Chameleon: An Adaptive Wear Balancer for Flash Clusters

Nannan Zhao, Ali Anwar, Yue Cheng†, Mohammed Salman, Daping Li⋆, Jiguang Wan⋆,
Changsheng Xie⋆, Xubin He‡, Feiyi Wang*, Ali R. Butt
Virginia Tech, †*George Mason University*, ⋆*Huazhong University of Science and Technology*,
‡*Temple University*, **Oak Ridge National Laboratory*

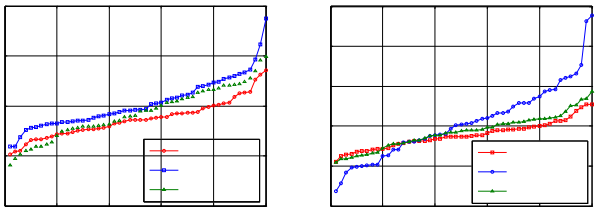
***Abstract*—NAND flash-based Solid State Devices (SSDs) offer the desirable features of high performance, energy efficiency, and fast growing capacity. Thus, the use of SSDs is increasing in distributed storage systems. A key obstacle in this context is that the natural unbalance in distributed I/O workloads can result in wear imbalance across the SSDs in a distributed setting. This, in turn can have significant impact on the reliability, performance, and lifetime of the storage deployment. Extant load balancers for storage systems do not consider SSD wear imbalance when placing data, as the main design goal of such balancers is to extract higher performance. Consequently, data migration is the only common technique for tackling wear imbalance, where existing data is moved from highly loaded servers to the least loaded ones.**

In this paper, we explore an innovative holistic approach, Chameleon, that employs data redundancy techniques such as replication and erasure-coding, coupled with endurance-aware write offloading, to mitigate wear level imbalance in distributed SSD-based storage. Chameleon aims to balance the wear among different flash servers while meeting desirable objectives of: extending life of flash servers; improving I/O performance; and avoiding bottlenecks. Evaluation with a 50 node SSD cluster shows that Chameleon reduces the wear distribution deviation by 81% while improving the write performance by up to 33%.

I. Introduction

Flash memory has emerged as a viable storage alternative for mobile computing devices due to its high throughput, persistence, and lower power consumption. The development of commodity flash devices such as solid state drives (SSDs) has also expanded flash memory's role in enterprise storage servers. All-flash or disk-free server storage systems (e.g., FlashStore [1] and Analyzethis [2]) are being developed. Flash-based storage servers that can play a significant role in accelerating application performance are clustered together and managed as a single entity for high reliability and availability in many distributed storage platforms such as FAWN [3], BlueDBM [4], QuickSAN [5] and CORFU [6].

Unlike magnetic disk drives, flash devices read and write data at the granularity of pages but *erase* data in units of a block.SSDs typically provide a flash translation layer (FTL) within the device to manage garbage collection (GC). If some valid pages are physically located in a block (called victim block) that has some invalid pages that need recycling, GC will first copy the valid data to a free page and then erase the victim block to make the block available for new writes. Consequently, a write operation can lead to multiple writes, resulting in *write amplification*. In this case, GC is time consuming (relative to read/write speeds) and also affects device endurance, as the number of sustainable erasures (P/E cycles) of a given block is limited.



(a) Erasure distribution under REP. (b) Erasure distribution under EC.

Fig. 1: Wear imbalance in a 50-server flash cluster. X-axis shows flash servers sorted by total erasure count.

To improve flash endurance and lifetime, FTL uses wear leveling (WL) to evenly wear all the flash blocks within a flash device so that no block will be worn out faster than others (detailed in Section II). However, the I/O workloads served by the flash based storage servers are imbalanced, which incur *wear imbalance among flash servers*. For example, Facebook's distributed key-value (KV) store workload analysis [7] reports high access skew and time varying workload patterns. The flash devices associated with heavily loaded servers serve more writes and perform more GCs, and thus wear out faster than others in a deployment. The maintenance of SSD devices raises many concerns. For instance, any maintenance required by the storage devices may require taking the entire flash server offline. This not only incurs administrative cost but also performance degradation, especially when SSDs are crucial, e.g., for burst buffer I/O nodes performance [8]. Non-uniform I/O workload, coupled with wear imbalance, also impact write performance because frequent GCs in flash servers with high utilization cause overall system slow down.

Moreover, wear imbalance worsens when fault tolerant or data redundancy schemes such as replication (REP) or erasure coding [9], [10], [11] (EC) are applied in a flash cluster. This is mainly because storing extra redundant data generates more writes, which in turn severely impact flash endurance.

Motivational Study. To quantify the impact of non-uniform write intensity on the SSD erasure count, we built a distributed

flash-based KV store that maps data to a 50-node cluster using consistent hashing [12]. Each node stores data locally in an SSD device that is simulated using FlashSim [13]. We applied two kinds of redundancy policies separately: REP with replication level $r = 3$, and EC with RS (6,4) encoding [14]. We measure total erasure count under YCSB workload [15] with Zipf-like access pattern ($YCSB - zipf$), and two block-level traces from MSR-Cambridge data center servers [16], namely, prn_0 and $proj\_0$.

Figures 1(a) and 1(b) show the sorted erasure count distribution under REP and EC, respectively. The largest erasure count is $4\times$ more than the smallest erasure count for $proj\_0$, and $3\times$ for both prn_0 and $YCSB - zipf$ under REP. Under EC, the largest erasure count is $12\times$ more than the smallest erasure count for $YCSB - zipf$, and $3\times$ for the other two workloads. These results show that the erasure counts are highly skewed among flash servers both under REP and EC schemes. Moreover, REP experiences almost $2\times$ more erasure counts than EC.

To address the challenge of balancing wear across flash servers, many researchers take inspiration from data migration [17], [18], [19]. For example, EDM [19] is a data migration-based wear balancing algorithm. It moves data from the flash servers with higher erasure count to the ones with lower erasure count for balancing the wear speeds. However, the extra writes generated by data migration create additional overhead, which incurs a considerable write amplification overhead and consequently causes more GCs and significant extra erasure count to the flash cluster. Moreover, the redundancy policies are completely ignored during wear balancing. However, we observed that the redundancy policies can provide useful information that can be leveraged to improve wear balance and flash lifetime, as well as performance.

