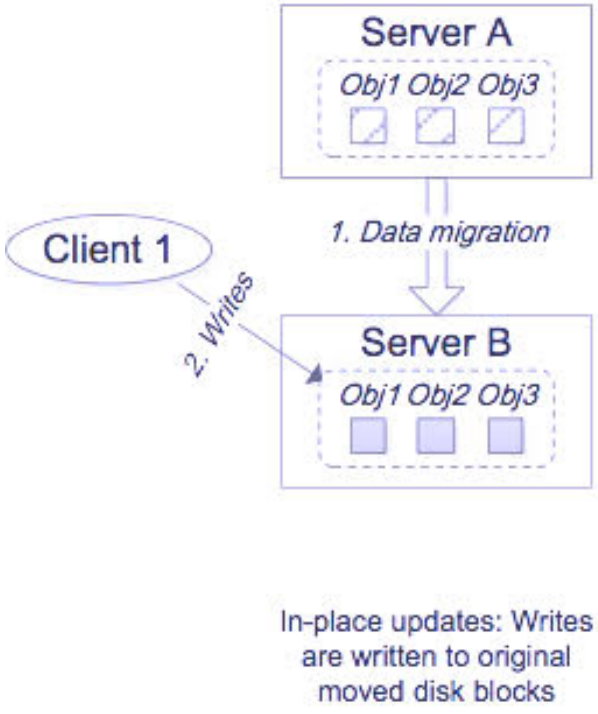


Figure 4.1: Data migration in disk-based storage servers

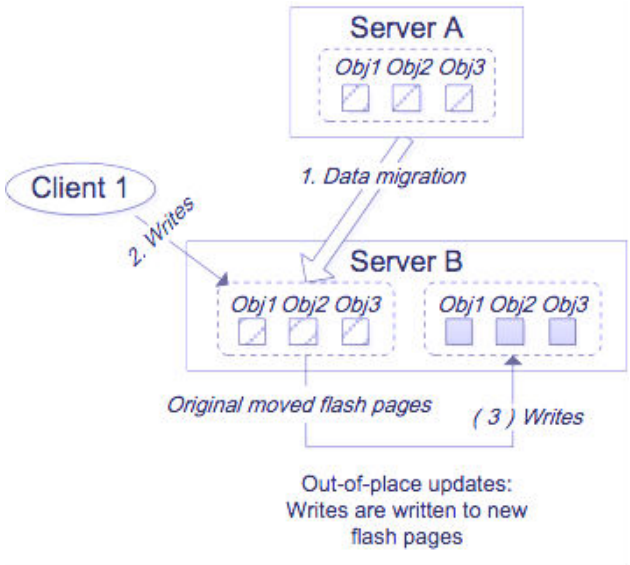


Figure 4.2: Data migration in flash-based storage servers

free pages. Then, after GC starts, the approximation of the block erase cycles caused by data migration $E_{migration}$ is:

$$\begin{aligned} E_{migration} &= \frac{W_{migration}}{B_p \times (1-\mu)} \\ &= \frac{M_{obj} \times n_{pages}}{B_p \times (1-\mu)} \end{aligned} \tag{4.2}$$

where $E_{migration}$ denotes the extra-wear caused by data migration. Extra-wear $E_{migration}$ increases with the number of data migrations.

Instead of moving data from source server to the destination server, write off-loading [60] [61] redirects the updates from source sever to the destination server. In this case, the *wasted* writes can be avoided. Then, for write off-loading, its wasted writes $W_{offloading} = 0$, and its extra-wear $E_{offloading} = 0$.

4.2 Wear balancing algorithm

The aim of wear balancing algorithm is to keep the variance in the *wear* (i.e. erase cycles) of flash servers to a minimum so that no single flash server wears out faster than others and the lifetime of flash memories associated with different servers can be improved.

EWO is based on a simple idea: re-distribute *writes* to flash servers based on the server's wear so that the erasures are evenly distributed among the flash servers. That is, a certain amount of writes are off-loaded to servers with a lower erasure count to ensure the overall standard deviation is within the threshold limit. Furthermore, EWO employs a hot-slice off-loading policy to explore the trade-offs among extra-wear cost, the amount of meta-data and the overhead of lookup operations.

4.2.1 Wear balancing

Consider that the server I/O workloads display unpredictable fluctuation with peaks and troughs. If the flash-based storage cluster is not provisioned for its peak I/O load, the write intensity during peaks leads to huge erase cycles. EWO must therefore periodically monitor the *wear* of flash servers and prevent the flash server with high erase cycles from further erasing.

EWO first tracks the wear periodically and then decides whether the wear balancing process should be triggered. We define the wear variance σ as the standard deviation of erase cycles. Wear balancing process begins when $\sigma > \sigma_c$, where σ_c is a wear variance threshold and indicates that the system has a significant wear imbalance.

For a given flash server i, if its wear (i.e. erase cycles) E_i satisfies $E_i - \overline{E_T} > \overline{E_T} \times \eta_c$, where η_c is a server wear variance threshold and $\overline{E_T}$ is the cluster-wide average wear, flash server i is denoted as a source server for write off-loading. Otherwise, it belongs to the destination server set of write off-loading.

Let W_t be the number of page writes to a flash server during a certain period denoted as *epoch_t*. B_p is the number of blocks per pages while μ is the utilization of the individual SSD. Then, after GC starts, the approximation of the erase cycles caused by W_t page writes is as follows:

$$E_t = \frac{W_t}{B_p \times (1 - \mu)} \tag{4.3}$$

The utilization of the individual SSD cannot be estimated directly as in most cases it is proprietary. However, previous works such as [63], [50], [57], [62],[74] does provide an empirical relation to calculate the utilization of the individual SSD from the average cluster utilization as long as the average utilization of the cluster is less than 85% . This is given by -

$$\mu_{avg} = \frac{\mu - 1}{ln\mu} + \sigma \tag{4.4}$$

The term σ, also referred to as the offset factor, is calculated empirically. The offset factor is calculated by taking into account multiple classes of traces and for our emulator it was found to be 20. Let F(μ) be the function which calculates μ from μ_{avg}, we can then estimate the wear of an SSD by combining 4.3 and 4.4 , leading to equation 4.5

$$E_c(W_c, \mu) = \frac{W_c}{B_p * (1 - F(\mu))} \tag{4.5}$$

Intuitively, wear balancing algorithm balances the wear by iteratively off-loading a certain amount of writes from the servers with highest wear to the server with lowest wear until the wear variance below the threshold σ_c. Figure 4.3 shows an example of wear balancing. The wear for each flash server is illustrated in figure 4.3a. As shown, server A has the highest erase cycles while server D has the lowest erase cycles. In this case, a certain mount of writes will be off-loaded from A to D. During each iteration k, the amount of writes that will be off-loaded ΔW_k can be calculated as follows:

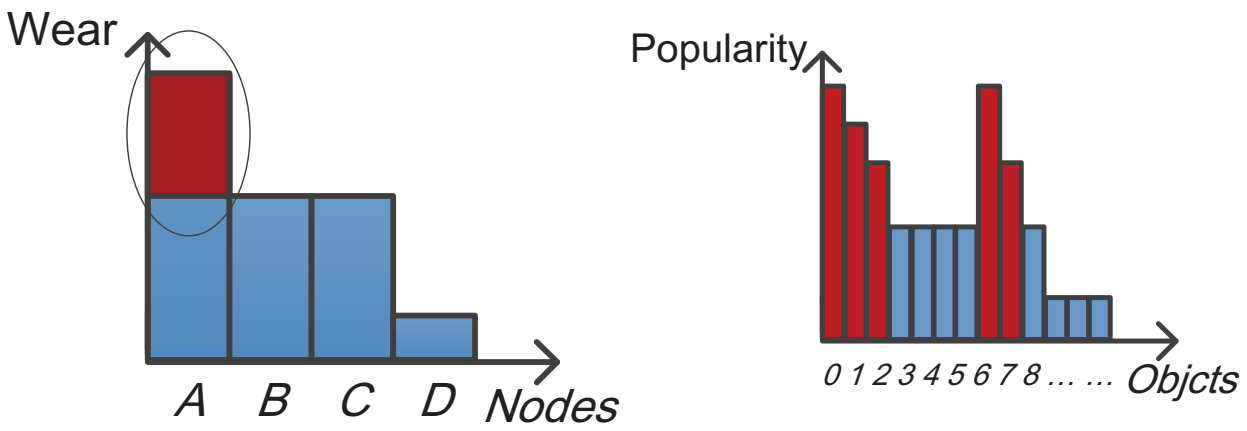
$$\begin{aligned} \Delta E_k &= E_{max\_k} - \overline{E_T} - \overline{E_T} \times \eta_c \\ \Delta W_k &= \Delta E_k \times B_p \times (1 - \mu) \end{aligned} \tag{4.6}$$

where E_{max_k} denotes the highest and lowest wear in the cluster for a given iteration k respectively.

4.2.2 Write off-loading

With the information of the amount of writes that will be off-loaded, we then consider what *writes* will be off-loaded during wear balancing process. Intuitively, since most writes access the *hot* objects, we can balance the wear by off-loading the writes to *hot* objects (i.e. write-frequently objects) from the servers with high wear to the servers with low wear, just as shown in figure 4.3b. Hot objects are marked as red. The writes to these hot objects will be off-loaded from server A to server D.

Previous work has shown that the object popularity can be calculated by weighting the write requests to the object over the time-line using an exponential decay function [63]. For



(a) Wear / Erase cycles for four flash servers

(b) Popularity of each object stored in server

Figure 4.3: Wear balancing

a given object i, the time-line which is from the time when object i is created to present is spitted into m + 1 epochs, $epoch\_0, \cdots, epoch\_m$. We define the popularity of each object as follows:

$$p_i = \sum_{j=0}^{m} \frac{w_j}{2^{m-j}} \tag{4.7}$$

Where w_j denotes the number of writes that access the object during an $epoch\_j$. p_i denotes the popularity of the object at the end of $epoch\_m$.

For a given object i, if its popularity p_i satisfies $p_i > \overline{p_T} \times \gamma_c$, where γ_c is a tunable threshold and $\overline{p_T}$ is the cluster-wide average object popularity, object i is denoted as a *hot* object. Otherwise, it is denoted as a *cold* object.

Although we can balance the wear by off-loading the writes to the *hottest* objects, the normal data access to write-frequently objects is significantly influenced. Typically, the meta-data concerning off-loaded objects is maintained as a map. It increases with the number of write off-loadings. Thus, it consumes a considerable time for querying the map for each read/write request and a huge memory for storing the map.

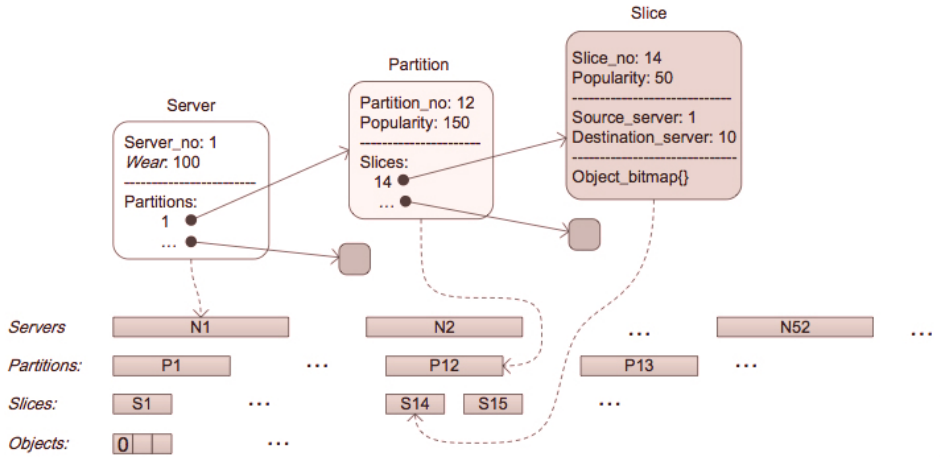


Figure 4.4: Slice and Off-loading Map (OM)

1. **Slice**

   EWO off-loads writes in number of *slices* instead of objects to mitigate meta-data overhead. A slice is a logically contiguous array of objects. Slices are accessed by random or sequential streams, which represents I/O access patterns. Slice size d_s is defined by the number of objects in the slice. $d_s \geq 1$. d_s is a configurable parameter.

   In most storage clusters, hash-based data placement scheme is commonly used to uniformly distribute data among storage servers [33] [68] [35], such as DHT and CHT. In distributed key value storage systems, objects are *hashed* on to a hash space, which is partitioned among storage servers. We consider a hash space divided into several partitions. We divide each hash partition into small, fixed-sized *slices* as shown in figure 4.4.

   For a given slice i, its containing objects are denoted as follows: $obj_{i \times d_s}, \cdots, obj_{(i+1) \times d_s - 1}$. The slice popularity is calculated by accumulating its containing objects popularity as follows:

$$P_s(i) = p_{i \times d_s} + p_{i \times d_s + 1} + \cdots + p_{(i+1) \times d_s - 1} \tag{4.8}$$

Let ϵ_i be the ratio of *hot* objects to *cold* objects in a slice i.

$$\epsilon_i = \frac{n_{hot}}{n_{cold}} \tag{4.9}$$

Where n_{hot} denotes the number of hot objects while n_{cold} denotes the number of cold objects in slice i.

For a given slice i, if ϵ_i satisfies $\epsilon_i > \epsilon_c$ and $P_s(i) > \overline{P_T} \times \rho_c$, where ϵ_c and ρ_c are two tunable thresholds while $\overline{P_T}$ is the cluster-wide average slice popularity, slice i is denoted as a hot slice. Otherwise, it is denoted as a cold slice.

2. **Off-loading Map (OM)**

   The writes are not stored at fixed locations because of write off-loading. The data locations are dynamically changed. Furthermore, different versions of the same data could be stored on multiple different locations because of write off-loading. Read request might access the data that is off-loaded. EWO must ensure that reads always go to the location holding the latest version of the data. Some reads will go to the source server while other reads will go to the destination server since its desire data is recently off-loaded.

   To maintain data consistency and reduce meta-data overhead, EWO provides two level of indirection for locating servers. For a given slice i, the first level indirection indicates its source location/server. While the second level indirection represents its destination location/server. All the meta-data of slices aggregates together into a slice list for data location as illustrated in figure 4.4. Besides, each slice meta-data contains a object bitmap that indicates whether the object is off-loaded recently.

   As shown in figure 4.4, EWO maintains three lists: server list, partition list and slice list. Server list consists of a number of server buckets. Typically, the storage system generates a unique identifier for each server when the server joins the storage cluster using common hash function. Each server bucket maintains the partitions this server is responsible for by using partition pointers. Additionally, each server bucket contains

the wear of its flash memories. Figure 4.4 summarizes the fields included in the server buckets. The meta-data of partition includes its identifier, popularity, the locations in the storage cluster and its containing slices. Each partition bucket maintains its slices by using slice pointers.

3. **Hot-slice off-loading (HSO)**

   Since EWO maintains two level of address indirection for each slice, all the data in the *to-be-off-loaded* slice must be eventually *off-loaded* to the destination server before next wear balancing process. For a given *to-be-off-loaded* slice i, it might contain some cold objects and there are no writes/updates to them. These cold objects must be eventually migrated to the destination server before next wear balancing process.

   As mentioned in Section 2, the more data written into a flash server, the more erase cycles it has. Apparently, migrating cold objects across different servers results in redundant cold object copies in the storage cluster and leads to wasted flash pages as well as extra-wear cost. Hot slice contains more write-frequently objects and lesser cold objects. To minimize extra-wear cost caused by cold data migration, EWO rebalances the wear by off-loading the writes to some *hottest* slices out from the flash servers with higher wear to lower ones. Consequently, the locations of the write-frequently slices are rearranged across cluster, which leads to a balanced wear across different flash servers.

   Algorithm 1 illustrates the process of hot-slice off-loading (HSO) and slice rearrangement. HSO starts when wear variance σ grows greater than a predefined threshold σ_c. Before starting wear balancing process, HSO first migrates the cold objects that are not off-loaded to its destination server by first consulting OM as shown in lines 3 to 14 of Algorithm 1. After all cold objects in a given *to-be-off-loaded* slice are migrated, HSO updates both the object meta-data and slice meta-data in OM as shown in line 15. Then, HSO starts slice rearrangement by off-loading some hottest slices out from the servers with highest erase cycles to the servers with lowest erase cycles by first quantitatively calculating the amount of writes based on the frequency of garbage

collection as shown in lines 17 to 29 of algorithm 1.

4.2.3 Off-loading writes

On a write request, the system performs a lookup for the request object identifier (OID) by consulting off-loading map (OM) and obtains two kinds of locations of the object: *source_server* and *destination_server* as shown in lines 1 to 2 of algorithm 2. If the request object is currently being off-loaded, specifically, the object falls into a to-be-off-loaded slice, the write request will be sent to *destination server* as shown in lines 3 to 6. Otherwise, the writes will be sent to *source server* as shown in line 8. Each write must be written/updated along with a version tag or time-stamp tag to maintain data consistency.

As mentioned, for some cold objects that falls into a to-be-off-loaded slice, it is possible that there is no writes or updates to them. In this case, these cold objects will be eventually migrated to destination server before next wear balancing process.

4.2.4 Off-loading reads

On a read request, the system first consults OM to find out if the request object is off-loaded as shown in lines 3 to 5 of algorithm 3. If the request object falls into a to-be-off-loaded slice and is currently off-loaded, the read request will be sent to its destination server as shown in lines 3 to 7. Otherwise, the request will be sent to its source server as shown in line 9.

4.3 Extra cost analysis

According to Section 3.1, the total number of wasted writes equals additional page writes caused by data migration. For EWO scheme, wasted writes can be avoided during write

Algorithm 1 Hot-slice off-loading

Require: $\sigma > \sigma_c$

Ensure: $\sigma <= \sigma_c$

```
1: // start hot-slice off-loading
2: // first, migrate cold data
3: for each slice s_i present in OM do
4:   if Get_slice_state(s_i) == OFFLOADING then
5:     for each object o_j present in object_bitmap_i do
6:       if Get_object_state (o_j) ! = OFFLOADED then
7:         // Migrate cold data and update OM
8:         Migrate_cold_object_to_destination(o_j)
9:         Update_object_state(o_j, OFFLOADED)
10:      end if
11:    end for
12:  end if
13:  Update_slice_state(s_i, COMPLETE)
14: end for
15: Update(OM)
16: // then, rearrange slices among servers
17: while k < iteration_steps do
18:   x ▷ extract server with max erase cycles
19:   y ▷ extract server with min erase cycles
20:   ΔW_k ▷ call Eq.4
21:   while ΔW_k > 0 do
22:     s_i ▷ Get_hottest_slice_from(x)
23:     // Remove s_i from x; Add s_i to y; Update OM.
24:     Modify_slice_destination(s_i, y)
25:     Modify_slice_source(s_i, x)
26:     ΔW_k = ΔW_k − P_s(s_i)
27:   end while
28:   k−−
29: end while
30: Update (OM)
```

Algorithm 2 Off-loading writes

```
1: L_s = Get_object_source(obj)
2: L_d = Get_object_destination(obj)
3: s_obj ▷ Get_slice(obj)
4: if Get_slice_state(s_obj) == OFFLOADING then
5:     Write_to_server(L_d, obj)
6:     Update_object_state(obj, OFFLOADED)
7: else
8:     Write_to_server(L_s, obj)
9: end if
```

Algorithm 3 Off-loading reads

```
1: L_s = Get_object_source(obj)
2: L_d = Get_object_destination(obj)
3: s_obj ▷ Get_slice(obj)
4: if Get_slice_state(s_obj) == OFFLOADING then
5:     if Get_object_state(obj) == OFFLOADED then
6:         Read_from_server(L_d, obj)
7:     end if
8: else
9:     Read_from_server(L_s, obj)
10: end if
```

off-loading. However, EWO involves cold data migration, which incurs extra-wear. EWO off-loads some hottest slices out from heavily loaded servers to lightly loaded servers. The hottest slices might contain several cold objects. These cold objects must be eventually migrated to their destination servers, which incurs extra-wear.

We first consider extra wear in the case of $d_s = 1$, where d_s denotes the number of objects in each slice. If $d_s = 1$, each slice contains a single object and EWO off-loads writes on the scale of objects. In this case, there is no cold data migration since EWO only off-loads writes to the hot objects. Thus, $W_{EWO} = 0$ and $E_{EWO} = 0$, where W_{EWO} denotes wasted writes caused by EWO and E_{EWO} denotes extra-wear caused by EWO.

We then consider extra wear in the case of $d_s > 1$. Assume that M_{slice} slices are designed to be off-loaded. Each object is divided into n_{papes} pages. We define $\bar{\epsilon}$ as the average hot/cold object ratio of these M_{slice} slices and ζ as the average cold object rate for these slices. Then, $\zeta = \frac{1}{\bar{\epsilon}+1}$. Average cold object rate ζ is greatly determined by slice size d_s, data placement scheme, and the workload I/O behavior.

The total number of wasted writes W_{EWO} is:

$$W_{EWO} = M_{slice} \times d_s \times n_{papes} \times \zeta \tag{4.10}$$

The extra-wear E_{EWO} is:

$$\begin{aligned} E_{EWO} &= \frac{W_{EWO}}{B_p \times (1-\mu)} \\ &= \frac{M_{slice} \times d_s \times n_{papes} \times \zeta}{B_p \times (1-\mu)} \end{aligned} \tag{4.11}$$

Slice size d_s is a key design parameter because it affects extra-wear cost and table size (The size of OM decreases with d_s). Since an object is the smallest unit of the address space, fine-grained write off-loading strategies with $d_s = 1$ can achieve a better wear balance without extra-wear cost. However this will lead to a larger table. On the other hand, EWO off-loads writes on the scale of slices with $d_s > 1$. With a large slice, OM can be small and fast. But off-loading large slices of objects to other servers may cause considerable extra wear.