Contributions. To solve the problems of multi-server wear imbalance, we propose a practical and efficient global wear balancing technique, Chameleon. Chameleon quickly detects the presence of erasure imbalance in a flash cluster. The goal is to balance the erasure count and improve both lifetime and performance of the flash cluster.

Specifically, this paper makes the following contributions:

- We exploit two redundancy policies–REP and EC–to help improve wear balance and flash lifetime, while also improving performance.
- We take advantages of the out-of-place update feature of flash memory by directly offloading the writes/updates across flash servers instead of moving data across flash servers to mitigate extra-wear cost, which includes late replicating (Late REP), late encoding (Late EC), and endurance aware write offloading (EWO).
- We provide two adaptive wear-balancing algorithms, redundancy policy transition (ARPT) and Hot/Cold data swapping (HCDS), coupled with write offloading and redundancy policies to balance the erasure count and improve both lifetime and performance of a flash cluster.
- We integrate our Chameleon emulator with a distributed flash-based KV store application. Emulation results on real-world workloads show that Chameleon outperforms the state-of-the-art data migration based wear balancing technique, reducing up to 81% wear variance while improving the write performance by up to 33%.

II. Related work

Flash endurance A large body of work has examined flash endurance [20], [21], [22], [23]. Techniques such as log-structured caching [23], inclusion of combining multiple bad blocks into virtual healthy blocks [22] have been explored to improve the lifetime of flash devices. These works are orthogonal and complementary to Chameleon.

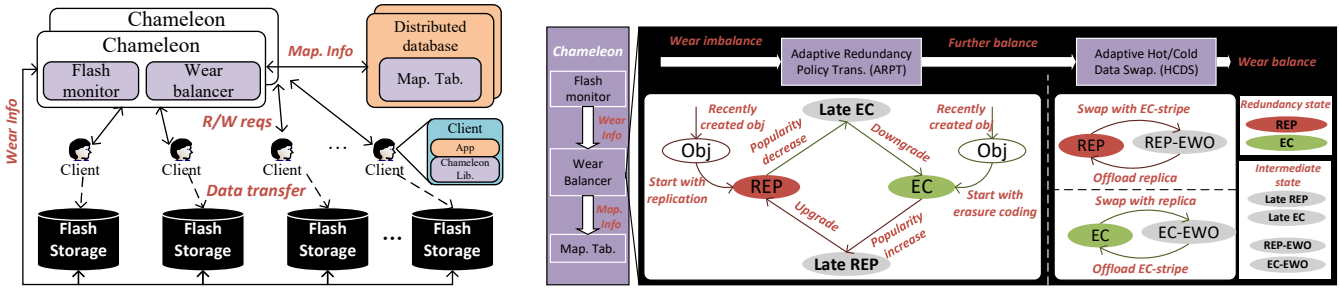
Intradisk wear leveling Dynamic [24], [25] techniques aim to achieve a good wear evenness while keeping the overhead low. Similarly, static wear leveling techniques [26], [27], [22], [28] move cold data to the blocks with higher erasure counts, thereby improving the even spread of wear. Chameleon leverages such approaches for extending the lifetime of individual SSDs in its target distributed setting.

Interdisk wear leveling Application of SSD arrays in enterprise data-intensive applications is growing. In such an environment, we have observed significant variance in number of writes and merge operations that are sent to individual SSDs. Recent work [29] 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 [30] also targets SSD arrays but use data migration to achieve wear balance across the SSDs in the array. SWANS [31] 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 Chameleon the goal of wear leveling across an SSD array, however unlike them Chameleon considers the role of redundant policies at various storage hierarchy and their impact on overall wear balancing.

Distributed flash storage systems FAWN [3] 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. CORFU [6] 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. Other works [32], [33], [34], [35] focuses on tiered storage to reduce the load on flash devices. Similarly, [36], [37] use data partitioning to evenly distribute load. In contrast, Chameleon focuses on EC storage solutions, which offer higher storage efficiency and exploits the interactions between the storage hierarchy to improve overall flash lifetime in flash-based clusters.

III. Design of Chameleon

Chameleon is aimed to addresses challenges arising from modern I/O workloads that exhibit high skewness across distributed flash servers. If a flash cluster does not implement



(a) System architecture.

(b) Adaptive wear balancing algorithms: ARPT and HCDS.

Fig. 2: Chameleon system.

server-level wear balancing, a flash server with high write intensity can have huge erasure counts and be worn out, while other flash servers are underutilized. This uneven use will trigger degraded performance, and eventual maintenance and downtime that would affect overall system performance and availability.

Chameleon is designed to balance the wear among different flash servers with the goal of: (1) reducing the unnecessary administration cost of replacing worn-out or failed flash devices; (2) improving the average lifespan of all flash devices; and (3) improving performance especially for write operations.

We first describe the framework of Chameleon in Section III-A followed by two adaptive wear-balancing algorithms detailed in Section III-B. The mapping table is discussed in Section III-C.

A. Overview of Chameleon architecture

Figure 2(a) shows an overview of Chameleon architecture comprising four modules: flash monitor, wear balancer, mapping table, and client library. Instances of Flash monitor and wear balancer are distributed across the flash-based storage servers (called flash servers). These components monitor and balance the wear of the whole cluster. Chameleon keeps track of objects related metadata (e.g., location and popularity) and stores the metadata in a distributed database as a distributed mapping table. The use of distributed database helps Chameleon scale as needed. The Client library provides a basic interface for users to read and write data to flash servers. Read or write requests are sent to Chameleon instances that determine the location of the flash devices for serving the requests.

Flash monitor monitors the statistics (i.e. , erasure count and flash space utilization) of flash devices, and sends them to the wear balancer. In our current implementation, Chameleon assumes the host has full control over garbage collection (GC) as provided by open-channel SSDs [38]. The argument behind moving GC management from the flash to the host is that the host has better overall knowledge (e.g., total erasure count) that offers both better performance and more optimization opportunities, compared to the individual device FTL [39]. We focus on open-channel SSDs, as more and more of such devices are being introduced to the market, e.g., LightNVM [40]. Thus, by focusing on open-channel SSDs we can optimize both the current available components, as well as a growing number of components that will become available in the near future.

Wear balancer is responsible for balancing the wear of the whole cluster. Balancer has two major components as shown in Figure 2(b): (1) An adaptive redundancy policy transition (ARPT) module that dynamically converts data redundancy and adapts to workload changes for balancing the wear while ensuring good performance and low erasure overhead. (2) A data swapping module that swaps data between the servers with higher erasure counts and servers with lower erasure counts to further improve wear balance.