Ideally, the slice size must be chosen carefully such that both extra-wear and OM is small. Practically, we set the slice size to 4KB empirically in our current implementation. Our results show that EWO with slice size is 4KB can achieve a good wear balance with small extra-wear and meta-data cost.

Chapter 5

Flashsim- A event-based SSD Simulator

Flashsim is an event driven simulator which is developed using object oriented programming models for modularity. The simulator is based on the block diagram as mentioned in 5.1

5.1 FlashSim Components

The simulator is written as a single-threaded program in C++. C++ provides a comprehensible object-oriented scheme where each class instance represents a hardware or software component. Classes are defined in Flashsim to model the hardware and the software components of an actual SSD.

5.1.1 Hardware Component Design

1. **SSD**: The SSD class serves to provide an interface to Disksim and provide a single class to instantiate in order 0to create the SSD simulator module. The SSD class creates

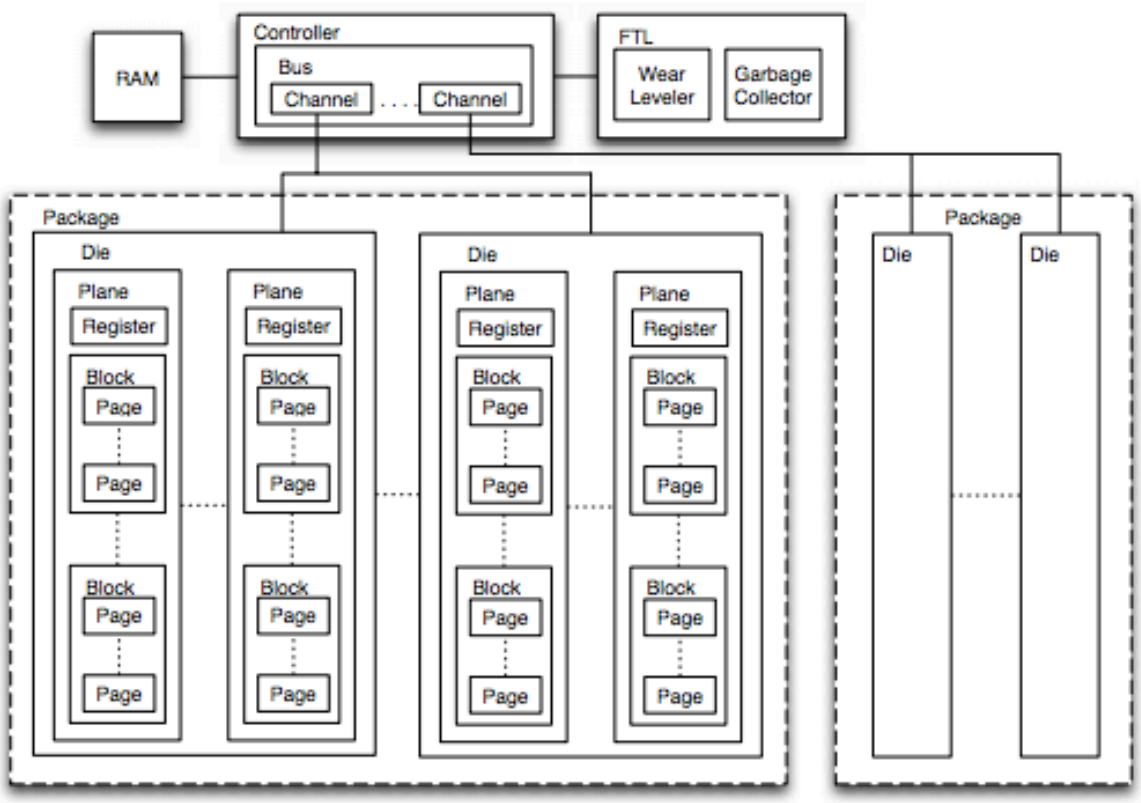


Figure 5.1: Flashsim architecture [45]

event objects to wrap the Disksim ioreq event structures and returns the event time to disksim.

2. **Channel**: Channels must schedule usage for events and update the event time values. Each channel keeps a scheduling table that keeps track of channel usage, and new events are scheduled at the next available free time slot after dependencies have been met. The scheduling table size is synonymous to queue length.

3. **Bus** : The bus class has a number of channels that are each shared by all the dies in a package. The bus examines addresses in events and passes the event object on to the proper channel.

4. **RAM** : The RAM class calculates how long it takes to read or write data to itself. The RAM buffers virtual event data for the controller to send across the bus.

5. **Controller** : The controller class receives event objects from the SSD and consults the FTL regarding how to handle each event. The controller sends the virtual data for events to the RAM for buffering before sending the event object to the bus.

6. **Page** : Each page maintains its state and updates event objects with the read and write delays of the given flash technology. Page states include free/empty after erasure, valid after a successful write, and invalid after being copied to a new location in a merge operation.

7. **Block** : A block is comprised of pages and is the smallest component that can be individually erased. When a block is erased, all pages in it are erased and can then be written to again. The corresponding event object is updated with the erase delay time. A block can only be erased a finite number of times because of reliability constraints.

8. **Plane** : Planes are comprised of blocks and provide a single page-sized register to buffer page data for bus transfers. The register is also used as a buffer for merge operations inside planes. The corresponding event object is updated with merge delays for merge operations and considers register delays.

9. **Die** : A die is a single flash chip organized into a set of planes. Dies are connected to bus channels, but individual planes contained in the die buffer bus transfers. In future development, the highest level at which merge operations may take place will be at the die level. The corresponding event object is updated with the merge delay time.

10. **Package** : The package class represents a group of flash dies that share a bus channel. The package class allocates its dies in its constructor and connects the dies to a bus channel. The package also facilitates addressing.

5.1.2 Software Component Design

1. **Event** : First, the event class keeps track of its corresponding Disksim ioreq[45] event structure. Second, the event class holds methods and attributes to do all the record-keeping for the SSD simulators state, including SSD addresses. Simulator objects pass event class objects and update the event objects statistics.

2. **Address**: Addresses are comprised of a separate field for each hardware address level from the package down to the page. We provide an address class instead of a struct to help make a clear interface to assign and validate addresses.

3. **FTL**: The FTL provides address translation from logical addresses to physical addresses. It determines how to process events that involve many pages by producing a list of single-page events to be processed in-order by the controller. The FTL is responsible for taking advantage of hardware parallelism for performance. The FTL also has a wear leveler and garbage collector to facilitate its tasks.

4. **Wear Leveler**: The wear leveler class helps spread the block erasures over all blocks in the SSD. The wear leveler is responsible for keeping as many blocks functional for as long as possible because blocks of pages can only be erased for reuse a finite number of times.

5. **Garbage Collector** : The garbage collector is activated when a write request cannot be satisfied because the selected block is not writable or there is not enough free space

in the selected block. The garbage collector seeks to merge partially-used blocks and free up blocks by erasing them. Any other algorithm for GC can also be simulated

Chapter 6

Implementation

Figure 6.1 shows the overall architecture of the EWO emulator comprising four modules: distributed managers, distributed metadata servers,flash server monitors, and client library. EWO serves as a wear-balanced distributed flash storage system, and performs wear balancing at configurable object granularity, e.g., at block or chunk level, or at KV pair level, etc.

6.1 Flashmanager

The flash manager consists of two sub-components, a wear balancer and a flash stats/ information collector. The wear balancer is responsible for balancing the wear of the whole cluster by offloading the writes based on the algorithm mentioned in section 4. To facilitate wear balancing, the EWO manager keeps track of the following information-

- Object's access history
- Flash utilization for each server

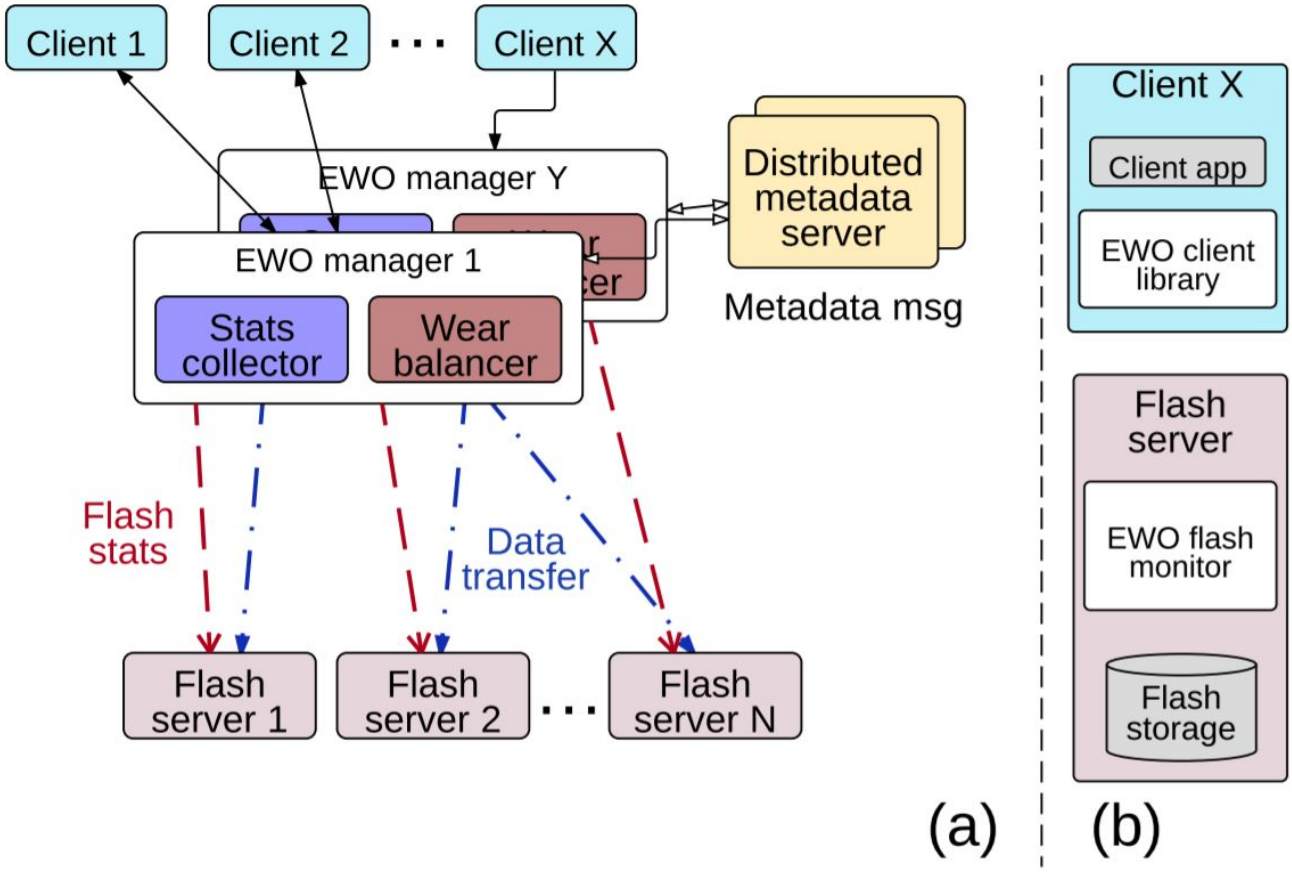


Figure 6.1: EWO Emulator Setup.

- Erase

Flash stats collector gathers information such as the flash space utilization and erasure count by exchanging heartbeat messages with the EWO flash monitor. Objects related metadata (objects state, popularity, and location) and flash stats information are stored in the metadata servers.

A client library provides a basic interface to read and write the data to the remote flash devices.

A flash server monitor is a daemon that runs on each flash server.It monitors the statistics of flash devices, and sends them to the wear balancer. Single-device wear leveling implementation is described in 3. In practice, there are two ways to estimate the wear (the total number of erase cycles) for SSD devices, one approach is to use a monitoring tool. For example, S.M.A.R.T [27] can detect various indicators of SSD reliability and lifetime. Another option is to estimate the wear during a time period by using the number of write requests and the device utilization, such as [63]. In our current implementation, EWO assumes the host has full control over garbage collection . The argument behind moving the functionality from a flashserver to the flashmanager is that the flashmanager has better knowledge (e.g., total erasure count) of the cluster metrics compared to the individual device FTL.

We implemented a prototype emulator in C++ which was close to about 8k lines of code [29]. The application being tested was a KV store which was built from scratch. The data is mapped to each individual flashservers by using a consistent flash-based data distribution algorithm. The flash function being used in our experiment is FVN-a1 [1]. Each of the traces that we use (i.e. YCSB, MSR, HPC) can be considered as consisting of multiple trace records. Each trace record maps to a logical object, which is represented using a unique object ID calculated using the FVN-a1 function. We chose the FVN-a1 function over other algorithms such as SHA-1 due to its speed. The logical data object is then stored in the appropriate server by using the consistent flash table. We use flashsim [42] to simulate the SSD behavior as flashsim can accurately show the block erase cycles.

Chapter 7

Evaluation

7.1 Experimental Methodology

We emulate a large flash array by running multiple instances of our flash simulator in virtual machines. Each virtualized instance emulates a flash server node. For most of our tests, we use an evaluation testbed with 50 flash server nodes. We also use a bigger 100-200 nodes flash server setup for tests with HPC workloads. We launch one EWO manager for every 10 flash servers. We determined this number by repeating the experiments with different settings.Each flash server node is equipped with one SSD that is simulated by Flashsim [46]. Table 1 summarizes the parameters that are commonly used to simulate SSD [69]. All the nodes are emulated on a small scale cluster where each node runs inside a separate Docker container [50]. The setup is connected with 10 Gbps network. For distributed metadata

Table 7.1: Trace details

Traces	*YCSB*	*usr*	*proj*	*src1*	*src2*	*stg*	*mds*	*HPC*
Total Requests	1232656	1871727	13794208	6616626	4936315	7277740	1299007	6615888
Amount R/W data (GB)	55.48	194.77	474.35	608.51	212.92	342.81	44.33	27.77
Write Ratio	81.123%	83.63%	13.80%	54.61%	71.93%	13.61%	63.85%	93.22%

Table 7.2: SSD parameters used in our tests.

Flash page size	4KB
Flash block size	256KB
Page read latency	25us
Page write latency	200us
Block erase latency	1.5ms
Over-provisioned space	15%

servers, we launch a 3-node Redis cluster.

Table 7.2 talks about the SSD parameters used in our study.

In this section, we evaluate EWO by comparing it with the baseline system without wear balancing scheme, and a conventional migration-based balancing scheme called EDM [63]. The I/O traces we used in our experiments are collected at the block device level from Microsoft Cambridge [14] as shown in table 7.1. We map the traces to servers by using a consistent hash-based data distribution algorithm for distributing data evenly to servers. The hash function used in our experiments is FVN-a1 [1], which is used by two popular distributed key value storage systems, Voldemort [5] and Sheepdog [2]. Each trace record maps to a logical object, which corresponds to a unique object ID calculated by using consistent hash function. The logical object is then stored in a certain server by consulting consistent hash table.

Our experimental testbed consists of 50 to 200 servers. Each storage server is equipped with only one SSD that is simulated by a flash simulator. We use a flash simulator [45] for the three schemes to accurately measure the block erase cycles. In all the experiments, object size is equal with block size where block is the unit of flash erasures. In the FTL configuration of flash simulator, we set the page size to 4KB.

In the next sections, We first measured the wear variance of the storage cluster under two wear balancing schemes in. We then evaluated the impact of the two wear balancing schemes

on flash lifetime in terms of flash endurance, extra writes, and extra erase cycles in Section . We also measured the impact of two wear balancing schemes on performance in Section 4.3. Finally, the meta-data size of the two wear balancing schemes is illustrated in Section 4.4.

7.2 Wear balance

To evaluate the wear variance of flash-based storage cluster, we calculate the standard deviation of the flash server erase cycles. The number of erase cycles for each server is calculated by aggregating the erase cycles performed by its flash devices. Figure 7.1 shows the results of using Baseline, EDM and EWO under six different workloads. First consider the results of Baseline without using wear balancing algorithm. These results show that Baseline had the highest wear variance among three schemes. The baseline's standard deviations were much larger than that of the two wear balancing schemes.

As shown in figure 7.1, although EDM did improve the deviation of block erases, EWO still outperformed EDM in most cases (*usr*, *proj*, *src1*, and *mds*). For example, the standard deviation for the EWO scheme was at most 93.7 under workload *usr* while the standard deviations were 141.2 and 412.9 for EDM and Baseline respectively as shown in figure 7.1. For the rest two workloads (*src2* and *stg*), EWO also delivered a comparable wear balance compared with EDM. In particular, its standard deviations were 140 and 414.6 under workloads *src2* and *stg*, respectively while that of the EDM were 138.4 and 410, respectively.

Figure 7.2 shows the standard deviation of write requests for the three schemes. Compared with figure 7.1, although the standard deviations of writes requests were much larger than that of erasure counts, the two kinds of standard deviation are positively related. Consequently, it makes sense to balance the wear among different flash servers by off-loading writes or migrating objects among them.

Figure 7.3 shows the standard deviations of erase cycles over 68 hours under the workload *usr* for Baseline, EWO and EDM. As shown, for baseline, the standard deviation grows with

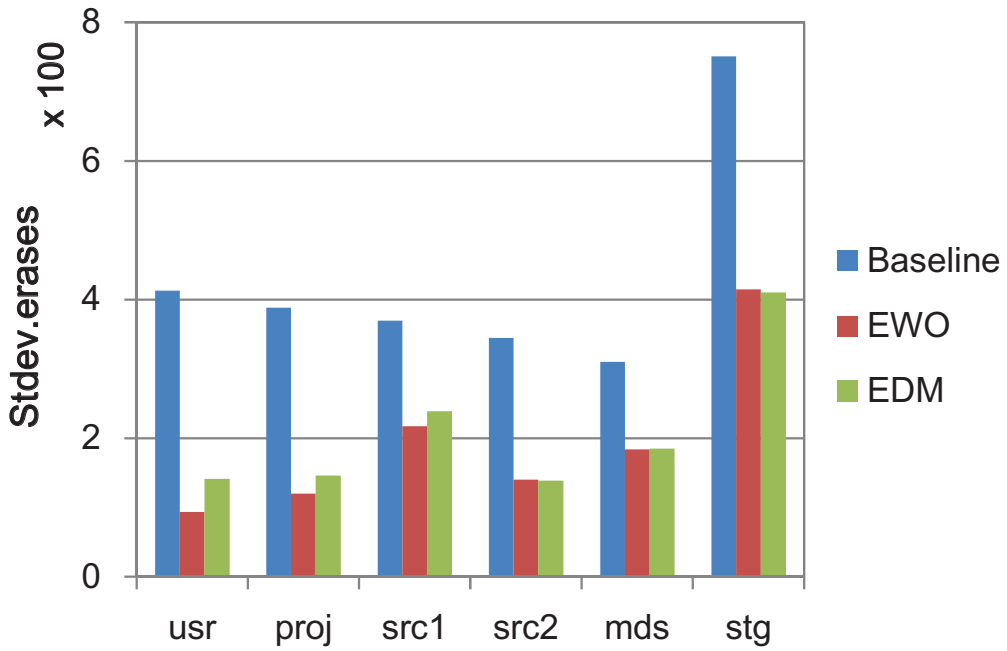


Figure 7.1: Standard deviations of block erase cycles for Baseline and two wear balancing schemes under six different workloads.

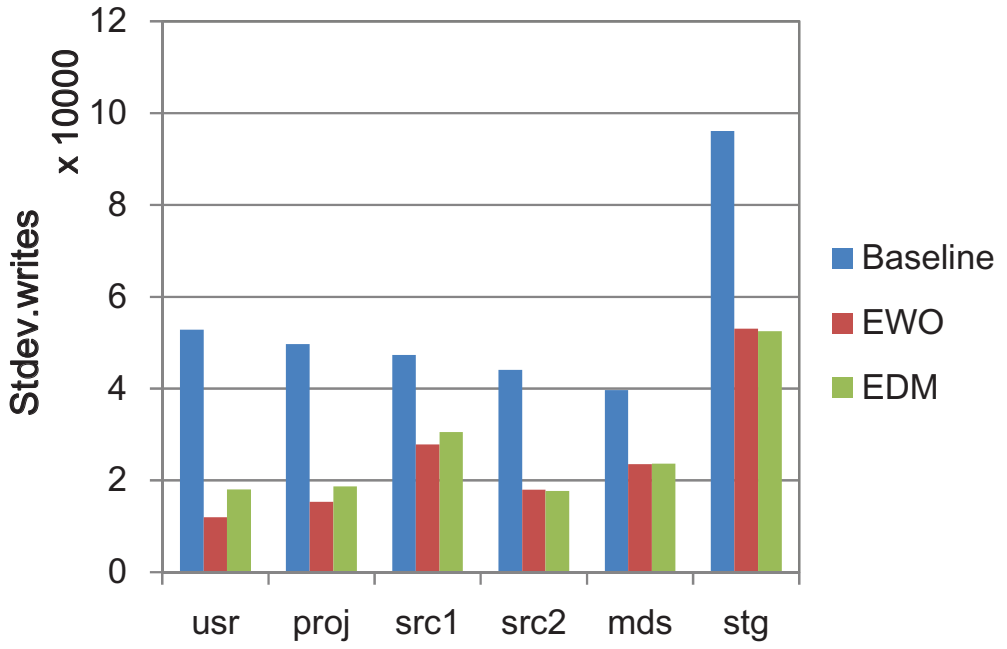


Figure 7.2: Standard deviations of write requests for Baseline and two wear balancing schemes under six different workloads.