Mapping table keeps track of the updates made to the metadata when data object's addresses and redundancy policies are changed during the balancing process. The table stores objects' metadata, such as object's state, popularity, and location. It also keeps track of the object access history (i.e. , popularity) to facilitate wear balancing. Mapping table is kept in a distributed database to avoid memory overhead.

B. Adaptive wear balancing algorithms

In this section, we describe two wear balancing algorithms used by Chameleon: (1) Adaptive redundancy policy transition (ARPT) algorithm, and (2) Hot/cold data swapping (HCDS) algorithm.

ARPT adopts a hot/cold data segregation approach by leveraging: (1) REP to store a small fraction of mostly frequently updated data (write hot data) to provide overall low I/O latency for the system; and (2) EC to encode all the remaining relatively cold data to realize a low erasure overhead. Moreover, ARPT dynamically adapts to workload changes by using late-REP or late-EC (§III-B1) to switch data state between two redundancy schemes and remap data for balancing the wear of whole cluster with low overhead. Furthermore, HCDS is used to swap hot and cold data between servers with higher erasure counts with the ones with lower erasure counts to further improve the wear balance.

1) Adaptive redundancy policy transition (ARPT): Chameleon tracks erasure counts to decide when the wear balancing process should be triggered. We define the wear variance σ as the standard deviation of erasure counts. If the system develops significant wear imbalance—indicated by $\sigma > \sigma_{ARPT}$, where σ_{ARPT} is a preset wear variance threshold

Algorithm 1: Adaptive redundancy policy transition.

Input: σ: cluster wear variance, σ_{ARPT}: wear variance threshold, ℓ_{hot}: object popularity threshold
Require: $\sigma > \sigma_{ARPT}$
Ensure: $\sigma <= \sigma_{c_{ARPT}}$
// *Step 1, Screen candidates from each server*
for each object obj_i that is in flash cluster **do**
  if obj_popularity(obj_i) >= ℓ_{hot} && obj_i *is neither in REP nor late-REP state* **then**
    //Convert its'redundancy scheme to late-REP
    Convert_object_state(obj_i, late-REP)
  end if
  if obj_popularity(obj_i) < ℓ_{hot} && obj_i *is neither in EC nor late-EC state* **then**
    //Convert its'redundancy scheme to late-EC
    Convert _object_state(obj_i, late-EC)
  end if
end for
// *Step 2, Rearrange candidates among nodes*
while $\sigma > \sigma_{ARPT}$ **do**
  $X(x_1, x_2, x_3)$ ▷ extract servers with MIN erasure counts
  $Y(y_1, y_2, y_3, y4, y5, y6)$ ▷ extract servers with MAX erasure counts
  obj_i ▷ Get_hottest_candidate (from step 1)
  obj_j ▷ Get_coldes_candidate (from step 1)
  Map_object_to(obj_i, X)
  Map_object_to(obj_j, Y)
  σ ▷ Estimate wear variance
end while

(Table I)—the balancing process is triggered periodically until the wear variance drops below the threshold.

Moreover, Chameleon also records the object write heat changes. Each object is classified as either hot or cold based on their write heat changes and the object's state switches between REP and EC.

Chameleon performs a periodic scan through all the replicated data for "cooled down" data and convert such data's redundancy policy from REP to EC, a process denoted as **downgrade**. Similarly, encoded data is also scanned for new hot data and these new hot data's redundancy policy is switched from EC to REP, denoted as **upgrade**.

Late-EC & Late-REP The additional erasure count caused during downgrade/upgrade operations is nontrivial. The downgrade operation requires network transfers of the replicated objects from different locations to encode them into RS code, along with invalidation of the old replicated objects. Upgrade operation needs to retrieve the data stripes from different locations to k-way replicate them and invalidate the old stripes. Both downgrade and upgrade operations will incur network overhead and extra erasure cycles. To mitigate this, Chameleon implements two additional optimizations, late-REP or late-EC, to support downgrade/upgrade with low overhead. Here, the conversion due to upgrade and downgrade are delayed until the next update, which not only reduces conversion overhead but also avoids unnecessary conversions, such as, a downgrade followed by an upgrade for the same data.

Downgrade/upgrade operations are delayed as long as the wear variance remains tolerable. The late policies trades-off the probability of wear imbalance with network traffic overhead. To do this, we exploit the out-of-place update feature of flash memory by delaying the redundancy policy transition until clients issue the write/update requests to the objects whose redundancy policies need to be converted. Then, we directly convert the requested data into the desired redundancy policy state (replicas or EC stripes) and re-distributes them to their respective destinations. Thus, the network traffic overhead can be reduced and the number of extra writes during redundancy transition process are mitigated.

TABLE I: Terminology & List of Acronyms.

Acronym	Description
REP	Replication
EC	Erasure coding
ARPT	Adaptive redundancy policy transition
HCDS	Hot cold data swapping
EWO	Endurance aware write offloading
Downgrade	Conversion from REP to EC
Upgrade	Conversion from EC to REP
Late-REP	Late replicating
Late-EC	Late erasure coding
σ	Standard deviation of erasure counts
σ_{ARPT}	Wear variance threshold that triggers ARPT
ℓ_{hot}	Popularity threshold
w_j	Number of writes to the object during epoch j.
p_k	Write heat of the object at the end of epoch *k*
μ	Utilization of a victim block that needs to be cleaned
B_p	Number of pages per block
W_t	Number of page writes during a certain epoch *t*.
E_t	Block erasure counts during epoch *t*
σ_{HCDS}	Wear variance threshold that triggers HCDS

As shown in Figure 2(b), we define two kinds of states for objects: redundancy states which contain REP and EC, and intermediate states that contain late REP, late EC, REP-EWO, and EC-EWO (detailed in III-B2). Figure 2(b) shows the redundancy policy transition of an object. As the write heat of an object increases, the object either stays in REP state or is converted from EC to late REP state by ARPT. The object stays in the late REP state until the next write/update is received and the state is changed to REP. Similarly, if the write heat of an object decreases, its state either stays in EC or is converted to late EC state if the current state is not EC. The state will be eventually converted to an encoded state upon next write/update.