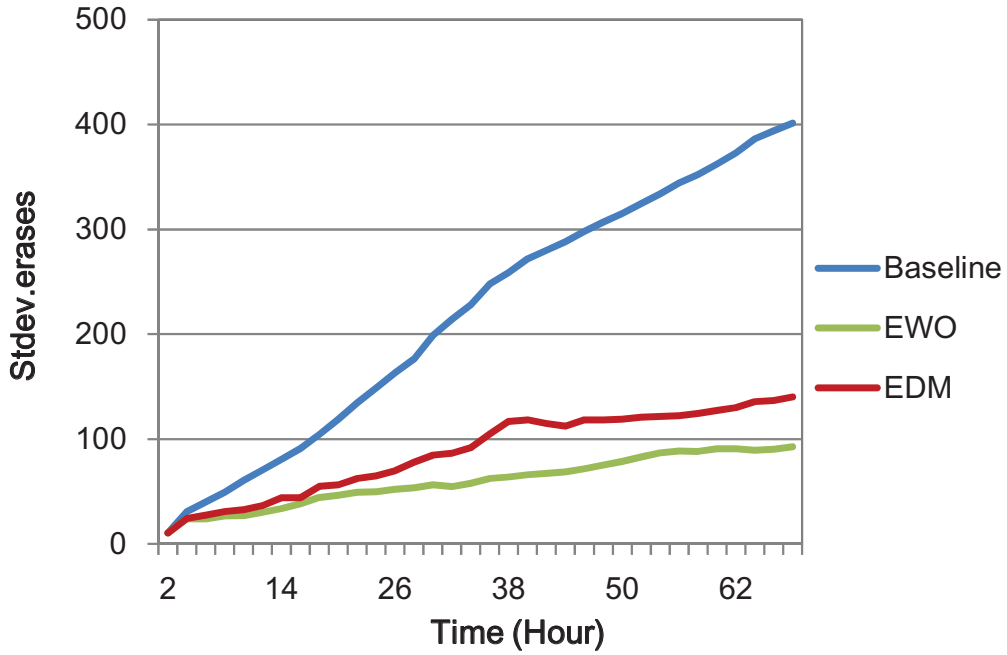


Figure 7.3: Standard deviations of erase cycles for 68 hours.

time. In contrast, for EWO and EDM, the standard deviations remain relatively lower. This is mainly because the wear variance among different flash servers is kept below a predefined tunable threshold by off-loading writes or migrating objects as shown in figure 7.3. The standard deviation of writes is positively related to that of erase cycles as shown in figure 7.4 and 7.3. Compared with EDM, EWO delivered the lower standard deviations of erase cycles under workload *usr*. This is because EDM introduced much more extra block erases (as shown in figure 7.5). The large extra write requests and block erases also affect the wear balance during wear balancing process.

7.2.1 Flash Lifetime

To evaluate the flash endurance, we calculate the aggregate erase cycles for all flash servers. The results are shown in figure 7.5. As shown, the aggregate erase cycles when replaying the workload *mds* is relatively lower than that when replaying others. This is because that workload *mds* has lower number of writes than other workloads as shown in Table 7.1.

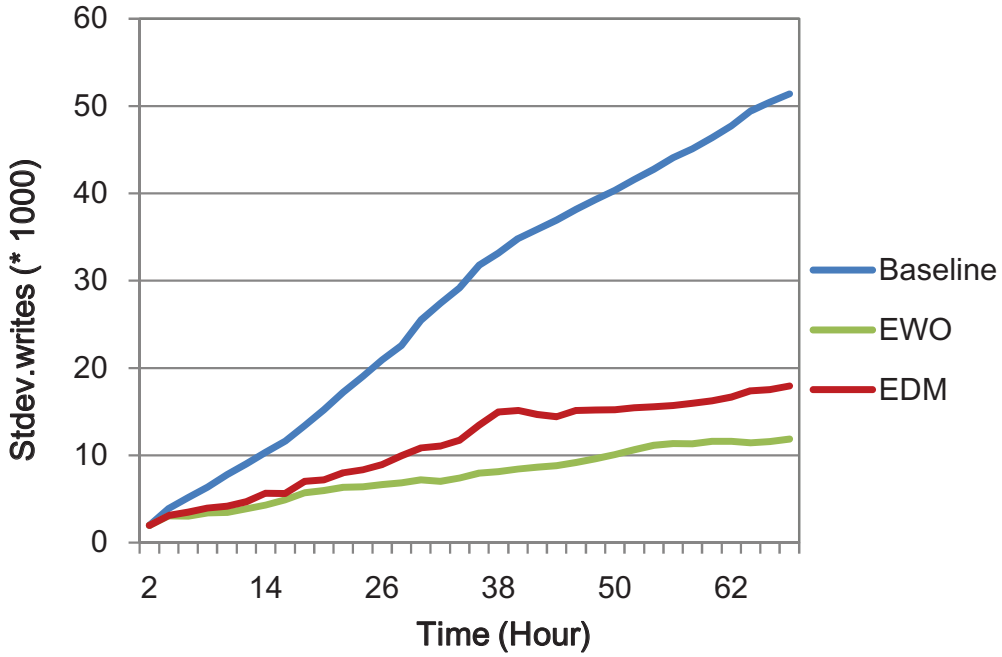


Figure 7.4: Standard deviations of write requests for 68 hours

Interestingly, comparing EWO with the baseline scheme, we observe that the EWO can reduce the cluster-wise aggregate block erase cycles under two I/O workloads: *src2* and *stg* compared with baseline. There are several reasons. First, EWO achieves a good wear balance distribution, which mitigates the overall write amplification due to garbage collection. For one hand, the utilization of hot flash servers is reduced due to endurance-ware write offloading. During write off-loading, the write requests are off-loaded from the servers with larger utilization to others. For the other hand, EWO introduces less writes to the destination servers compared with EDM as shown in figure 7.6. Consequently, the aggregate erase cycles can be reduced. As is shown in figure 7.7, we see that EWO can save up the aggregate block erases by up to 1% compared with baseline.

For EDM, the block migration process introduces significant wear overhead to overall cluster due to extra write overhead. Figure 7.6 shows the aggregate write requests among all flash server under the six different workloads. EDM delivered the highest aggregate write requests among the three schemes. High aggregate write requests introduce considerable wear overhead. As we can see in figure 7.7, the block erasure count is increased by up to 14% due to

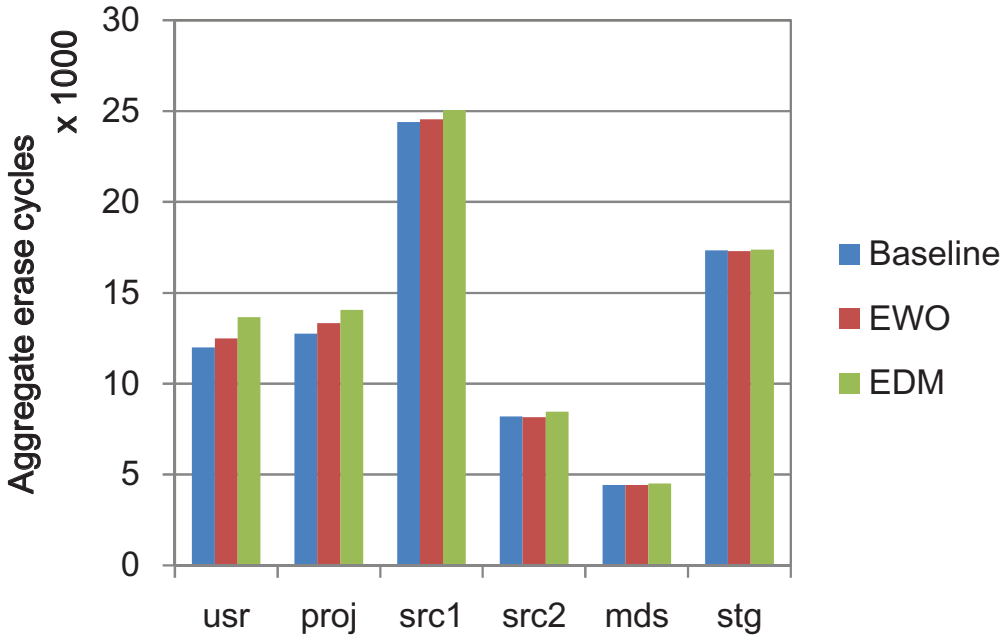


Figure 7.5: Aggregate erase cycles among all flash servers under six different workloads.

data migration.

Additionally, we can see that the aggregate erasure count is slightly increased in the rest cases (*usr*, *proj*, *src1* and *mds*) for EWO. This is mainly due to cold data migration. Since our OM table only maintains two level of address indirection in our experiment, all the to-be-off-loaded objects must be eventually *off-loaded/migrated* to the destination server before next write off-loading. Although EWO can reduce the wasted writes during wear balancing process by off-loading the updates to the destination servers, the wasted writes caused by cold data migration cannot be avoid. As shown in figure 7.8, EWO has different extra-write reduction when replaying different workloads compared with EDM. We observe that EWO can reduce the extra writes and extra wear by up to 79% compared with EDM.

Figure 7.7 and figure 7.8 show the extra erase cycles and extra write requests respectively. First, EDM had much larger extra erase cycles than EWO due to extra write overhead during data migration. Second, EWO reduced aggregate erase count under two workloads (*src2* and *stg*). We observe that EWO can reduce up to 1% erase cycles compared with baseline, and EWO can reduce the extra writes and extra wear by up to 79% compared with EDM.