Specifically, if object obj_i's popularity is greater than a predefined threshold (ℓ_{hot}) denoted as **hot object**, and its state is neither REP nor late-REP, obj_i's state will be converted to late-REP. In contrast, if object obj_i's popularity is smaller than ℓ_{hot} denoted as **cold object**, and its state is neither EC nor late-EC, the state will be converted to late-EC. Here, **object popularity** can be calculated by using Equation 1.

We use an exponential decay function [19] to record the write heat of an object. For a given object *i*, the time duration from the time when the object *i* is created to the present time is split into *k + 1* epochs, epoch *0*, ..., epoch*k*. We define the popularity of each object as follows:

$$p_k = \sum_{j=0}^{k} \frac{w_j}{2^{k-j}}, \tag{1}$$

where w_j denotes the number of writes that access the object during an epoch j. p_k denotes the write heat of the object at the end of epoch k.

Remapping A key challenge is to determine where to store the converted replicas or EC stripes after redundancy transition

to ensure a good wear balance across different flash servers. The wear balancing process uses an effective endurance-aware greedy algorithm. As shown in lines 1 to 11 of Algorithm 1, Chameleon first screens candidates whose popularity state changes from hot to cold by sorting objects based on their popularly.

During upgrade, Chameleon's greedy algorithm iteratively re-distributes the k (where k = 3) replicas of hottest candidate object to the flash servers with the lowest erasure count as shown in lines 11 to 21, the replicas of obj_i are mapped to server array $X(x_1, x_2, x_3)$. While during downgrade, the n (where $n = 6$) stripes of coldest candidate object are remapped to the flash servers with the highest erasure count as shown in lines 11 to 21, the stripes of obj_j are mapped to server array $Y(y_1, y_2, y_3, y_4, y_5, y_6)$.

To estimate the erasure count caused by a specific number of writes, we first define the erasure cost for flash memory as $1 - \mu$ according to [41], [42], where μ is the utilization of a victim block that needs to be cleaned during the 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-μ). Let B_p be the number of pages per block and W_t be the number of page writes during a certain epoch t. Then, after GC starts, the approximation for block erasure counts caused by W_t page writes during epoch t is:

$$E_t = \frac{W_t}{B_p \times (1 - \mu)} \tag{2}$$

At the end of each iteration, we estimate the new cluster wear variance σ. If $\sigma <= \sigma_{ARPT}$, ARPT will stop the iteration. To estimate the new σ, we first estimate the new erasure count of each server x in array X after re-mapping obj_i, as E_x: $E_x = E_x + E(obj_i)$, where $E(obj_i)$ can be calculated by using Equation 2. While the new erasure count of each server y in array Y after re-mapping obj_j can be estimated as E_y: $E_y = E_y + E(obj_j)$, where $E(obj_j)$ can be calculated by using Equation 2.

Algorithm 2: Hot/cold data swapping.

Require: $\sigma > \sigma_{HCDS}$
Ensure: $\sigma <= \sigma_{HCDS}$
1: **while** $\sigma > \sigma_{cHCDS}$ **do**
2: x ▷ extract server with max erase cycles
3: y ▷ extract server with min erase cycles
4: obj_i ▷ Get_hottest_candidate from x
5: obj_j ▷ Get_coldest_candidate from y
6: Map_object_to(obj_i, y)
7: Map_object_to(obj_j, x)
8: σ ▷ Estimate wear variance
9: **end while**

2) Hot/cold data swapping (HCDS): To further improve wear balance, Chameleon uses data swapping to exchange the storage location of hot replicas and cold EC stripes. As shown in Algorithm 2, Chameleon first selects two servers, server x with highest erasure cycles and server y with lowest erasure cycles. Then, the coldest object obj_i from x and the hottest objects obj_j from server y are exchanged until their erasure count difference (σ) is less than a preset threshold σ_{HCDS} (Table I).

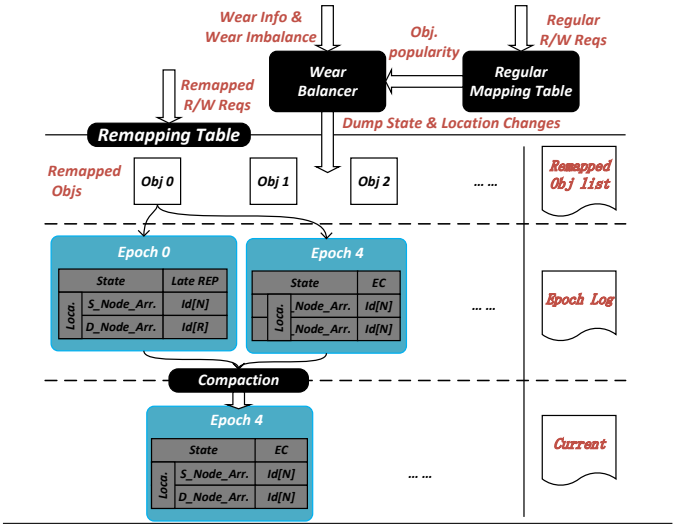


Fig. 3: Chameleon's mapping table.

After we map obj_i to server y, we estimate its new erasure count as E_y: $E_y = E_y + E(obj_i)$, where $E(obj_i)$ can be calculated by using Equation 2. Similarly, the new erasure count of server x is given by E_x: $E_x = E_x + E(obj_j)$. The data swapping process stops when the predefined erasure variance threshold $\sigma <= \sigma_{HCDS}$ is met.

Endurance-aware write offloading (EWO): To minimize the network traffic and wear balancing overhead, Chameleon offloads the writes/updates to the replicas or stripes to their destination servers instead of migrating data via bulk data transfer through the network. Incoming writes/updates trigger the data mapping change eventually.

As shown in Figure 2(b), the hot/cold data swapping module swaps the hot and cold data from the highly loaded (in terms of erasures) server to lightly loaded server using endurance-aware write offloading. There are two intermediate states: REP-EWO and EC-EWO. As seen in the figure, a replica is selected to be exchanged with a EC-stripe. It will first be converted to the intermediate state REP-EWO until the next write/update is received and the state is changed to REP. Similarly, if an encoded stripe is chosen to swap with a replica, its state is converted to EC-EWO state. The state will be eventually converted to the EC state upon next write/update.

Ideally, EWO can offload all the candidate replicas or stripes to their destination servers. However, there is rarely accessed cold data that has not been accessed for a long period. For such data, we trade off network traffic overhead with better wear balancing by migrating the cold stripes to the destination servers.

C. Mapping table

The data mapping relationship from client to flash servers is changed dynamically. Different versions of the same data can be stored on multiple different locations because of redundancy scheme conversion and data swapping. Chameleon must

ensure that reads always go to the location holding the latest version of the data. To ensure read correctness, Chameleon uses a mapping table to efficiently manage the objects whose redundancy scheme and addresses have been changed during wear balancing process as shown in Figure 3.

Source/Destination server array: Chameleon provides two levels of indirection for locating servers so as to reduce metadata overhead while maintaining read and write correctness. For a given Obj_i, the first level indirection indicates the former data host, i.e., the source server array for intermediate states, e.g. late-EC, late-REP, EC-EWO, or REP-EWO state. The second level indirection represents the object's destination server array for intermediate sates or its current host for redundancy states, e.g. REP or EC state.

Assume Chameleon monitors the cluster wear variance in a fixed time interval, denoted as **epoch**. As shown in figure 3, during epoch 0, Obj_0 is selected for redundancy policy transition from EC to REP. Obj_0's state is late-REP, which means that Chameleon will wait until an update/write request accesses Obj_0. When a write request accesses Obj_0, Chameleon directly replicates the request data and distributes its R replicas on the associated destination servers denoted as array D_node_arr and then changes Obj_0's redundancy state to REP.

For a read request, if the requested object's state is an intermediate state, e.g. late-EC, late-REP, EC-EWO, or REP-EWO state, Chameleon sends the request to the object's source server. The source server is denoted as the array $S\_Node\_arr.$ and holds the latest version of the data as shown in Figure 3. Otherwise, read requests will be sent to the object's destination servers.

Compaction As mentioned before, to reduce network traffic during the wear balancing process, Chameleon uses late-EC/REP, and EWO techniques to make a compromise between network traffic and the risk of temporary wear imbalance by waiting for an update request to the state change object. However, this wait can be for a very long time especially for cold data. Moreover, even for a hot object, there may not be an update request to such an object during an epoch as workloads are unpredictable.

As shown in Figure 3, a hot Obj_0 is selected to convert its redundancy scheme from EC to REP during epoch 0, but until epoch 3, there is still no updates to Obj_0. So in the epoch 4, Chameleon classifies the object as cold data and converts its redundancy policy from REP to EC. Chameleon creates a metadata object with version 4 for Obj_0, and appends the metadata to Obj_0's epoch log. In this case, Chameleon can keep track of each converted objects' state/location changes for failure recovery. However, epoch log would incur considerable memory overhead since epoch log increases with number of wear balancing process and the amount of involved objects.

Chameleon uses compaction to combine epoch log for each remapped object to reduce memory overhead. As shown in Figure 3, the metadata object associated with Obj_0 is updated from epoch version 0 to epoch version 4. The object's state is marked as EC since till epoch 3, there is no update to convert the state from late EC to REP. This means that Obj_0 is still encoded as stripes stored on its source destination array $S\_Node\_arr.$. Thus, in epoch 4, the object's source destination array $S\_Node\_arr.$ becomes its destination array as shown in Figure 3.

Consequently, Chameleon only maintains a single updated metadata object for the current epoch version, which not only ensures the correctness of R/W requests but also can mitigate matadata overhead.

TABLE II: SSD parameters.

Page size	4KB
Block size	256KB
Read latency	25us
Write latency	200us
Erase latency	1.5ms
Over-provisioned space	15%

IV. Evaluation

A. Implementation

We have implemented a prototype emulator of Chameleon using $16k$ lines of C++ code. We built a KV-store from scratch as a test application. We map data to servers by using a consistent hash-based data distribution algorithm that distributes data evenly across participating servers [12]. The hash function used in our experiments is FVN-a1 [43]. Each trace record maps to a logical object, which corresponds to a unique object ID calculated by using the consistent hash function. The logical object is then stored in the appropriate server by consulting the consistent hash table. We use ISA-L [14] for encoding and decoding operations. The Intel ISA-L library provides a highly optimized implementation of Reed-Solomon codes that significantly decreases the time taken for encoding and decoding operations. Specifically, we implemented our Chameleon as follows:

Flash server and flash cluster We emulate a large flash cluster by running multiple instances of our SSD simulator as flash server nodes. We use Flashsim [13] to simulate the SSD behavior as Flashsim can accurately show the block erase cycles. For all of our tests, we use an evaluation testbed with 50 flash server nodes. Each flash server node is equipped with one SSD that is simulated by FlashSim. To improve the performance especially write performance, we built a local log on top of SSD simulator. All the writes are appended to tail of the log. Table II summarizes the parameters that are commonly used to simulate SSD.

Flash monitor runs on each flash server, monitors the statistics of SSDs, and sends them to the wear balancer. We modified Flashsim by adding a flash statistics collector to the code. For connectivity between flash server nodes, we integrate Google Protocol Buffer [44] in Flashsim to facilitate communication, such as protocol parsing and messaging.

Wear balancer and mapping table are also implemented along with flash monitor on each flash server. We integrated ZooKeeper [45] in our KV-store as a distributed coordination

TABLE III: Trace characteristics.

Parameters	ycsb-zipf	mds-0	web-1	usr-0	hm-0
Reqs. cnt	1.2M	1.3M	1.3M	2.2M	4.0M
Dataset(GB)	10.4	3.1	3.8	2.5	1.9
Reqs. Data(GB)	55	44	18	194	135
Write ratio	81.1%	93.2%	76.9%	83.6%	86.6%

TABLE IV: Test schemes.

Schemes	Technique details
Chameleon	Implement two wear balancing techniques: ARPT and HCDS
EDM	Implement a data migration based wear balancing technique
REP-baseline	Apply only REP without any wear balancing technique
EC-baseline	Apply only EC without any wear balancing technique
REP+EC-baseline	Apply Hybrid REP/EC without any wear balancing technique

service. One flash server is chosen as a coordinator. The wear balancer running on the coordinator node gathers statistics of each flash server, such as the flash space utilization and erasure count by exchanging the heartbeat messages with the flash monitor running on each flash server. We installed MySQL on the flash cluster as a metadata service for storing the mapping table. Before performing wear balancing, the balancer running on the coordinator node first requests object popularity statistics from the mapping table. After wear balancing, the coordinator updates the metadata changes related to the remapped objects to the mapping table.