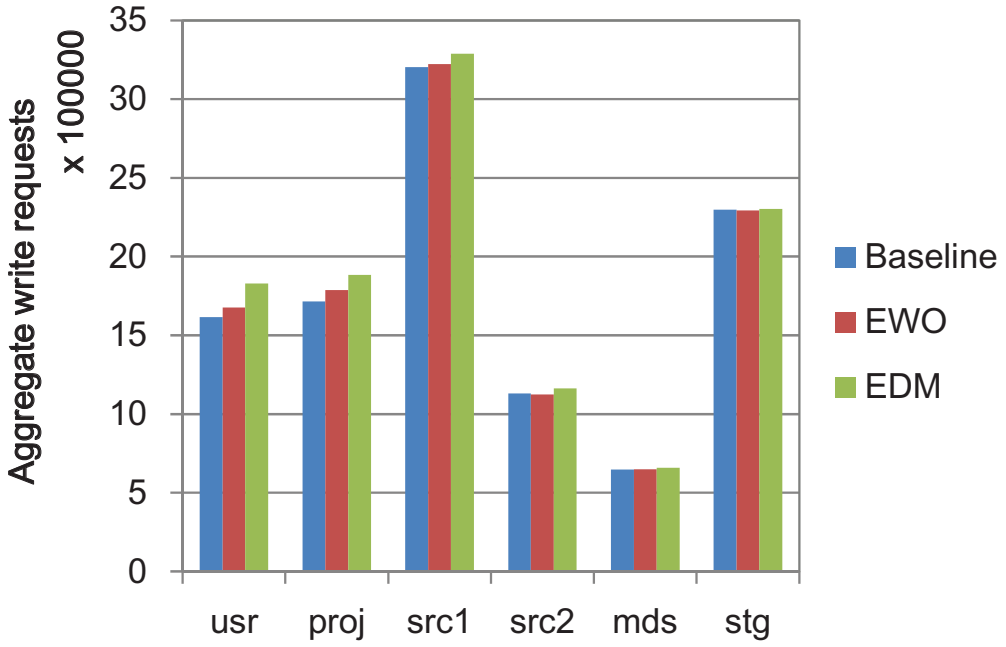


Figure 7.6: Aggregate write requests among all flash servers under six different workloads.

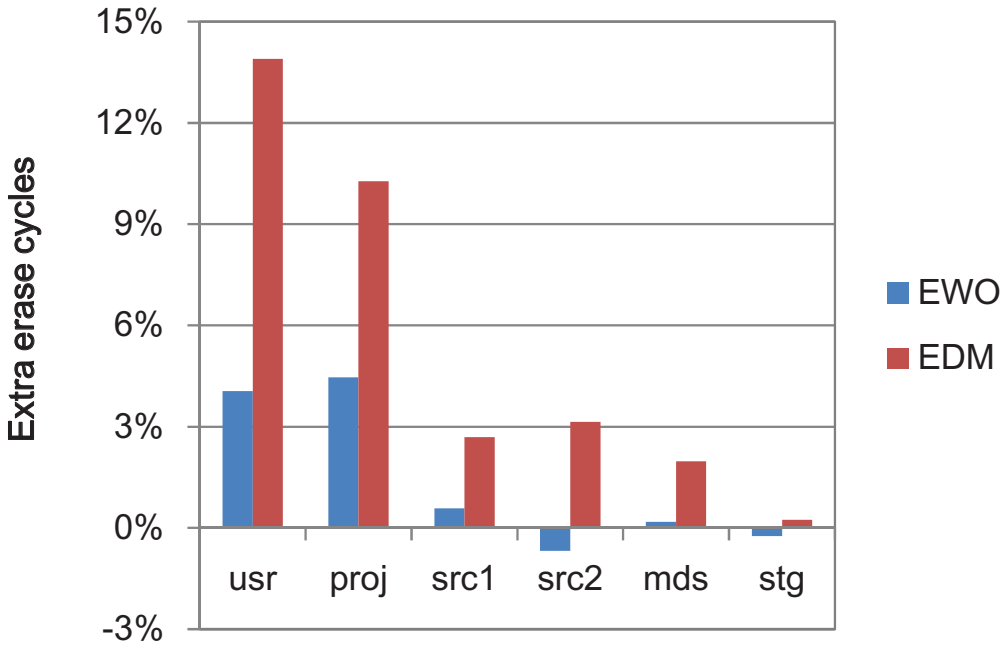


Figure 7.7: Aggregate extra-erase cycles of two wear-balancing schemes.

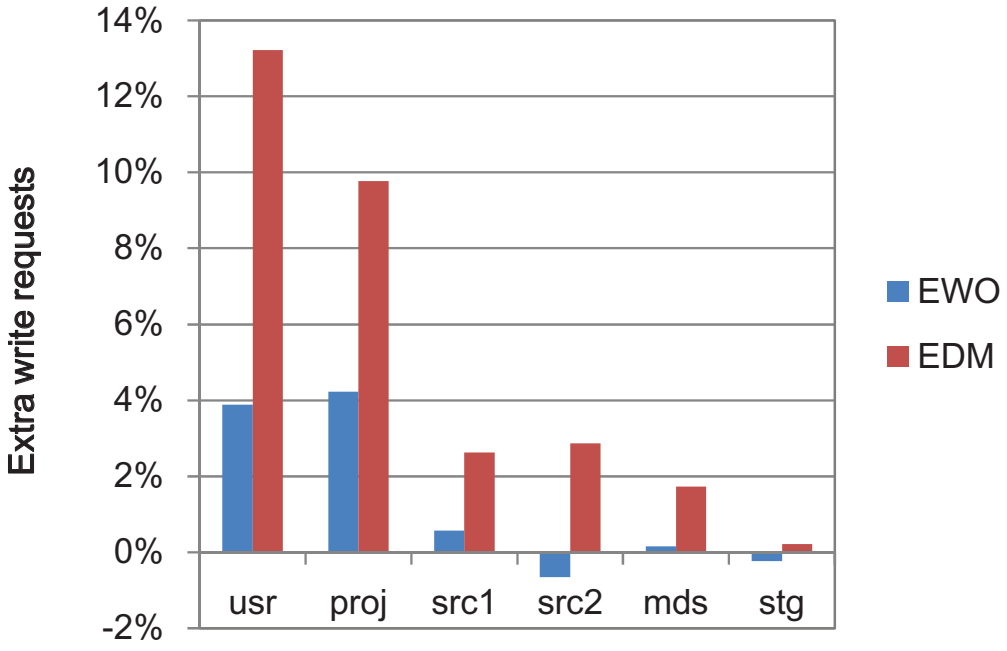


Figure 7.8: Aggregate extra-write requests of two wear-balancing schemes.

7.2.2 Performance

To evaluate the system performance, we measure the average response time under six different workloads for the three schemes. The results are shown in figure 7.9. As shown, the average response time when replaying the workload *mds* is relatively lower than that when replaying others. This is because that workload *mds* have lower number of writes than others as shown in Table 7.1. Moreover, the Baseline's average response time is the highest among the three schemes because of imbalance wear. For example, the average response time of Baseline is 1.46 ms while that of EDM and EWO is 0.96 ms and 0.94 ms respectively.

Compared with EDM, EWO has a better performance. This is because EWO can achieve a good wear balance with minimum extra overhead. In contrast, EDM introduce considerable extra overhead during wear balancing process.

Figure 7.10 shows the average response time over 68 hours of the workload *usr* for the three schemes. We can see that for the first four hours, all the three schemes had the same average

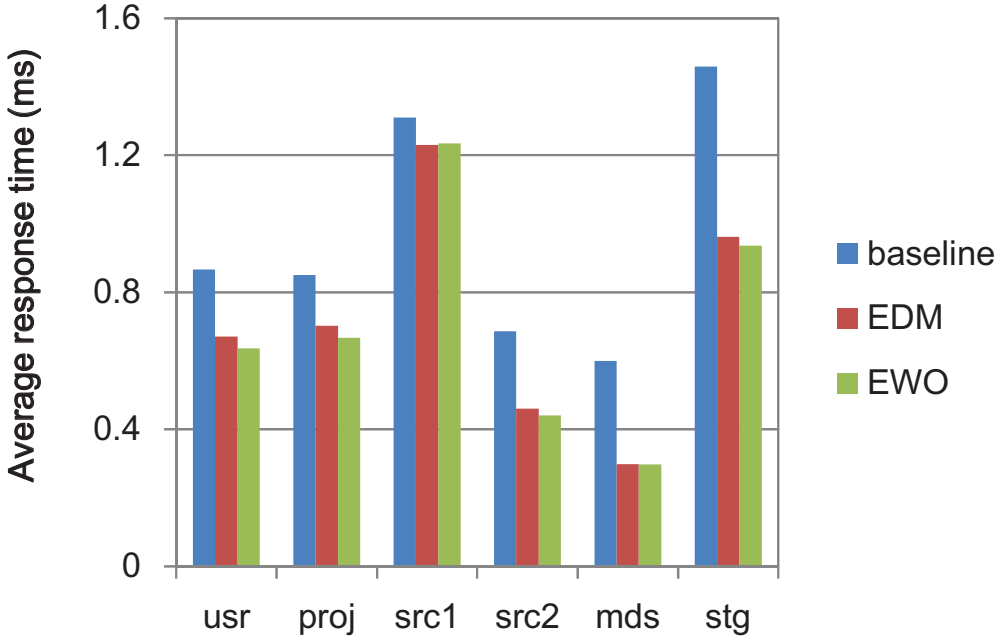


Figure 7.9: Latencies under six different workloads for Baseline, EWO and EDM.

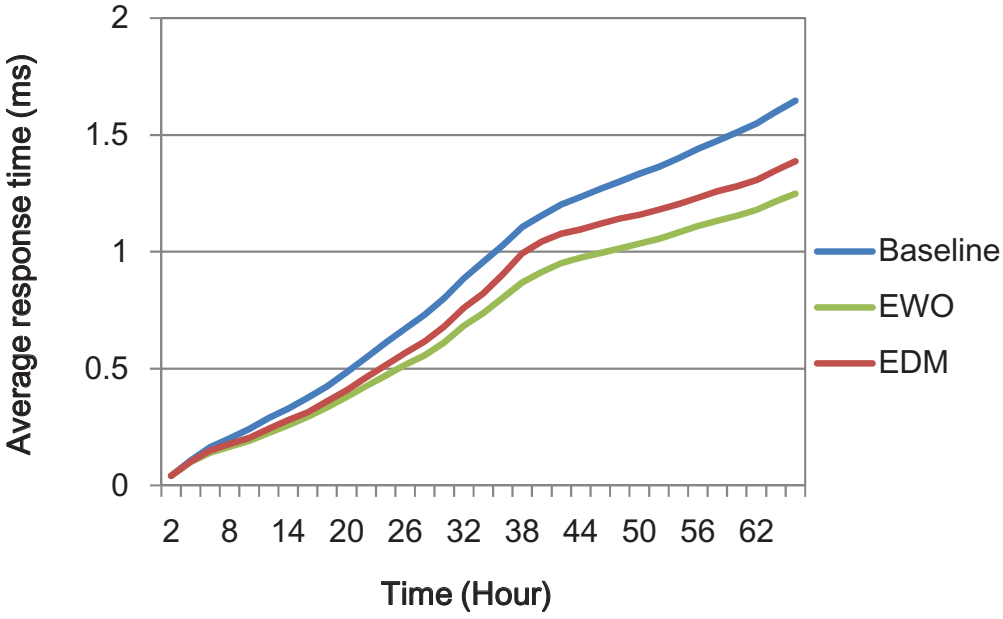


Figure 7.10: Latencies over 68 hours for Baseline, EWO and EDM under workload *usr*.