Client library provides a basic API to read/write the data to flash cluster and to choose between REP or EC as initial redundancy policy. For EC, the data is split into several data stripes and encoded with few parity stripes. Throughout our evaluation, we use RS (6,4) for EC (4 data stripes and 2 parity stripes) and 3-way replication for REP.

B. Experimental Methodology

1) Traces: We use two kinds of workloads for our tests: YCSB workload with Zipf-like access pattern [15] and four block-level traces from MSR-Cambridge data center servers [16]: $YCSB\_zipf$, mds_0, web_1, usr_0, and hm_0. YCSB trace is generated by YCSB benchmark which is often used to evaluate the performance of different key-value stores and cloud serving stores [15]; MSR traces are collected at the block device level from Microsoft Cambridge.Table III shows the details of trace characteristics, such as, total request count (Reqs. cnt), total dataset size (Dataset (GB)), total R/W request data (Reqs. Data(GB)), and write ratio.

2) Evaluated Schemes: We evaluated Chameleon by comparing it with multiple different schemes as shown in Table IV. To compare the state-of-art redundancy techniques, we implemented a hybrid REP/EC baseline scheme named REP+EC-baseline without using any wear balancing technique, similar to HDFS-RAID [46]. REP+EC-baseline replicates recently created data, and converts cold date from REP to EC. We also tested other two baselines schemes that applies only REP and only EC separately without using any wear leveling, denoted as REP-baseline and EC-baseline.

To compare the state-of-art wear balancing technique, we implemented and evaluated a data migration based wear balancing technique called EDM [19](detailed in II). Note that

(b) Wear variance under Chameleon and EDM

Fig. 4: Wear variance.

EDM does not consider the impact of redundancy schemes on wear balancing, so we only applied a single redundancy scheme to EDM scheme, either REP or EC.

C. Experimental Results

1) Wear Balance: To evaluate the wear variance of flash cluster, we calculate the standard deviation of the total erasure counts along with the average erasure count across 50 flash servers. Figure 4 shows the results of using three baseline redundancy schemes and two wear balancing schemes. The Y-axis shows the average erasure counts across 50 flash servers. Error bars represent one standard deviation.

First consider the results of three baseline schemes without using wear balancing algorithm as shown in Figure 4(a). Among these three baseline schemes, EC-baseline's standard deviation error bars were much smaller than that of the two baseline schemes. This is mainly because EC naturally reduces the storage overhead by eliminating redundant copies and EC can distribute data more evenly than replication since we use RS(6, 4) for EC while 3-way replication for REP. REP+EC-baseline's standard deviation errors were almost similar to that of REP-baseline. This is because REP+EC-baseline replicates all the newly created data and converts replicas to stripes only after they are cool down.

To compare Chameleon with EDM in terms of balancing the erasure count across 50 servers, we applied EC for the request data and evaluated EDM scheme and Chameleon scheme respectively. The reason we chose EC is that EC can achieve a smaller wear variance than both REP and REP+EC-baseline as shown in Figure 4(a). As shown in Figure 4(b), although EDM did improve the deviation of erasure counts, Chameleon significantly outperformed EDM under all workloads. For example, the standard deviation for the Chameleon scheme was at most ˜1,000 under workload *Hm_0* while the standard deviations were 1,880 and 2,316 for EDM and EC-baseline respectively as shown in Figure 4(b). For the two workloads, *Web_1* and

(b) flash endurance under EDM and Chameleon

Fig. 5: Flash endurance.

(b) SSD write latency under EDM and Chameleon (normalized to that of Chameleon).

Fig. 6: Write latency.

YCSB_zipf, which exhibit relatively smaller standard deviation error bars compared to others, Chameleon also delivered a better wear balance compared with EDM. In particular, its standard deviations were 162 and 704 under workloads *Web_1* and *YCSB_zipf* respectively while that of the EDM were 190 and 876, respectively.

Overall, Chameleon can reduce wear variance by 52% on average and at-most 81%, compared to EC-baseline. Chameleon can reduce the wear variance by 43% on average and at-most 70%, compared to EDM.

2) Flash endurance: To evaluate flash endurance, we calculate the aggregate erase cycles for all flash servers. The results are shown in Figure 5. The Y-axis shows cluster-wise total erasure counts. As shown, the total erase cycles when replaying the workload *web_1* is relatively lower than that when replaying others. This is because that workload *web_1* has lower amount of write request data than other workloads as shown in Table III.

As shown in Figure 5(a), we observe that among three redundancy policies, REP (shown as REP-baseline) has more erasure count than other two redundancy policies because REP writes almost $3\times$ more data to the whole cluster and entails more erasure count. While EC-baseline has the lowest erasure count since it consumes less storage than REP. The total erasure count of REP-baseline is ˜$2\times$ higher than that of EC-baseline.

To compare Chameleon with EDM about their compact on flash endurance across 50 servers, we applied EC for the request data and evaluated EDM scheme and Chameleon scheme, respectively, since EC can achieve a smaller total erasure count than other two redundancy policies, REP and REP+EC-baseline. Comparing Chameleon with EC-baseline scheme, we observe that Chameleon has a similar amount of cluster-wise aggregate block erase cycles with EC-baseline scheme while EDM has a significant higher total erasure count than both Chameleon and EC-baseline as shown in Figure 5(b). This is because Chameleon introduces less writes to the destination servers compared with EDM by using late EC/REP and EWO techniques. For EDM, the data migration process introduces significant wear overhead to the flash cluster due to extra write overhead. As we can see in Figure 5(b), the block erasure count of EDM is increased by up to ˜20% under workload Usr_0, Mds_0, and Hm_0 due to data migration.

3) Impact on SSD write latency: We measured the average write response time in each SSD simulator to see the impact of wear balancing on write performance as GC has a significant influence on write performance as shown in Figure 6. Note that the write latency is measured as the time interval between SSD simulator receiving a write request and finishing the write request. Y-axis shows normalized write latency.

The write latency normalized to REP-baseline are shown in Figure 6(a). As shown, the average write response time when replaying the workload *Web_1* is relatively lower than that when replaying others. This is because that workload *Web_1* have lower amount of write request data than others as shown in Table III. Moreover, EC-Baseline's average write response time is the highest among three redundancy schemes. The write latency of EC-baseline is 1.12 on average and at-most 1.35 higher than that of REP-baseline. This is because, under EC, the writes are scattered across multiple servers (e.g., 6 in RS-(6,4)) at a smaller stripe granularity, while REP performs writes at a bigger object-level and therefore has a higher sequentiality of writes. With increasing sequentiality of writes (Figure 6(a)), the write performance of SSDs is observed to be improved.

To compare Chameleon with EDM with respect to impact on write performance, we applied REP for the request data and evaluated EDM scheme and Chameleon scheme since REP can achieve a better write performance than both EC and REP+EC-baseline. Compared to EDM, Chameleon has

(b) Write amplification under EDM and Chameleon.

Fig. 7: Write amplification.

a better write performance. Chameleon can reduce the write latency by 25% on average and at-most 33%, compared to REP-baseline. In contrast, EDM can only reduce the write latency by 7% on average and at-most 20%, compared to REP-baseline. This is because Chameleon can achieve a good wear balance with minimum extra overhead. In contrast, EDM introduces considerable extra overhead during wear balancing process.

4) Impact on write amplification: We measured the write amplification (WA) after GC starts in each SSD simulator to see the impact of wear balancing on write amplification. The results are shown in Figure 7. Y-axis shows the write amplification.

EC-Baseline's WA is the highest among three redundancy schemes as shown in Figure 7(a). We compare the WAs of EC-baseline and REP-baseline. The WA of EC-baseline is 2.11 on average and at-most 2.8, while that of REP-baseline is 1.4 on average and 1.7 at-most. The reason is the same as that of write latency: under REP, the writes have higher sequentiality because REP performs writes at a bigger object-level while the writes are scattered across multiple servers (e.g., 6 in RS-(6,4)) at a smaller stripe granularity under EC. With increasing sequentiality of writes (Figure 7(a)), the WA of SSDs improved.

Theoretically, write amplification can be defined as $1/(1-\mu)$, where μ is the utilization of victim block that needs to be cleaned during GC. That is, to make room for (1-μ) new writes, μ valid pages need to be rewritten so the total number of writes is $(1-\mu)+\mu = 1$. Thus, write amplification is directly affected by the victim block utilization. Redundancy policy impacts victim blocks utilization by changing the size and destination of write requests. Moreover, the relationship between redundancy policy and write amplification is not a simple linear relationship as shown in Figure 7(a).

To compare Chameleon with EDM about the compact on WA, we applied REP for the request data and evaluated EDM scheme and Chameleon scheme since REP can achieve a lower WA than both EC and REP+EC-baseline. Compared with EDM, Chameleon has a lower WA. Chameleon can reduce the WA by 12% on average and at-most 20%, compared to REP-baseline. While EDM can only reduce the WA by 6% on average and at-most 13%, compared to REP-baseline. There are several reasons for this behavior. First, Chameleon achieves a better wear balance distribution, which mitigates the overall write amplification due to garbage collection. For one hand, the utilization of hot flash servers is reduced. For the other hand, Chameleon introduces less writes to the destination servers compared with EDM.

Fig. 8: Data state changes over 85 hours under Chameleon by replaying YCSB-zipf workload.

5) Data state changes over time: As discussed in Section III, Chameleon achieves better wear balance by using two adaptive wear balancing techniques, ARPT and HCDS. ARPT converts data redundancy polices while HCDS off-loads data via EWO. Consequently, data has two redundancy states, REP and EC, and four intermediate states, Late REP, Late EC, REP-EWO, and EC-EWO.

To see how the data state changes over time, we calculated the aggregate amount of data in different states individually for each hour. Figure 8 shows data state changes over 85 hours by replaying workload $YCSB-zipf$. Y-axis shows the percentage of data in different states. As shown, We combine the REP-EWO and EC-EWO together as EWO state since the amount of data in REP-EWO state is roughly similar to that of data in EC-EWO.

First, all the data started with EC state since we applied EC for all newly created data and after three hours, ARPT started to convert a small mount of hot data from EC to late REP and later cover to REP when their update requests come. During the 5th hour, Chameleon detected that wear imbalance happened and started HCDS to swap hot data with cold data for wear balancing. After that we see the data in EWO state increased up to 20% during the 20th hour and then fluctuated during the period of the 25th-65th hour. During this period, up to 20% of data was involved in HCDS for wear balancing. The slight decrease during this period means that a certain amount of data was offloaded and covered to a redundancy state.

After the 65th hour, we see a decrease for the amount of data in EWO state, implying that not only the wear but also the workload were almost balanced. Moreover, the data involved in HCDS was almost converted from a intermediate EWO state to a finial redundancy state.

Overall, we can see the data involved in HCDS was less

than 20% for each hour, while the data involved in ARPT was less than 5% for each hour. We conclude that only a relative small amount of data's popularity changes and HCDS plays a major role in wear balancing.

V. Conclusion

We have presented the design and implementation of Chameleon, a wear balancer for distributed flash-based storage cluster. Chameleon aims to improve both the flash endurance and runtime performance. First, Chameleon takes advantages of the out-of-place update feature of flash memory by directly offloading the writes/updates across flash servers instead of moving data across flash servers to mitigate extra-wear cost. We implemented several optimizations to this end: late replicating (Late REP), late encoding (Late EC), and endurance aware write offloading (EWO). Second, Chameleon provides two adaptive wear balancing algorithms, namely, redundancy policy transition (ARPT) and Hot/Cold data swapping (HCDS) to balance the wear distribution across the flash servers, coupled with write offloading and redundancy policies to balance the erasure count and improve both lifetime and performance. Evaluation shows that Chameleon reduces the wear distribution deviation by up to 81%, while improving the write performance by up to 33%. In the future, we aim to realize Chameleon in real flash hardware such as Open-channel SSD [38] and integrate Chameleon to other distributed storage types such as distributed file systems.

Acknowledgments This work is sponsored by the NSF under the grants: CNS-1565314, CNS-1405697, and CNS-1615411. The work performed at Temple is partially sponsored by NSF grants CCF-1547804, CNS-1702474, and CCF-1717660. The work at HUST is sponsored by National Natural Science Foundation of China under grant number 61472152, and 6143200.

References

[1] B. Debnath, S. Sengupta, and J. Li, "Flashstore: high throughput persistent key-value store," *VLDB '10*.