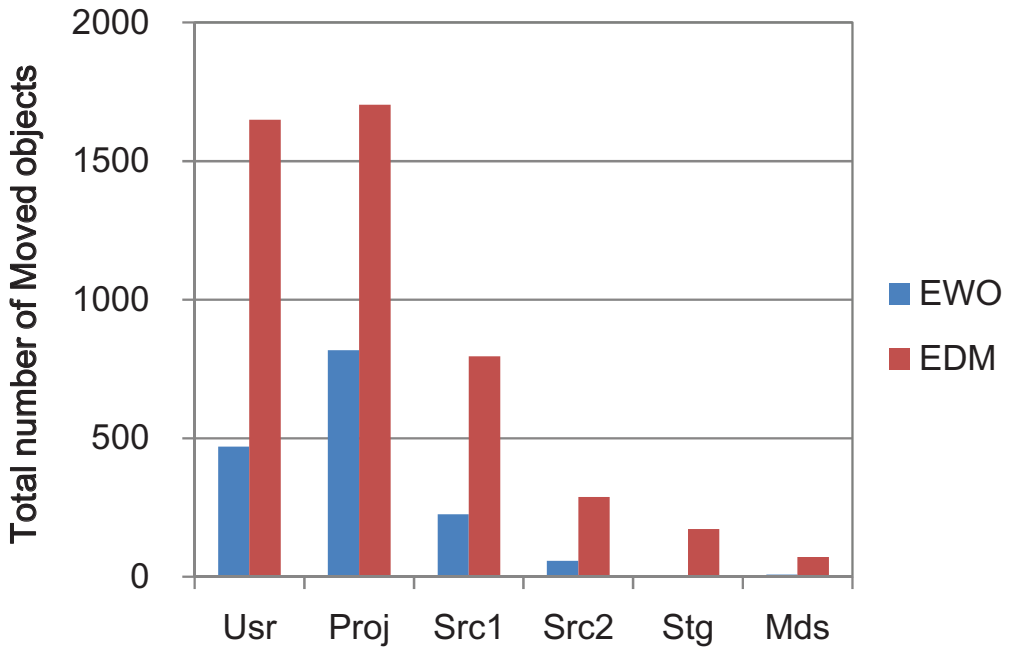


Figure 7.11: The total number of moved objects for two wear balancing schemes under six different workloads.

response time. After that, the average response time of EWO and EDM is shown to below that of Baseline. This is because Baseline has the largest wear variance among the three schemes because of unbalanced workload and the huge wear variance causes performance degradation. As shown, EWO has the smallest average response time among the three schemes because it has a better wear balance, which help improve the system load balance, as shown in Figure 7.9.

7.2.3 Analysis of Metadata overhead

To evaluate the metadata overhead, we first measure the total number of moved objects for two wear balancing schemes under the six workloads. The results are shown in Figure 7.11. As shown, we see that EDM had much larger number of moved objects than EWO. The total number of moved objects with EWO is relatively small. This is because that EWO can reduce considerable data migration by write off-loading. Moreover, EDM performs much

more data migrations in two cases, *usr* and *proj*. This results in more objects being moved during data migration for EDM. For EDM, the meta-data about mapping table increases with the number of moved objects. This illustrates that EDM introduces considerable meta-data overhead.

The meta-data for EWO is independent of the number of moved objects. EWO's mapping table, OM, is proportional to the number of accessed slices. In our experiments, the meta-data size in our experiment for EWO scheme is only approximately 50KB.

7.2.4 HPC Trace Results

Figure 7.12 shows the standard deviation of the erase counts for the HPC trace generated using the IOR benchmark. Further, we also carry out the scalability results for HPC by increasing the number from 50 to 200. We observed that EWO scales efficiently as the number of nodes goes on increasing. Further, as the number of nodes are increased, EWO seems to perform better making the case for it to be used in high-end computing systems such as Summit[4].

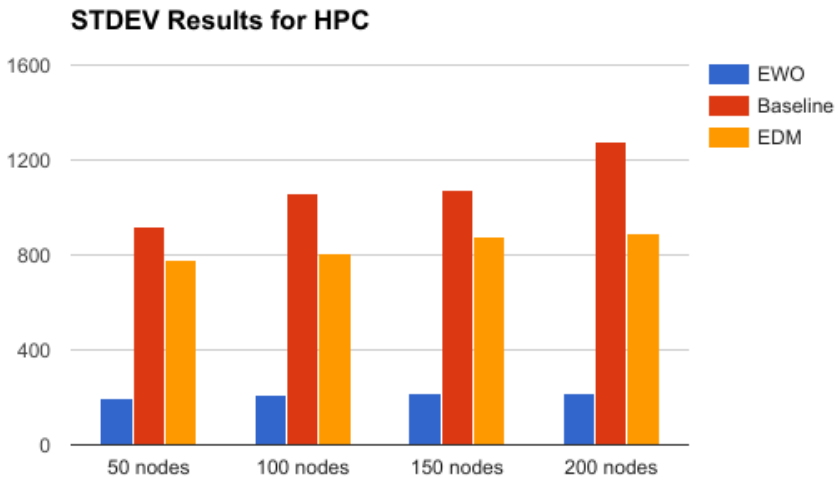


Figure 7.12: HPC Trace Results with increasing number of nodes

Chapter 8

Conclusion

This thesis analyzes the problem of additional wear overhead caused by data migration based wear balancing scheme. Although data migration-based wear balancing scheme can balance the wear of the flash-based storage cluster, the writes generated by data migrations are considered as an additional overhead and consequently cause considerable extra-wear of the whole cluster.

We proposed a practical technique called Endurance-aware Write Off-loading (EWO) for balancing the wear across different flash servers with minimal extra cost. Based on the observation that data migration based wear balancing technique generates considerable additional writes, which incurs significant extra wear, EWO are designed to exploit the out-of-place update feature of flash memory by off-loading the writes/updates across flash servers instead of moving data across flash servers to mitigate extra-wear cost. To evenly distribute erase cycles to flash servers, EWO off-loads writes out from the flash servers with high erase cycles to the ones with low erase cycles by first quantitatively calculating the amount of writes based on the frequency of garbage collection. To reduce meta-data overhead caused by write off-loading, EWO employs a hot-slice off-loading policy to explore the trade-offs among extra-wear cost and the meta-data overhead. Our evaluations show that EWO outperforms data migration based wear balancing technique, reducing considerable extra erase

cycles. In the meantime, EWO achieves a better wear balance and performance improvement compared with data migration based wear balancing technique in most workloads.

Chapter 9

Future Work

9.1 Analysis of the effect of data redundancy techniques on Endurance and Performance

Most systems store multiple copies of data to prevent hotspot creation [10]. Two of the most commonly used data redundancy techniques include replication and erasure coding. Replication is done in a k-way scheme where they are k replicas stored. Similarly, erasure coding involves striping the data and storing parity as well for the data. In our current implementation, we do not consider data redundancy policies as our focus is mainly on HPC systems where there is no data redundancy involved. However , most web servers employ these data redundancy techniques. The workload imbalance is amplified by order of magnitude in these systems . The reason is because storing redundant replicas of data would inevitably increase the severity of wear imbalance due to erasure oblivious data /load placement policies. Even in the case of erasure coding which adds CPU overhead both in terms of encoding/decoding processing and scattered network traffic (encoded stripes are stored at different locations of the storage cluster), thus imposes performance scalability issue. By storing replicas, the read performance can be increased as the load can be balanced

among each of the replicas. An hybrid approach seems to be the most intuitive solution. Data can be categorized as hot and cold based on the object popularity. Hot data, begin regularly accessed, is replicated while the cold data is erasure coded. As an when the object changes its state, a lazy migration scheme converts the object state from replication to erasure coding and vice versa.

9.2 Use active SSD's to deploy System Prototype in Hardware

The current implementation is built around an emulator where the flashmanager and meta-data servers are distributed and the client libraries. The biggest impediment to a prototype is the difficulty in estimating the erase counts for each individual server. As discussed before , either the SMART tool or other techniques can be used to estimate the erase counts.

9.3 Alternate methods to provide fault tolerance

The current implementation does not have the provision for system recovery in case one of the nodes goes down. Once redundancy techniques are incorporated into the algorithm, a comprehensive study can be done where nodes are allowed to fail randomly and the effect of recovery on erasure counts and write performance is evaluated.

Bibliography

[1] Fnv hash. http://www.isthe.com/chongo/tech/comp/fnv/.

[2] Home sheepdog/sheepdog wiki. https://github.com/sheepdog/sheepdog/wiki.

[3] Introduction to the EMC XtremIO Storage Array. https://www.emc.com/collateral/white-papers/h11752-intro-to-XtremIO-array-wp.pdf.

[4] Summit (ornl). https://www.olcf.ornl.gov/summit.

[5] Voldemort. http://www.project-voldemort.com/voldemort/.

[6] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 1–14. ACM, 2009.

[7] Ali Anwar, Andrzej Kochut Anca Sailer, Charles O. Schulz, Alla Segal, and Ali R. Butt. Scalable metering for an affordable it cloud service management. In *Proceedings of the IEEE International Conference on Cloud Engineering*, IC2E '15, Tempe, AZ, March 2015.

[8] Ali Anwar, Yue Cheng, Aayush Gupta, and Ali R. Butt. Taming the cloud object storage with mos. In *Proceedings of the 10th Parallel Data Storage Workshop*, PDSW '15, pages 7–12, New York, NY, USA, 2015. ACM.

[9] Ali Anwar, Yue Cheng, Aayush Gupta, and Ali R Butt. Mos: Workload-aware elasticity for cloud object stores. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 177–188. ACM, 2016.

[10] Ali Anwar, Yue Cheng, Hai Huang, and Ali R Butt. Clusteron: Building highly configurable and reusable clustered data services using simple data nodes. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*. USENIX Association, 2016.

[11] Ali Anwar, K. R. Krish, and Ali R. Butt. On the Use of Microservers in Supporting Hadoop Applications. In *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, Madrid, Spain, Sep 2014.

[12] Ali Anwar, Anca Sailer, Andrzej Kochut, and Ali R Butt. Anatomy of cloud monitoring and metering: A case study and open problems. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, page 6. ACM, 2015.

[13] Ali Anwar, Anca Sailer, Andrzej Kochut, Charles O Schulz, Alla Segal, and Ali R Butt. Cost-aware cloud metering with scalable service management infrastructure. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 285–292. IEEE, 2015.

[14] Storage Networking Industry Association et al. Msr cambridge traces, 2010.

[15] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[16] Mahesh Balakrishnan, Dahlia Malkhi, John D Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. Corfu: A distributed shared log. *ACM Transactions on Computer Systems (TOCS)*, 31(4):10, 2013.

[17] Simona Boboila and Peter Desnoyers. Write endurance in flash drives: Measurements and analysis. In *FAST*, pages 115–128, 2010.

[18] Adrian M Caulfield and Steven Swanson. Quicksan: a storage area network for fast, distributed, solid state disks. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 464–474. ACM, 2013.

[19] Li-Pin Chang and Tei-Wei Kuo. An efficient management scheme for large-scale flash-memory storage systems. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 862–868. ACM, 2004.

[20] Li-Pin Chang and Tei-Wei Kuo. An efficient management scheme for large-scale flash-memory storage systems. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 862–868. ACM, 2004.

[21] Yuan-Hao Chang, Jen-Wei Hsieh, and Tei-Wei Kuo. Improving flash wear-leveling by proactively moving static data. *IEEE Transactions on Computers*, 59(1):53–65, 2010.

[22] Feng Chen, Tian Luo, and Xiaodong Zhang. Caftl: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *FAST*, volume 11, 2011.

[23] Yue Cheng, Fred Douglis, Philip Shilane, Grant Wallace, Peter Desnoyers, and Kai Li. Erasing belady's limitations: In search of flash cache offline optimality. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 379–392, Denver, CO, 2016. USENIX Association.

[24] Yue Cheng, Aayush Gupta, and Ali R. Butt. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 4:1–4:16, New York, NY, USA, 2015. ACM.

[25] Yue Cheng, M. Safdar Iqbal, Aayush Gupta, and Ali R. Butt. Cast: Tiering storage for data analytics in the cloud. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 45–56, New York, NY, USA, 2015. ACM.

[26] Yue Cheng, M. Safdar Iqbal, Aayush Gupta, and Ali R. Butt. Pricing games for hybrid object stores in the cloud: Provider vs. tenant. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA, 2015. USENIX Association.

[27] Samsung Electronics Co. S.m.a.r.t. self-monitoring, analysis and reporting technology, 2014.

[28] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[29] Al Danial. Cloc–count lines of code. *Open source*, 2009.

[30] Biplob Debnath, Sudipta Sengupta, and Jin Li. Flashstore: high throughput persistent key-value store. *Proceedings of the VLDB Endowment*, 3(1-2):1414–1425, 2010.

[31] Biplob K Debnath, Sudipta Sengupta, and Jin Li. Chunkstash: Speeding up inline storage deduplication using flash memory. In *USENIX annual technical conference*, pages 1–16, 2010.

[32] Zhijie Feng, Zhiyong Feng, Xin Wang, Guozheng Rao, Yazhou Wei, and Zhiyuan Li. Hdstore: An ssd/hdd hybrid distributed storage scheme for large-scale data. In *International Conference on Web-Age Information Management*, pages 209–220. Springer, 2014.

[33] DeCandia Giueppe et al. Dynamo: Amazons highly available key-value store. *Retrieved September*, 9:2010, 2007.

[34] Kevin M Greenan, Darrell DE Long, Ethan L Miller, Thomas Schwarz, and Avani Wildani. Building flexible, fault-tolerant flash-based storage systems. In *The Fifth Workshop on Hot Topics in Dependability (HotDep09), Lisbon, Portugal*, 2009.

[35] Consistent Hashing and Random Trees. Distributed caching protocols for relieving hot spots on the world wide web. *David R. Karger and Eric Lehman and Frank Thomson*

Leighton and Rina Panigrahy and Matthew S. Levine and Daniel Lewin. STOC, 97:654–663, 1997.

[36] Stephen Herbein, Dong H. Ahn, Don Lipari, Thomas R.W. Scogland, Marc Stearman, Mark Grondona, Jim Garlick, Becky Springmeyer, and Michela Taufer. Scalable i/o-aware job scheduling for burst buffer enabled hpc clusters. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, pages 69–80, New York, NY, USA, 2016. ACM.

[37] Xavier Jimenez, David Novo, and Paolo Ienne. Phœnix: reviving mlc blocks as slc to extend nand flash devices lifetime. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 226–229. EDA Consortium, 2013.

[38] Xavier Jimenez, David Novo, and Paolo Ienne. Wear unleveling: improving nand flash lifetime by balancing page endurance. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 47–59, 2014.

[39] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, et al. Bluedbm: Distributed flash storage for big data analytics. *ACM Transactions on Computer Systems (TOCS)*, 34(3).

[40] Dawoon Jung, Yoon-Hee Chae, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A group-based wear-leveling algorithm for large-capacity flash memory storage systems. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 160–164. ACM, 2007.

[41] Myoungsoo Jung, Wonil Choi, John Shalf, and Mahmut Taylan Kandemir. Triple-a: A non-ssd based autonomic all-flash array for high performance storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 441–454, New York, NY, USA, 2014. ACM.

[42] Woon-Hak Kang, Sang-Won Lee, and Bongki Moon. Flash-based extended cache for higher throughput and faster recovery. *Proc. VLDB Endow.*, 5(11):1615–1626, July 2012.

[43] Taeho Kgil and Trevor Mudge. Flashcache: a nand flash memory file cache for low power web servers. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 103–112. ACM, 2006.

[44] Han-joon Kim and Sang-goo Lee. A new flash memory management for flash storage system. In *Computer Software and Applications Conference, 1999. COMPSAC'99. Proceedings. The Twenty-Third Annual International*, pages 284–289. IEEE, 1999.

[45] Youngjae Kim, Brendan Tauras, Aayush Gupta, and Bhuvan Urgaonkar. Flashsim: A simulator for nand flash-based solid-state drives. In *Advances in System Simulation, 2009. SIMUL'09. First International Conference on*, pages 125–131. IEEE, 2009.

[46] Youngjae Kim, Brendan Tauras, Aayush Gupta, and Bhuvan Urgaonkar. Flashsim: A simulator for nand flash-based solid-state drives. In *Advances in System Simulation, 2009. SIMUL'09. First International Conference on*, pages 125–131. IEEE, 2009.

[47] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 29:1–29:15, New York, NY, USA, 2016. ACM.

[48] K. R. Krish, Ali Anwar, and Ali R. Butt. hatS: A Heterogeneity-Aware Tiered Storage for Hadoop. In *The 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2014.

[49] K. R. Krish, Ali Anwar, and Ali R. Butt. Sched: A Heterogeneity-Aware Hadoop Workflow Scheduler. In *Proc. 22nd IEEE/ACM MASCOTS*, Paris, France, Sep. 2014.

[50] Hunki Kwon, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H Noh. Janus-ftl: finding the optimal point on the spectrum between page and block mapping schemes. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 169–178. ACM, 2010.

[51] Andrew W Leung, Shankar Pasupathy, Garth R Goodson, and Ethan L Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX annual technical conference*, volume 1, pages 2–5, 2008.

[52] William Loewe, T McLarty, and C Morrone. Ior benchmark, 2012.

[53] Chenyang Lu, Guillermo A Alvarez, and John Wilkes. Aqueduct: Online data migration with performance guarantees. In *FAST*, volume 2, page 21, 2002.

[54] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 257–270, 2013.

[55] Dongzhe Ma, Jianhua Feng, and Guoliang Li. Lazyftl: a page-level flash translation layer optimized for nand flash memory. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1–12. ACM, 2011.

[56] Souvik Mahapatra, Dipankar Saha, Dhanoop Varghese, and P Bharath Kumar. On the generation and recovery of interface traps in mosfets subjected to nbti, fn, and hci stress. *IEEE Transactions on Electron Devices*, 53(7):1583–1592, 2006.

[57] Jai Menon. A performance comparison of raid-5 and log-structured arrays. In *High Performance Distributed Computing, 1995., Proceedings of the Fourth IEEE International Symposium on*, pages 167–178. IEEE, 1995.

[58] Vidyabhushan Mohan, Taniya Siddiqua, Sudhanva Gurumurthi, and Mircea R Stan. How i learned to stop worrying and love flash endurance. *HotStorage*, 10:3–3, 2010.

[59] Muthukumar Murugan and David HC Du. Rejuvenator: A static wear leveling algorithm for nand flash memory with minimized overhead. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12. IEEE, 2011.

[60] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):10, 2008.

[61] Dushyanth Narayanan, Austin Donnelly, Eno Thereska, Sameh Elnikety, and Antony IT Rowstron. Everest: Scaling down peak loads through i/o off-loading. In *OSDI*, volume 8, pages 15–28, 2008.

[62] Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H Noh. Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems. In *FAST*, volume 12, 2012.

[63] Jiaxin Ou, Jiwu Shu, Youyou Lu, Letian Yi, and Wei Wang. Edm: An endurance-aware data migration scheme for load balancing in ssd storage clusters. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 787–796. IEEE, 2014.

[64] Mohit Saxena, Michael M Swift, and Yiying Zhang. Flashtier: a lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 267–280. ACM, 2012.

[65] Lei Shi, Zhimin Gu, Lin Wei, and Yun Shi. An applicative study of zipfs law on web cache. *International Journal of Information Technology*, 12(4):49–58, 2006.

[66] Hyogi Sim, Youngjae Kim, Sudharshan S Vazhkudai, Devesh Tiwari, Ali Anwar, Ali R Butt, and Lavanya Ramakrishnan. Analyzethis: an analysis workflow-aware storage system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 20. ACM, 2015.

[67] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending ssd lifetimes with disk-based write caches. In *FAST*, volume 10, pages 101–114, 2010.

[68] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[69] Guangyu Sun, Yongsoo Joo, Yibo Chen, Yiran Chen, and Yuan Xie. A hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement. In *Emerging Memory Technologies*, pages 51–77. Springer, 2014.

[70] Hong Tang, Aziz Gulbeden, Jingyu Zhou, William Strathearn, Tao Yang, and Lingkun Chu. A self-organizing storage cluster for parallel data-intensive applications. In *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, pages 52–52. IEEE, 2004.

[71] Kenton Varda. Protocol buffers: Googles data interchange format. *Google Open Source Blog, Available at least as early as Jul*, 2008.

[72] Chundong Wang and Weng-Fai Wong. Observational wear leveling: an efficient algorithm for flash memory management. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 235–242. IEEE, 2012.

[73] Wei Wang, Tao Xie, and Abhinav Sharma. Swans: An interdisk wear-leveling strategy for raid-0 structured ssd arrays. *ACM Transactions on Storage (TOS)*, 12(3):10, 2016.

[74] Wenguang Wang, Yanping Zhao, and Rick Bunt. Hylog: A high performance approach to managing disk layout. In *FAST*, volume 4, pages 145–158, 2004.

[75] Michael Wu and Willy Zwaenepoel. envy: a non-volatile, main memory storage system. In *ACM SIGOPS Operating Systems Review*, volume 28, pages 86–97. ACM, 1994.

[76] Aayush Gupta Yue Cheng, M. Safdar Iqbal and Ali R. Butt. Provider versus tenant pricing games for hybrid object stores in the cloud. *IEEE Internet Computing*, 20(3):28–35, 2016.

[77] Dongfang Zhao and Ioan Raicu. Hycache: A user-level caching middleware for distributed file systems. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1997–2006. IEEE, 2013.