[2] H. Sim, Y. Kim, S. S. Vazhkudai, D. Tiwari, A. Anwar, A. R. Butt, and L. Ramakrishnan, "Analyzethis: an analysis workflow-aware storage system," in *ACM/IEEE SC '15*.

[3] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "Fawn: A fast array of wimpy nodes," in *SOSP '09*.

[4] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu *et al.*, "Bluedbm: Distributed flash storage for big data analytics," *TOCS '16*.

[5] A. M. Caulfield and S. Swanson, "Quicksan: a storage area network for fast, distributed, solid state disks," in *ACM SIGARCH Computer Architecture News '13*.

[6] M. Balakrishnan, D. Malkhi, J. D. Davis, V. Prabhakaran, M. Wei, and T. Wobber, "Corfu: A distributed shared log," *ACM TOCS '13*.

[7] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," *SIGMETRICS Perform. Eval. Rev.*

[8] "Summit (ornl)," https://www.olcf.ornl.gov/summit.

[9] M. Xia, M. Saxena, M. Blaum, and D. A. Pease, "A tale of two erasure codes in hdfs," in *USENIX FAST '15*.

[10] H. Ohtsuji and O. Tatebe, "Server-side efficient parity generation for cluster-wide raid system," in *IEEE CloudCom '15*.

[11] P. Cao, S. B. Lin, S. Venkataraman, and J. Wilkes, "The tickertaip parallel raid architecture," *ACM Trans. Comput. Syst.*

[12] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *ACM STOC '97*.

[13] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar, "Flashsim: A simulator for nand flash-based solid-state drives," in *Advances in System Simulation '09*.

[14] "Intel ISA-L," https://github.com/01org/isa-l.

[15] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *ACM SOCC '10*.

[16] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *Trans. Storage '08*.

[17] Y. Cheng, A. Gupta, A. Povzner, and A. R. Butt, "High performance in-memory caching through flexible fine-grained services," in *ACM SOCC '13*.

[18] Y. Cheng, A. Gupta, and A. R. Butt, "An in-memory object caching framework with adaptive load balancing," in *ACM EuroSys '15*.

[19] J. Ou, J. Shu, Y. Lu, L. Yi, and W. Wang, "Edm: An endurance-aware data migration scheme for load balancing in ssd storage clusters," in *IEEE IPDPS '14*.

[20] S. Boboila and P. Desnoyers, "Write endurance in flash drives: Measurements and analysis." in *USENIX FAST '10*.

[21] Y. Cheng, F. Douglis, P. Shilane, G. Wallace, P. Desnoyers, and K. Li, "Erasing belady's limitations: In search of flash cache offline optimality," in *USENIX ATC '16*.

[22] X. Jimenez, D. Novo, and P. Ienne, "Wear unleveling: improving nand flash lifetime by balancing page endurance," in *USENIX FAST '14*.

[23] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber, "Extending ssd lifetimes with disk-based write caches." in *USENIX FAST '10*.

[24] D. Ma, J. Feng, and G. Li, "Lazyftl: a page-level flash translation layer optimized for nand flash memory," in *ACM SIGMOD '11*.

[25] D. Jung, Y.-H. Chae, H. Jo, J.-S. Kim, and J. Lee, "A group-based wear-leveling algorithm for large-capacity flash memory storage systems," in *ACM CASES '07*.

[26] L.-P. Chang, "On efficient wear leveling for large-scale flash-memory storage systems," in *ACM SAC '07*.

[27] M. Murugan and D. H. Du, "Rejuvenator: A static wear leveling algorithm for nand flash memory with minimized overhead," in *IEEE MSST '11*.

[28] X. Jimenez and D. Novo, "Phoenix: reviving mlc blocks as slc to extend nand flash devices lifetime." in *ACM DATE '13*.

[29] K. M. Greenan, D. D. Long, E. L. Miller, T. Schwarz, and A. Wildani, "Building flexible, fault-tolerant flash-based storage systems," in *USENIX HotDep '09*.

[30] J. Ou, J. Shu, Y. Lu, L. Yi, and W. Wang, "Edm: An endurance-aware data migration scheme for load balancing in ssd storage clusters," in *IEEE IPDPS '14*.

[31] W. Wang, T. Xie, and A. Sharma, "Swans: An interdisk wear-leveling strategy for raid-0 structured ssd arrays," *ACM TOS '16*.

[32] A. Anwar, Y. Cheng, A. Gupta, and A. R. Butt, "Mos: Workload-aware elasticity for cloud object stores," in *ACM HPDC '16*.

[33] K. Krish, A. Anwar, and A. R. Butt, "hats: A heterogeneity-aware tiered storage for hadoop," in *IEEE CCGrid '14*.

[34] Y. Cheng, M. S. Iqbal, A. Gupta, and A. R. Butt, "Cast: Tiering storage for data analytics in the cloud," in *ACM HPDC '15*.

[35] ——, "Pricing games for hybrid object stores in the cloud: Provider vs. tenant," in *USENIX HotCloud '15*.

[36] A. Anwar, Y. Cheng, A. Gupta, and A. R. Butt, "Taming the cloud object storage with mos," in *ACM PDSW '15*.

[37] A. Anwar, Y. Cheng, H. Huang, and A. R. Butt, "Clusteron: Building highly configurable and reusable clustered data services using simple data nodes." in *USENIX HotStorage '16*.

[38] "Open-channel ssd," https://openchannelssd.readthedocs.io/en/latest/.

[39] S. Lee, M. Liu, S. W. Jun, S. Xu, J. Kim, and A. Arvind, "Application-managed flash." in *USENIX FAST '16*.

[40] M. Bjørling, J. Gonzalez, and P. Bonnet, "Lightnvm: The linux open-channel SSD subsystem," in *USENIX FAST '17*.

[41] H.-j. Kim and S.-g. Lee, "A new flash memory management for flash storage system," in *IEEE COMPSAC '99*.

[42] L.-P. Chang and T.-W. Kuo, "An efficient management scheme for large-scale flash-memory storage systems," in *ACM SAC '04*.

[43] "FVN-a1 Hash," http://isthe.com/chongo/tech/comp/fnv.

[44] "Protocol Buffers," https://github.com/google/protobuf.

[45] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *USENIX ATC '10*.

[46] D. Borthakur, R. Schmidt, R. Vadali, S. Chen, and P. Kling, "Hdfs raid," in *Hadoop User Group Meeting*, 2010.